The Polygon Package

by

Ivan Sutherland

Technical Report #1438

1977 and 1978

Computer Science Department

California Institute of Technology

Pasadena, California 91125

Silicon Structures Project

sponsored by

Burroughs Corporation, Digital Equipment Corporation,

Hewlett Packard Company, Honeywell Incorporated,

International Business Machines Corporation,

Intel Corporation, Xerox Corporation,

and the National Science Foundation

Memo to SSP FILE #1438

The memoranda listed below describe the polygon package in its present state. For easy reference, these memoranda have been packaged together.

Some of the material included is obsolete. In general, the later-dated material is better than the earlier material. Some specific hazards are indicated in the index below.

CONTENTS

CALIFORNIA INSTITUTE OF TECHNOLOGY

TO DISPLAY FILE #1129 (3 figures) DATE November 9, 1977

FROM Ivan E. Sutherland IES

SUBJECT Broadening Polygons: Straight and Circular Edge Conventions

This memorandum describes some polygon conventions which Jim Rowson and I have implemented in a Simula program. These conventions enable one to broaden polygons. They are intended for use as part of the design rule checker.

WHY CIRCULAR EDGES

We define polygons with either straight or circular edges. Circular edges are necessary because broadening a polygon creates a circular section for every convex vertex of the original polygon. Polygons defined with straight and circular edges can be broadened or narrowed and will still contain only straight and circular edges. Algorithms for dealing with straight and circular edges appear to be quite simple to write.

POLYGON CONVENTIONS

Each edge of a polygon is directed. Polygons are traced counter clockwise so that the inside of the polygon is to the left of the edges. Straight edges are represented by a line equation and a terminal vertex whose location is not explicitly represented in the data format. The line equation, $AX + BY + C = 0$, is normalized so that $A^2 + B^2 = 1$. The vector $[A,B]$ points towards the inside of the polygon.

Curved edges are represented by a reference to a terminal point and to a circle with a center and radius which may be either positive or negative. Circles with positive radius are counter clockwise circles whose direction is the same as the direction in which polygons are traced. The center of a positive radius circle is towards the inside of the polygon. Circles with positive radius are used to represent rounded corners. Circles with negative radius indicate that their center is outside the polygon; they turn in the clockwise sense. Circles with negative radius represent fillets in concave parts of the polygon. As we shall shortly see, representing circles both positive and negative radius provides important simplifications in the broadening routine.

DISPLAY FILE # 1129
Broadening Polygons
Ivan E. Sutherland

Both straight edges and circular edges are represented by reference to a terminal vertex and an equation, circular or linear. There is no indication of where the edge begins. This indefinite representation provides important simplification in the polygon algorithms because one need not compute the initial vertex of an edge when computing the edge. This reduces the amount of data which must be known at the time an edge is created.

THE BROADENING ROUTINE

The routine described here produces a correct, but possibly redundant output polygon which is broader (or narrower) than the original polygon. The routine makes one parameter, the broadening distance. Positive broadening distance indicates that the polygon should be made larger. Negative broadening distance indicates that the polygon should be made smaller.

For each straight edge in the original polygon, the broadening routine produces two output edges: first, a circular edge with radius equal to the broadening distance and centered at the starting vertex of the given edge and second, a straight edge parallel to the original edge and displaced by the broadening distance. The initial circular segment has either positive or negative radius according to the sign of the broadening distance. For a circular edge, the widening routine again produces two output edges: first a new circle centered at the initial vertex with radius equal to the broadening distance and second, a circular edge whose radius is larger or smaller than the given circular edge by the amount of the broadening distance. These processes are outlined in Figure 1A and 1B.

In most cases, the edges adjacent to a circular edge will be tangent to it. The vertex circles produced by the algorithm described here will therefore occupy no space. If a polygon in broadens and then narrows by an equal amount, a great many degenerate circles will be produced. We produce to clean up these degeneracies as a separate process.

