Using Solution Matrices for Provably Correct Polygon Algorithms

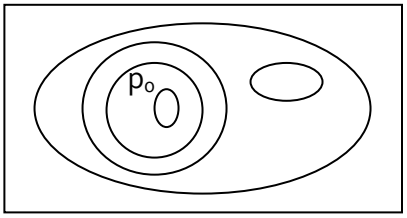
By Tom Wright,
Retired

keywords: polygon comparison, polygon set operations, polygon clipping, winged-edge structures, provably correct algorithms

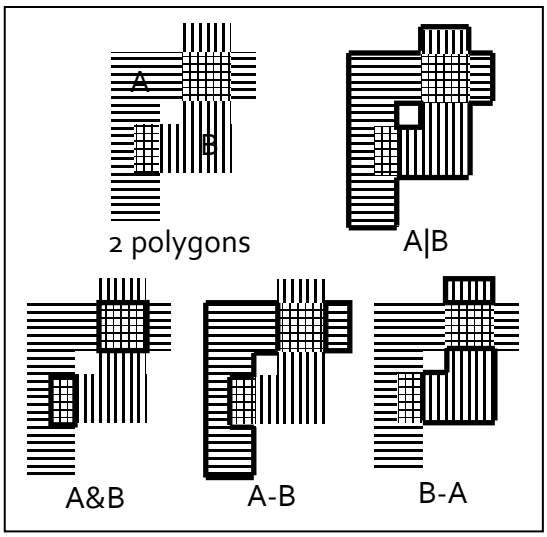
Abstract: Starting from a simple foolproof way of relating a point to a line, a provably correct algorithm is developed to perfectly classify the relationship between two line segments. Similarly, a provably correct algorithm is developed to sort rays in direction order. Both are based on a quick measure that divides the problem into a modest number (<100) of cases that can be solved once and stored. These two algorithms form the basis for algorithms for classically troublesome computational problems in computer graphics: determining if a point is inside a polygon and forming the sum, difference and intersection of two polygons.

1. Introduction

Some classic problems in computer graphics or computational geometry are to detect if a point is inside a polygon and to compare two polygons, computing their union, intersection and difference (below). While these problems have been solved theoretically for a very long time, efficient, robust algorithms have proved elusive. There are problems with round-off error, lines grazing vertices, infinite slopes, and so on. In building a polygon compare algorithm, solutions were found where complex situations could be quickly classified and the solution found in a matrix prepared beforehand. Since it can be proved that the classification step covered all possibilities and it can be shown that the solution for each classification is correct, the algorithms are provably correct.



Is p_o inside?



Of the three important low-level algorithms, two used solution matrix techniques. The three were 1) perfectly comparing a point with a line, 2) perfectly finding the relationship between two line segments and 3) perfectly sorting rays emanating from a point according to their angle from a reference line. From these it was relatively straightforward to formulate algorithms for finding if a point is in a polygon, finding intersections between edges of two polygons, and traversing combined edges of two polygons to generate the union, intersection and differences. Additionally, (somewhat surprisingly) standard polygon comparison techniques found a new application in an algorithm to correct illegal figure 8 constructs in input polygons.

Solution matrices, also called state tables, can be used for all sorts of computational problems. Suitable problems have a straightforward way of quickly classifying a particular set of data into categories. Each category has a solution (or potentially a set of instructions) stored for quick access. The time to compute the

solutions for the matrix is unimportant because they need only be computed once for all time.

Here a polygon is an ordered set of vertices and the edges that connect them (rather than, for instance, a set of pixels). Since graphical devices are almost exclusively raster based, polygon operations based on vertices and edges are for computational rather than display purposes. Converting vertices and edges to a set of pixels (polygon fill) has been solved for decades and will not be discussed further.

Other problems related to polygons are useful but can prove computationally difficult, such as the problems addressed here. Agathos *et al* [1] reviews the point in polygon problem. Weiler [7] reviews the algorithm for generating the comparisons of two polygons. The algorithm was based on an idea of having "winged edge" data structures for intersections put forth by Bungatt *et al* [3] to direct traversing the polygon edges to generate the set comparisons of two polygons. The difficulty is in the details of creating these structures and details of traversing them.

An example of the usefulness of polygon comparison, property boundaries are defined as the edges connecting vertices that are points on the earth. Two properties may theoretically share an edge without necessarily sharing any vertices. Are the edges coincident? Are there any disputed areas where the properties overlap? Are there areas between the two? Scale up to international borders and the questions become crucial. Polygon comparison has application to some CAD applications as well.