DISPLAY FILE #1129
Broadening Polygons
Ivan E. Sutherland

It is interesting to note that this algorithm functions equally well for positive and negative broadening distances. For negative broadening distances (as shown in Figure 1C), it produces negative radius for each vertex of the given polygon, it displaces straight edges inward, and it decreases the radius of circular edges. As suggested in Figure 1, the algorithm is obviously the correct thing to do when broadening a convex vertex (Figure 1 A,B) or when narrowing a concave vertex (Figure 1C). Figure 2 shows a square which has been shrunk according to the algorithm described. The circular culicues at the corners of the resulting figure are traced clockwise thus indicating that they are negative areas i.e., holes drilled in free space. Such negative areas are redundant and meaningless and could be removed by a suitable algorithm. The algorithm produces a similar double area when expanding the concave vertex of an initial polygon as shown in Figure 3. The circular curlicue thus generated represents an overlay of the polygon in which itself, though redundant, is not wrong.

When one broadens a concave vertex or narrows a convex vertex, however, the algorithm produces peculiar results which are, nevertheless, correct.

STILL TO DO

We are seeking an algorithm to remove redundant overlap and holes in free space. We believe that such an algorithm will also merge multiple polygons which have the same name into a single polygon. Such algorithms are fairly easy to imagine if one is willing to devote N^2 effort to them, since they involve computing all intersections of a polyson's perimeter with itself. We seek something better than N^2.