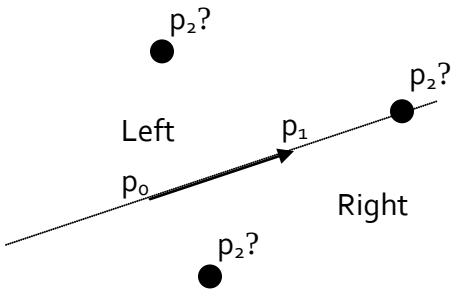
Leonov [5] reviews available implementations and notes "Most of the programs listed above are not strictly robust and use floating-point arithmetic with some tolerance values." For other programs "required memory size grows exponentially" and "not to be satisfactory for practical applications." Only one of the latter is considered robust. The new algorithms presented here, in contrast, have linear storage needs and are totally robust.

Polygon set operations can be used for polygon clipping. Azevedo *et al* [2] and Greiner *et al* [4] present algorithms for polygon clipping. Performance comparisons would be interesting if like implementations were available.

Schirra [6] reviews the reliability of testing a point for being in a polygon and finds various algorithms lacking in robustness for difficult cases. The algorithm discussed here cannot have these problems because it is provably correct.

2. Comparing a Point to a Line

This solution is fairly obvious and is presented only to show that following algorithms are built on a solid foundation. Given a line defined by two points p_0 and p_1 and another point p_2 where is the point? The answers could be 1) when at p_0 looking at p_1 the point p_2 is to the left, 2) p_2 is to the right or 3) p_2 is along the extended line. By "along the extended line" we mean ***exactly***. That is, the point is an exact solution for the equation for the line.



Throughout this paper we shall assume points are integers. In the code I have written they are short integers so that, barring deliberately pathological data, a variety of operations can be undertaken using long integers without fear of run-time errors such as overflow.

There is a well-known formula for computing the area of a polygon with *n* vertices [8]:

$$\text{Area} = \tfrac{1}{2} \sum x_i y_{i+1} - x_{i+1} y_i \quad \text{for } i = 0,\ldots,n-1$$

Note that throughout this article for all polygons the last point is not equal to the first. That is, there is an assumed segment from p_{n-1} to p_0. Also, when $i+1 = n$ in the above equation, 0 is used.

Importantly, in the above equation when the points are counterclockwise (CCW), the result is positive. The same points arranged clockwise (CW) give a negative result. And of course if the points are all collinear, the result is zero.

We can compute the area of the triangle formed by the line and the point using this formula. Further, if we normalize the triangle so p_0 is at {0,0}, two of the three terms in the summation become zero, so that

$$\text{Triangle Area} = \tfrac{1}{2} ((x_1-x_0)(y_2-y_0) - (x_2-x_0)(y_1-y_0))$$

If we compute twice the area, there is no division and no round-off error is possible. Algorithmically, if the result is positive, we return "left", negative we return "right" and zero we return "center" (colinear). Although the code is obvious, it is presented because a routine here is used later below. The types are explained in Appendix A. Note that if one term in an expression is long, the other terms are made long.

```
long TriangleAreaX2Signed(Point p0,Point p1,Point p2) {
   // compute twice the area of a triangle
   return ((long)p1.x-p0.x)*((long)p2.y-p0.y)-((long)p2.x-p0.x)*((long)p1.y-p0.y);
}
PointReLine WherePointReLine(Point lineBgn,Point lineEnd,Point p) {
   // find if a point is left or right of a line or on its extension
   long areaX2 = TriangleAreaX2Signed(lineBgn,lineEnd,p);
   return (areaX2>0)? Lft : (areaX2<0)? Rgt : Ctr;
}
```

Because the points were short integers, we can do the calculation using long integers and nothing can overflow (barring perverse data). We can see that this calculation takes five subtractions and two multiplies. For counting operations, addition and subtraction are considered the same thing, so we would say it takes two multiplies and five additions. The tests are each assumed to take a subtraction, so the total is two multiplies and seven additions. We have yet to use a solution matrix, but now we are ready.

3. Relating Two Lines

Given line A = p_0p_1 and B = p_2p_3, how do the line segments (rather than extended lines) relate? That is, do they miss completely? Do the cross? Does one point exactly touch the other line? Do the lines overlap? Since this algorithm will be used many times for the polygon algorithms, it needs to be both very fast and perfectly accurate. A straightforward algorithm to answer this question would likely require solving simultaneous equations (introducing round-off error) or involve special cases to avoid infinite slopes.

There is a better option. If we use the algorithm in Section 2 to relate p_0 to B we have three possible answers: left, right and center (collinear). Similarly we can relate p_1 to B, then p_2 to A and finally p_3 to A. Taken all together, we then have 3 x 3 x 3 x 3 or 81 possibilities. By examination we can solve the 81 cases. I used strips of paper for A and B with each point and left versus right marked so I wouldn't be too confused. Even so, initially I made a few errors. Some errors were found by flattening the 3 x 3 x 3 x 3 matrix to 9 x 9 and looking at the symmetries. The last error was found by testing. A proof could be created by carefully describing each of the 81 situations and what each implies. Note that some of the matrix elements are impossible situations and are so marked even though they are never used. The algorithm to relate two lines using this technique follows:

```
TwoLinesReltnshp TwoLinesRelated(Point p0,Point p1,Point p2,Point p3) {
   // find the exact relationship between 2 line segments, p0p1 & p2p3
   PointReLine side0 = WherePointReLine(p2,p3,p0);
   PointReLine side1 = WherePointReLine(p2,p3,p1);
   PointReLine side2 = WherePointReLine(p0,p1,p2);
   PointReLine side3 = WherePointReLine(p0,p1,p3);
   return twoLnsRel[side0][side1][side2][side3];
}
```

The variable "twoLnsRel" is the solution matrix, filled elsewhere with the solutions generated by hand. It contains values corresponding to "miss", "p_o exactly touches B", and so on. Note that "p_o exactly touches B" means only that happens. (See Appendix B.) As we saw above, the first four statements take eight multiplies and 28 additions. Computing the subscript takes three multiplies and three additions, so the total is 11 multiplies and 31 additions. This proved to be a very fast algorithm for perfectly relating any two arbitrary line segments.

Of the 81 cases, one, the collinear case, requires some post-processing in some contexts. If we need to see where on the collinear line the two segments reside, this can be done with a few simple foolproof comparisons.

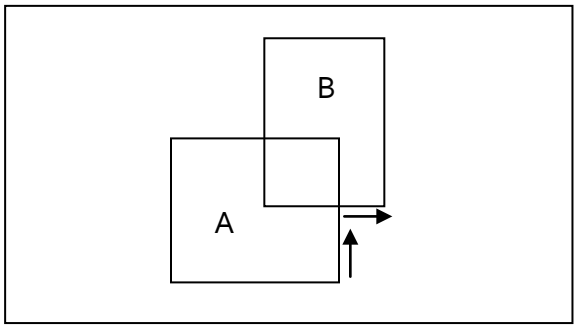
Because the 81 cases span all possibilities and each can be proved correct by examination, the algorithm is provably correct.

This algorithm is used to compare the line segments in the two polygons. If a segment of one polygon intersects the other polygon at other than a vertex, the intersection is computed and a point is added there to each polygon if needed.

4. Sorting Rays

As mentioned above, Bungatt *et al* [3] back in the 1970's worked out a technique for forming polygon operations "and", "or" and "minus." Perhaps the easiest way to understand is with an example.

Consider two simple rectangular polygons A and B that partially overlap by one corner of each. By convention here, the points are stored in CCW order. If you start on a point of A not inside B, and if you traverse A CCW until you find an intersection and then "bear right", you will trace out A|B, which is "A or B", otherwise known as the "union of A and B". The winged edge data structure was the repository of which segments were for bearing left (used below) and right.

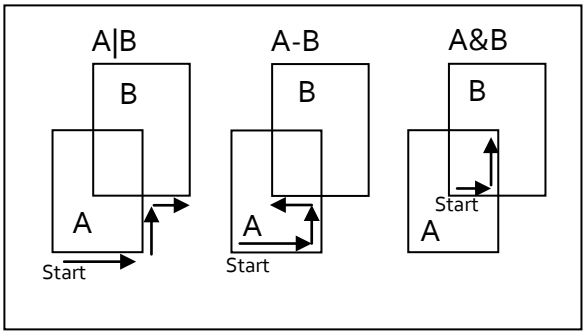


To be sure of finding all of the A|B outlines, every point of A not in B should be considered as a possible starting point, so while following an outline, each point visited must be marked as not a starting point. Also, points of B outside A must be considered as possible starting points. Simple examination shows this set of instructions produced correct results if A and B are separate and also if A or B is contained inside the other. Baring right also works in holes because they are CW. When done, the polylines generated must be nested and have their directions corrected.

To summarize the rules:

- A|B: all points of A outside B should be considered for starting traces; also points of B outside of A; bear right at intersections.
- A-B: only points of A outside B should be considered; bear left.

- A&B: all points of A inside B should be considered; also points of B inside A; bear left.



Why do we want to sort rays? The problem in bearing right is deciding which edge segment to follow when exiting the intersection in order to bear right. Let's just look at the edge segments of both polygons that have a vertex at the intersection. (Remember, if for each polygon there is no vertex at the intersection, we add one.) We can consider them to be rays emanating from the intersection. We arrived at the intersection on one of the rays, and we want to find which one allows us to bear right and continue following edges. This can be done via sorting. For each intersection we store a description of the line segments that touch the intersection. If we create a sorting of these rays in CCW order (the first ray being arbitrarily selected), then to bear right we figure which ray was the one we arrived on, and leave on the next one in the CCW sorting. To trace A-B, bear left by following the next previous ray in the CCW sorting. To trace A&B, start at a point of B inside A and bear left.