IES:bc:1129

DISPLAY FILE #1129

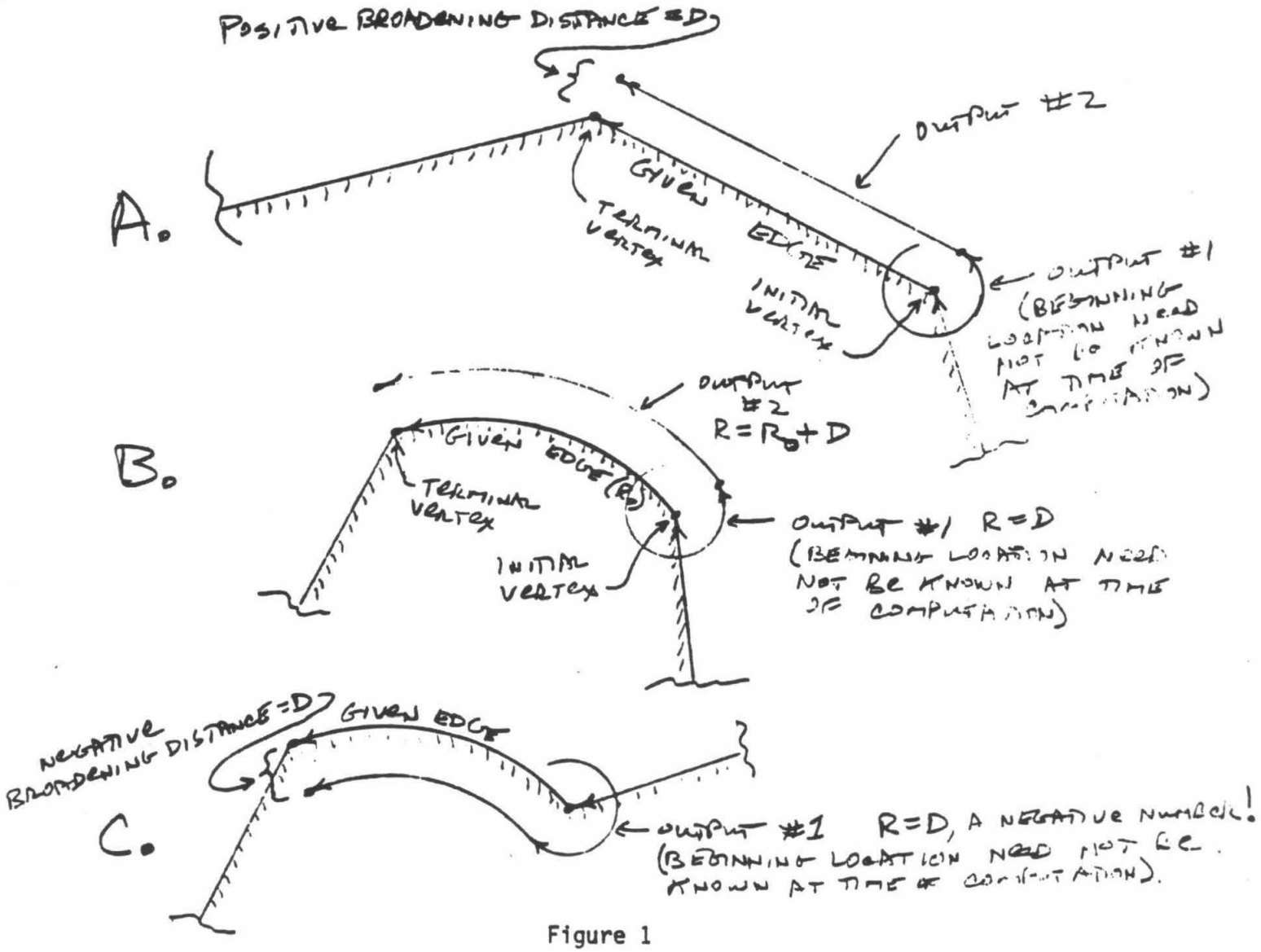


Figure 1

The broadening routine treats straight and circular edges alike. For each edge of the initial polygon, it creates two edges in the output polygon. First, a circle with radius equal to the broadening distance and centered at the initial vertex of the given edge. Second, a straight or curved edge displaced outward or inward from a given polygon edge by the amount of the broadening distance. The terminal vertex for the first circle is computed by moving outward or inward from the location of the initial vertex of the given edge. The terminal vertex of the main straight or curved output edge is computed by moving outward or inward from the location of the terminal vertex of the given edge.

DISPLAY FILE #1129

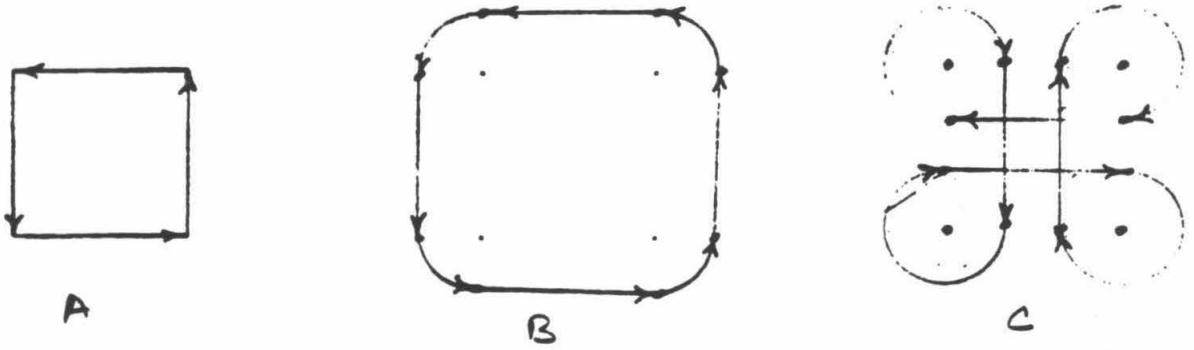


FIGURE 2. A SQUARE BROADENED AND NARROWED. THE SHADED AREAS ARE HOLES IN FREE SPACE WHICH SHOULD BE IGNORED.

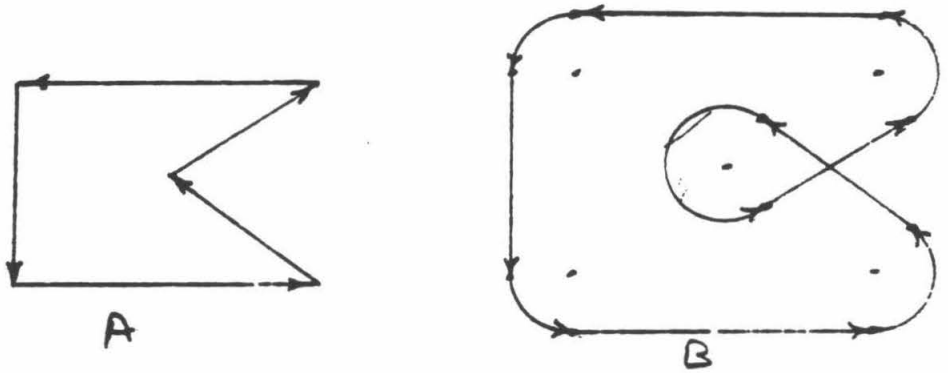


FIGURE 3. BROADENING A CONCAVE FIGURE. THE SHADED AREA IS A DOUBLE OVERLAP WHICH COULD BE REMOVED.

CALIFORNIA INSTITUTE OF TECHNOLOGY

TO DISPLAY FILE #1137 (2 figures) DATE November 9, 1977

FROM Ivan E. Sutherland IES

SUBJECT Geometry of Circles

In order to deal satisfactorily with circular segments we need to understand something about the geometry of circles. This memorandum discusses some computations that can be made with circular segments. These computations will be useful in clipping polygons which have circular segment edges and in computing the winding function for polygons with circular segment edges.

In what follows, I shall refer to a complete circle as a "circle", and a portion of a circle as a "circular segment". A circle has a center and a radius (which must be positive). A circular segment has a beginning point, B, a terminal point, T, a radius which may be either positive or negative, as well as a center. Positive radius circular segments are traced in the counter clockwise direction; negative radius circular segments are traced in the clockwise direction. By "line" I shall mean an infinite or semi-infinite line, and by "line segment" the portion of a line which lies between a beginning point, B, and a terminal point, T.

Let us consider some facts about circles as we have defined them. Consider first a view of B and T taken from the center of the circle C. The angle BCT measures the extent of the circular segment. Angle BCT will lie in the range:

$$-360^\circ < BCT < 360^\circ$$

with negative values corresponding to segments with negative radius and positive values corresponding to segments with positive radius. For any given B, C, and T there are two possible segments, one traced clockwise with negative radius, and one traced counterclockwise with positive radius.

Now consider the value of (B-C)x(T-C) which I define to be the Z component of the cross product of the two radial vectors, i.e.,

$$(B_x-C_x)(T_y-C_y) - (B_y-C_y)(T_x-C_x)$$

and abbreviate CR(BCT).

DISPLAY FILE #1137 (2 figures)
Geometry of Circles
Ivan E. Sutherland

The sign of this cross product says something about the direction of rotation that is experience at the center of the circle in going from B to T. Unfortunately because circular segments can have angles more than 180°, the cross product test is amgibuous. Nevertheless, consider making a cross product test from some point, P, in the circle near to its center. When P and C coincide, CR(BCT)=CR(BPT). As P moves away from C, CR(BCT) $\gtrless$ CR(BPT) as long as P does not cross the line defined by B and T.

Now let us consider in more detail what the cross product computation tells us. If the cross product is positive, it says that one must turn counter-clockwise to get from B to T through an angle whose magnitude is less than 180°. Thus, a counter-clockwise circle for which the cross produce computation on BCT is positive occupies less than 180°. In fact, if the sign of the cross product on BCT matches the sign of the radius, the segment occupies less than 180°. If the sign of the cross product of BCT and the sign of the radius are different, the circular segment occupies more than 180°. If the cross product is zero, the segment occupies either 0° or 180°.

The sign of the cross product from a point not at the center of the circle will match the sign of the cross product from the center only if that point is on the same side of the line through B and T as is the center.

Now let us consider lines which may or may not cross circular segments. If B and T are on opposite sides of a line L, the circular segment BT is bound to cross L exactly once. If both ends of the segment are on one side of the line, and the center of the circle is further from the line than that its radius, then the circular segment cannot cross the line. That leaves us with four cases as shown in Figure 2. In each case, the center is close enough so that we know that the circle crosses the line; the question is whether or nor the given circular segment crosses the line.

We first compute the point, P, on the line closest to the center of the circle. This is done by knowing the line equation, aX+bY+C=0 where the length of

vector ab is unity. Plugging the center of the circle into this equation tells us the distance, D, from the point to the line. By subtracting D times the vector a,b from the center of the circle we get the point P. If D is less than the magnitude of the radius, we are in the interesting cases.