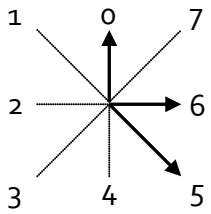
To a pure mathematician, putting the rays in CCW order would not even be considered a problem, but producing a computer algorithm for this gets quite tricky. Computing angles involves slow trig functions and using slopes involves watching out for lines that could cause division by zero, and that takes additional effort. And how can it be known that testing is adequate? A solution matrix solves all these problems, although the right situation to apply one has to be found. In a previous attempt at this problem over two decades ago I attempted to form a solution matrix that characterized all four rays about an intersection and found it too difficult to make it work. Also, in general there can be more than four rays when more than two edges intersect at a single point. I was so stuck I abandoned the problem. Recently (a mere quarter century later) I suddenly thought of a new way to look at the problem. By considering it as a sorting problem, we only need to look at three rays at once, the reference ray and the two to be compared. Such comparisons suggested using another solution matrix. (Why this suddenly occurred to me 25 years later remains a mystery.)

In practice, it was found that the collinear case of the point-line comparison above had to be divided into two cases: in the same direction and in the opposite direction. This is always possible because the rays all share a common point. The code to expand WherePointReLine into a second version returning with the extra case is obvious but is presented here for completeness.

```
#define INORDER(a,b,c) (((a)<(b)&&(b)<(c)) || ((a)>(b)&&(b)>(c)))
PointReLine WherePointReLine2(Point p0,Point p1,Point p2) {
   // used for rays; find if a point is left or right of, or on, a line
   // then split the collinear case into same direction or opposite
   PointReLine wprl = WherePointReLine(p0,p1,p2);
   if (wprl==Ctr) {
      if (p0.x==p1.x)
         wprl = INORDER(p1.y,p0.y,p2.y)? Opp : wprl;
      else
         wprl = INORDER(p1.x,p0.x,p2.x)? Opp : wprl;
   }
   return wprl;
}
```

Next, assume we have a reference line, $L_0 = p_0p_1$, and two lines, $L_1 = p_0p_2$ and $L_2 = p_0p_3$.

We can use the revised point to line comparison for L_0 and p2, L_0 and p3, and L_1 and p3. This gives us 4 x 4 x 4 possibilities. This time I chose a different method to fill the solution matrix rather than doing it by hand and testing. Imagine a sort of clock with three "hour" hands and eight hours. The imaginary clock's center is at {0,0}; Zero hour is at {0,1}; One is at {-1,1}; Two is at {-1,0} and so on. Note that our clock runs CCW! The clock is defined by an array with the eight points ending with the last point at {1,1}. The "hour" equals the index to the array of points and they equal the angle of the hand in units that equal -45 degrees from "up".



We start with the solution matrix filled with the value for "impossible." We leave the first hand at zero. This will be our fixed reference line. With nested loops we march the other two hands through the 64 possibilities. (This 64 is 8 x 8 for the different hours or angles, not 4 x 4 x 4 from above.) We compute the three comparisons above. We know how the angles compare by comparing the indices used to select the endpoint coordinates from the clock definition. An element in the solution matrix is set by using the three comparisons as indices and the angle comparison as the value. The same element can be set multiple times, but it always gets set to the same value. (See Appendix C.) Although not slow, the efficiency of this process is immaterial because it is run only once in history and the saved solutions can be used forever.

Given the saved matrix, the two-ray comparison to a reference ray code follows:

```
LessCCW LessCounterClockwise(Point p0,Point p1,Point p2,Point p3) {
   // given a line p0 to p1, is p0 to p2 is less CCW than p0 to p3?
   PointReLine re12 = WherePointReLine2(p0,p1,p2);
   PointReLine re13 = WherePointReLine2(p0,p1,p3);
   PointReLine re23 = WherePointReLine2(p0,p2,p3);
   return lessCCW[re12][re13][re23];
}
```

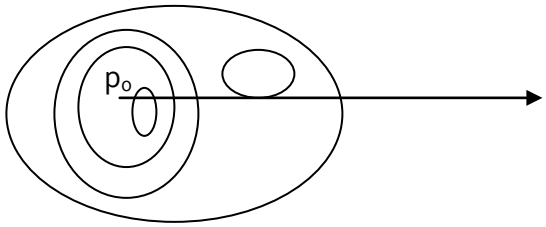
Ties require a bit of attention. A tie means that the two polygons share some edges. Except in one special case (identical polygons), we know they will diverge. Ties can be dealt with when sorting by following either path until they diverge and set the sorting back at the first intersection to the situation when they part ways. It may seem like you can say it is a tie and merely traverse either line until they diverge, but we have to mark each vertex as we traverse to prevent repeating outlines. If the selection of which polygon to follow is arbitrary, in certain pathological situations we may have to traverse back along this same path. We could be unlucky and erroneously think we already traversed somewhere that it is still needed. So rather than say it is a tie and leave finding the divergence later in the traversal, we pause the main traversal and do a quick special traversal only looking for the divergence and reporting the state of the original vertex as the state where the coincident edges diverged.

Now we have the tools needed to attack the higher level problems.

5. Notes on "Point Inside a Polygon?"

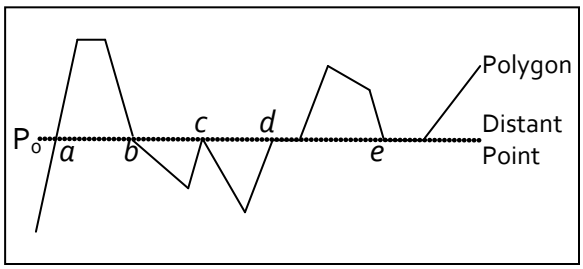
Finding whether a point is inside, outside, on an edge or on a vertex of a polygon is a classic problem in the field of computational computer graphics or computational geometry, what might be labeled a user level function in a software

library. It is also needed as a low level utility function for various higher-level functions such as properly nesting tracings of polygons with holes, islands within the holes and so on.



Is p_o inside?

A classic algorithm for finding if a point is inside a polygon is to draw a line to a distant point and count the crossings. An odd number means the point is inside. The details present some problems that must be dealt with.



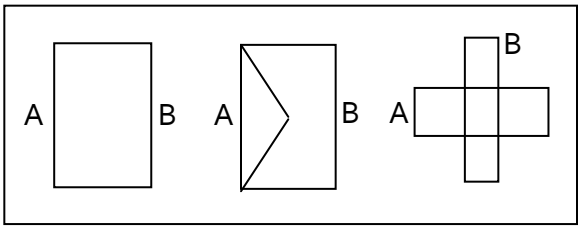
Crossing point ***a*** is the simplest, two segments crossing with no vertices involved. Recall that we add points as needed for intersections so situation ***a*** becomes the same as ***b***. Cross ***b*** has a crossing that could accidentally be counted twice. Fortunately, our algorithm for relating two lines exactly makes this problem easy to avoid because it identifies the exact situation for each segment and we can count only intersections at, say, ending points of polygon line segments and not beginnings. (This refers to going CCW around the polygon.) Intersection ***c*** must be recognized as a bounce, which is not a crossing. We can detect this by sorting the rays as above. And each extended crossing and extended bounce such as ***d*** and ***e*** must be recognized. Unfortunately, any of these can involve the implied segment from p_{n-1} to p_0, requiring careful coding. Taking these problems into account it is fairly straightforward to code an algorithm that can handle any situation and yield the correct result.

Since ***a*** through ***e*** cover all the cases of what can happen when a line meets a polygon edge, it is possible to prove the correctness of an "inside checking" algorithm.

Besides being a problem worth solving in its own right, this is needed for solving polygon comparisons as well.

6. Starting Various Tracings and Arranging Them

For the two partially overlapping rectangles in the example in Section 4, starting with a point of A outside B was sufficient to trace A|B. In general, we would also have to try starting at a point of B outside A to complete a number of cases, one of which is when A and B are separate. And because there can be independent tracings for parts of complex polygons, every point on every polygon must be checked as a possible starting point for a tracing. Unfortunately, even doing both of these, starting with a point of A outside B and starting with a point of A outside B for every vertex does not assure finding all of A|B.



In the left example A and B have the same outline. No point of A is outside B and no point of B is outside of A.

The second figure, a rectangle, A, and a rectangle with one bent edge, B, shows that unequal polygons can also lack external points for starting A|B.

A&B requires starting on a point on A inside of B and on B inside of A. The third figure shows this cannot always be done.

To overcome these problems, a little preprocessing is used. After any needed intersections are added as described above, all the coordinates are multiplied by 2 and wherever a polygon has two intersection points in succession, a point is added at the middle of the segment. Now all the problems above vanish except for identical polygons, which can be identified and treated as a special case.

To prevent retracing the same outlines from various starting points, a variable is associated with each vertex to record if it has been traced for the current operation. Thus, the type definition for Point in Appendix A is a simplification. Also, if the vertex is an intersection it references a structure where sorted rays are stored.

At this point it should be noted that it is impractical to restrict an implementation to simple polygons (one outline with no holes) because the polygon operations can all produce complex polygons with children (holes and islands in holes and so on) and siblings, that is, polygons in multiple parts. (It is possible to define holes as parts connected to the parent or sibling with two coincident line segments, one going in and one going out, but this results in many difficulties and the inability to render the outlines correctly.) The tracings for a given operation such as A|B are just a set of outlines not properly arranged. To create a correct answer it is necessary to create a nesting algorithm. This is not difficult given the algorithm from Section 5. Note that because these are traces, nesting is fairly simple since we know if any point on a trace is inside another trace, the whole polygon is inside (although some edges might be shared, which doesn't matter). The same goes for outside. Besides nesting the polygons, the nesting algorithm reverses the vertex order if need be to conform to the rules about CCW outlines and CW holes.

7. Pre-Cleaning

Depending on the source of the data, "pre-cleaning" steps could include removing repeated points, removing the middle of collinear points, removing degenerate polygons (zero area, less than three points) and dealing with certain problematic self-intersecting polygons. Once the outlines are cleaned, they must be nested and the order of the points may need to be reversed. All of these are straightforward except possibly dealing with self-intersecting polygons.

Self-intersecting polygons can break the rule of keeping the interior on the left. The traversing algorithm can be modified to help assure input data conforms to the rules about CCW versus CW and nesting. Specifically, a figure 8 with the points ordered the way most people draw an 8 does not conform to our definition of a well formed polygon because while traversing the points the interior isn't always be on the left. We need either to convert the 8 to two polygons that touch or one polygon where following the points in order causes us to "bounce" off the intersection.

To clean up a self-crossing polygon, we use these steps for each polygon treated independently. That is, we do siblings, holes and islands each by themselves:

1. Search the polygon for a self-crossing
2. Add intersection vertex if needed
3. Build a sorted list of all edges touching the intersection to use for traversing
4. Try every point as a possible starting point, traverse and bear left at intersections, marking each vertex traversed as not a possible starting point
5. Unmark all the vertices and repeat Step 4 except bear right
6. The resulting three traversals will be two short polygons and one longer one; the two short ones can be used to split the 8 into two o's while the long one is the 8 with no edges crossing, only vertices touching; substitute the desired solution for the original polygon
7. Repeat the above steps until there are no self-crossings

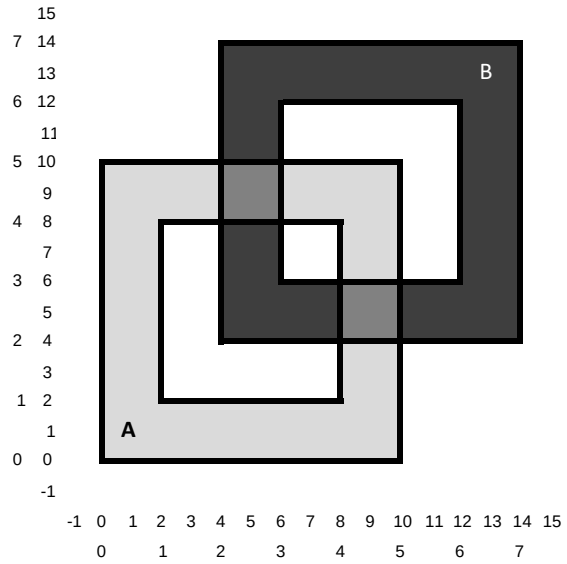
Other algorithms can address this problem. This one is presented to show the path tracing algorithms have unanticipated applications.

Pre-cleaning can be expensive. In some situations it can be guaranteed that all polygons are well behaved. For example the polygons can be the output of other algorithms that are known to be well behaved, such as contouring or character rendering. This implies preprocessing should be optional.

The same code is used for post-cleaning, which is needed because output polygons need to be properly nested and ordered.

8. An Example

The following uses polygons made up of horizontal and vertical lines to make the example easier to follow, but there no such restriction in the algorithms.



Here is the definition of the two polygons:

```
Coord rectA[] = {{0,0},{5,0},{5,5},{0,5}};
Coord holeA[] = {{1,1},{1,4},{4,4},{4,1}};
Coord rectB[] = {{2,2},{7,2},{7,7},{2,7}};
Coord holeB[] = {{3,3},{3,6},{6,6},{6,3}};
```

Hereafter, for each polygon listed, its name is followed by the nesting depth, number of points and the points themselves. Holes are stored as children of polygons; islands are stored as children of holes, and so on. Depth is which generation the outline is in the sequence.

After doubling the coordinate values per Section 6 and adding the intersections we have:

```
A 1  8 {0,0},{10,0},{10,4},{10,6},{10,10},{6,10},{4,10},{0,10}
A 2  8 {2,2},{2,8},{4,8},{6,8},{8,8},{8,6},{8,4},{8,2}
B 1  8 {4,10},{4,8},{4,4},{8,4},{10,4},{14,4},{14,14},{4,14}
B 2  8 {10,6},{8,6},{6,6},{6,8},{6,10},{6,12},{12,12},{12,6}
```

At each intersection a table is created of the segments that touch the intersection:

```
{10,4}: {10,0} {10,5} { 9,4} {14,4}
{ 4,10}: { 5,10} { 0,10} { 4,14} { 4,9}
{10, 6}: {10,5} {10,10} {12,6} { 9,6}
{ 6,10}: {10,10} { 5,10} { 6,9} { 6,12}
{ 4, 8}: { 2,8} { 5,8} { 4,9} { 4,4}
{ 8, 4}: { 8,5} { 8,2} { 4,4} { 9,4}
{ 6,8}: { 5,8} { 8,8} { 6,6} { 6,9}
{ 8,6}: { 8,8} { 8,5} { 9,6} { 6,6}
```

A table showing the CCW order of these segments is created. Because of the simplicity of our example (no bounces, no multiple intersections in one place), there are only two possible outcomes for sorting four rays as shown below:

```
{10, 4}: 0 3 1 2
{ 4,10}: 0 2 1 3
{10, 6}: 0 2 1 3
{ 6,10}: 0 3 1 2
{ 4, 8}: 0 3 1 2
{ 8, 4}: 0 2 1 3
{ 6, 8}: 0 2 1 3
{ 8, 6}: 0 3 1 2
```

Recall that to trace A-B we test all points on A outside of B and bear left at intersections. Each point is marked so the trace is not repeated. Here are the raw traces:

```
A-B  8 {10,10},{6,10},{6,9},{6,8},{8,8},{8,6},{9,6},{10,6}
A-B 12 {0,0},{10,0},{10,4},{9,4},{8,4},{8,2},{2,2},{2,8},{4,8},{4,9},
{4,10},{0,10}
```

The raw traces are processed to remove extraneous collinear points derived from the midpoints added above. This leaves only even numbers for the X and Y for every point. The coordinates' X and Y values are divided by 2, and the polygons are placed in a hierarchy where holes are children (but here we have no children so A-B has two siblings at depth 1):

```
A-B 1  6 {5,5},{3,5},{3,4},{4,4},{4,3},{5,3}
A-B 1 10 {0,0},{5,0},{5,2},{4,2},{4,1},{1,1},{1,4},{2,4},{2,5},{0,5}
```

A&B is an example of traces that start at added midpoints, so the raw traces look like this:

```
A&B  8 {5,10},{4,10},{4,9},{4,8},{5,8},{6,8},{6,9},{6,10}
A&B  8 {10,5},{10,6},{9,6},{8,6},{8,5},{8,4},{9,4},{10,4}
```

After removing extraneous points, dividing coordinates by 2 and forming the hierarchy, this is the result:

```
A&B 1  4 {5,3},{4,3},{4,2},{5,2}
A&B 1  4 {2,5},{2,4},{3,4},{3,5}
```

A|B has holes. The raw traces:

```
A|B  4 {8,8},{8,6},{6,6},{6,8}
A|B  6 {2,2},{2,8},{4,8},{4,4},{8,4},{8,2}
A|B  6 {10,10},{6,10},{6,12},{12,12},{12,6},{10,6}
A|B  8 {0,0},{10,0},{10,4},{14,4},{14,14},{4,14},{4,10},{0,10}
```

The usual post-processing yields this outline with three holes:

```
A|B 1  8 {0,0},{5,0},{5,2},{7,2},{7,7},{2,7},{2,5},{0,5}
A|B 2  6 {5,5},{3,5},{3,6},{6,6},{6,3},{5,3}
A|B 2  6 {1,1},{1,4},{2,4},{2,2},{4,2},{4,1}
A|B 2  4 {4,4},{4,3},{3,3},{3,4}
```

9. Implementation

The important result of this research is the two matrices with solutions to the crossing problem and the ray sorting problem. See Appendices B and C. However, to be sure the matrices are correct, an implementation was created written in what some people refer to as c+, meaning some c++ features are used without embracing classes and methods.

A simple application would acquire the polygon polylines and follow this example:

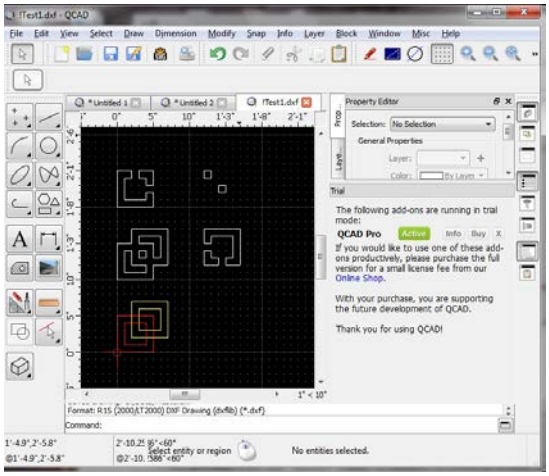
```
Coord **polyline[2];
int flags,nPoly[2],*nPolylinePoints[2];
PolygonStruct *psA,*psB,*psAmnsB,*psBmnsA,*psAandB,*psAplsB;
WingedEdgeStruct *wE;

// Acquire possibly several polyline Coord arrays per polygon, then...
// clean polygons (not needed in some applications)
nPoly[0] = GeometryClean(polyline[0],nPolylinePoints[0],nPoly[0],flags);
nPoly[1] = GeometryClean(polyline[1],nPolylinePoints[1],nPoly[1],flags);
// make nested polygon structures from polylines
psA = PolygonTreeFromCoords(polyline[0],nPolylinePoints[0],nPoly[0]);
psB = PolygonTreeFromCoords(polyline[1],nPolylinePoints[1],nPoly[1]);
// compute parameters for each intersection
wE = PolygonCompareGenWingedEdges(psA,"A",psB,"B",log);
// compute the combinations
psAplsB = PolygonTreeOpUnion(psA,psB,wE,log,"A+B");
psAandB = PolygonTreeOpInter(psA,psB,wE,log,"A&B");
psAmnsB = PolygonTreeOpMinus(psA,psB,wE,log,"A-B");
psBmnsA = PolygonTreeOpMinus(psB,psA,wE,log,"B-A");
// Now the resulting polygons are ready for other uses or output
```

Creating test cases was laborious. Fortunately, one of a variety of free programs, Qcad, was found to assist. By carefully constructing closed polylines in Qcad and using one color for the set of polylines for each polygon, what is called an SVG file could be produced in a consistent format. SVG is used to move CAD data between systems. A

straightforward program was written to read just that format (not general SVG files). It used the above code to compute the polygon combinations and then created SVG files in exactly the same format as the input.

The following shows Qcad with the above example input in red and yellow and the output polygons in white. They have been offset to separate them: A-B upper left of the four; A&B upper right; A|B lower left; B-A lower right.



10. Conclusion

Solution matrices support creating provably correct algorithms for comparing two line segments and sorting rays. The matrices can be filled by hand by studying each situation or in some cases by algorithms that use problems with known solutions to populate the matrix. Starting with using a polygon area formula for the triangle defined by a line and a point to position the point relative to the line, a family of algorithms was created for polygon operations. The two mentioned that use solution matrices are robust and efficient. They can be proved to be correct. Solution matrices can be used in other unrelated problems as well.

The source code is available at *to be determined*.

References

[1] Agathos, A., & Hormann, K.. (2001). The point in polygon problem for arbitrary polygons. *Comput. Geom., 20*, 131-144.

[2] Azevedo, L.G., & Güting, R.H.. (2007). Polygon Clipping and Polygon Reconstruction.*GEOINFO*.

[3] Bungatt, Bruce C., Winged Edge Polyhedron Representation, Computer Science Department Report No. CS-320, October, 1972, ftp://reports.stanford.edu/pub/cstr/reports/cs/tr/72/320/CS-TR-72-320.pdf.

[4] Greiner, Günther and Kai Hormann. "Efficient Clipping of Arbitrary Polygons." ACM Trans. Graph. 17 (1998): 71-83.

[5] Leonov, Michael, Comparison of the different algorithms for Polygon Boolean operations, http://www.complex-a5.ru/polyboolean/comp.html, 1998.

[6] Schirra, S.. (2008). How Reliable Are Practical Point-in-Polygon Strategies?. *ESA*.

[7] Weiler, Kevin, Polygon Comparison using a Graph Representation, SIGGRAPH 80 Proceedings, SIGGRAPH, Summer, 1980, pp. 10-18.

[8] https://en.wikipedia.org/wiki/Shoelace_formula

Acknowledgements

Parts of this research were supported by Integrated Software Systems Corporation (ISSCO) and Computer Associates. The rest was self-funded. The author wishes to thank Dr. Jim Blinn, Larry Diederichsen, Dr. Roland Sweet and Kenneth Martin for their help and encouragement.

The Author

Thomas Wright has a BS from the University of Colorado (1970, Applied Math). During the 1970s he worked at the National Center for Atmospheric Research, in time running the graphics group, which specialized in data representation graphics software libraries for scientists and engineers. During the 1980s he worked at ISSCO as Director of Development and Director of Research. He served on the board and was an officer of ACM SIGGRAPH. After ISSCO was acquired by Computer Associates he became Assistant Vice President. In the early 1990s he worked at Wavefront and then Megatek, but he switched his career to tax preparation in 1993. He retired in 2012 as an Enrolled Agent and owner of Tom Wright Taxes LLC..