Now if the segment is less than 180°, as shown in the upper two cases, CR(BCT) will match R in sign. If CR(BPT) also matches R in sign, we can be assured that the segment does not cross the line at all, as in case L_2. If CR(BPT) fails to match R, then the segment crosses the line twice.

If the segment is more than 180° in extent, as shown in the lower two cases of Figure 2, CR(BCT) will not match R in sign. If CR(BPT) matches CR(BCT) and thus does not match R, we can be assured that the circular segment crosses the line twice. If CR(BPT) does not match CR(BCT) and thus <u>does</u> match R in sign, the segment does not cross the line.

In both cases, we get the same result. If CR(BPT) matches R in sign, the circle does not cross the line.

IES:bc:1137

DISPLAY FILE #1137

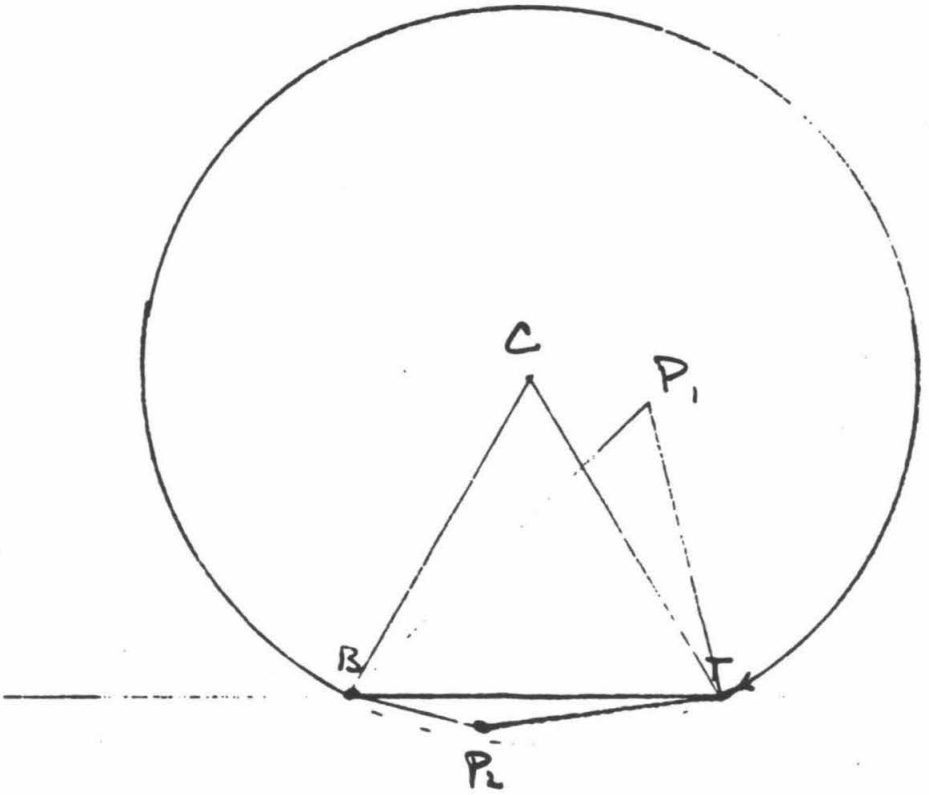


Figure 1 $CR(BCT) \overset{s}{=} CR(BP_1T) \overset{s}{\neq} CR(BP_2T)$

DISPLAY FILE #1137

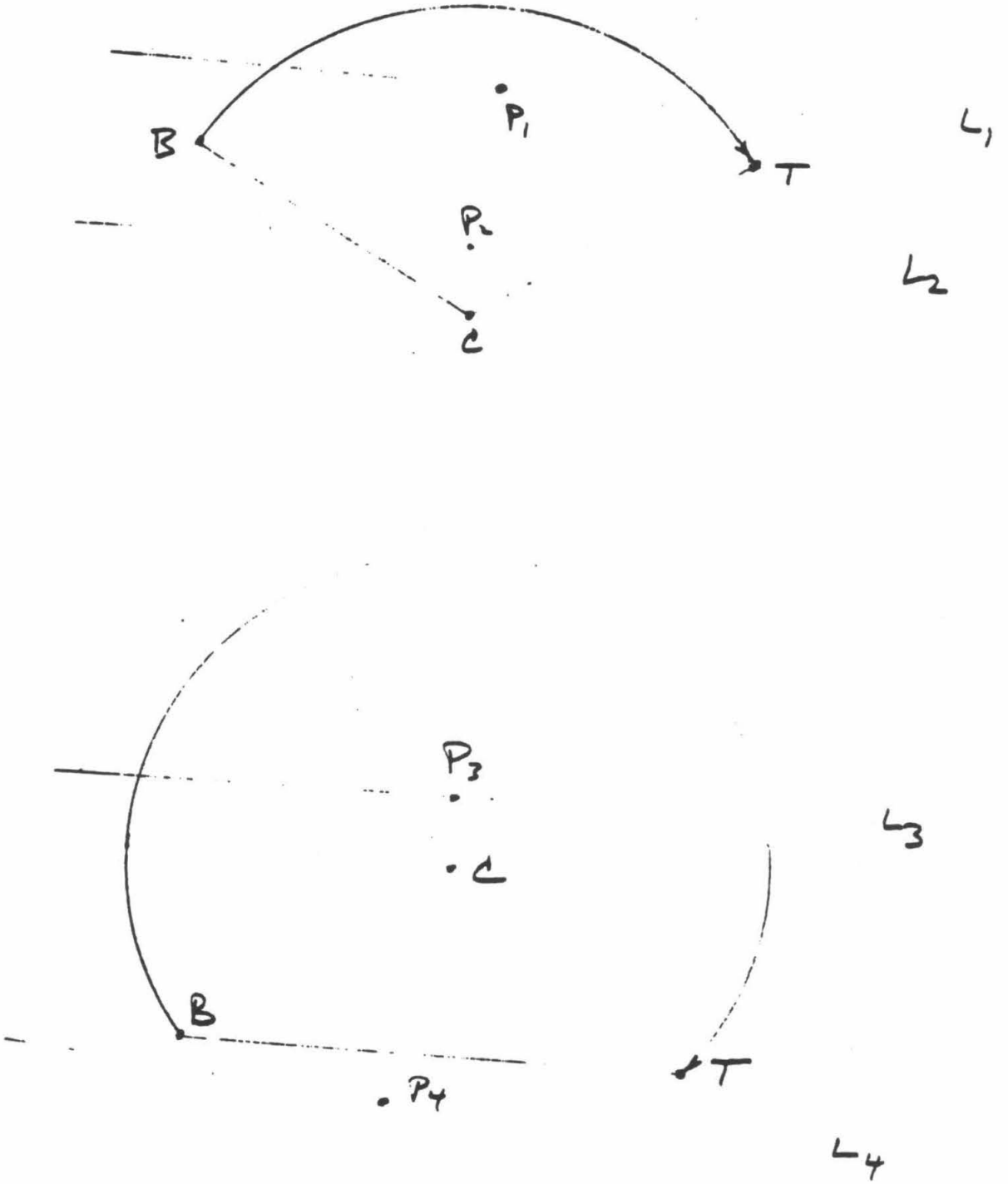


FIGURE 2 DOES THE CIRCULAR SEGMENT BT CROSS THE LINE L ?

CALIFORNIA INSTITUTE OF TECHNOLOGY

TO DISPLAY FILE #1146 (5 figures) DATE 11/9/77

FROM Ivan E. Sutherland IES

SUBJECT Computing the wrap number to determine if a point is inside a polygon one can compute the number of times the polygon "wraps" the point.

This memorandum describes a set of conventions for polygons with straight and circular edges which make the wrap number computation consistent. The technique involved computing the number of times that the perimeter crosses the horizontal test line from the test point to infinity.

I have been unable to find a consistent set of computations when the test point itself is included in the test line. Therefore, this memo describes a set of tests which consider the open semi-infinite interval to the right of the test point.

Figure 1 shows all possible relationships between a segment and the straight line test point. Consistent with the convention of tracing polygons in the counter clockwise direction (see DF memo), the edge arrows have been shaded on the left side to indicate that the "meat" of the polygon is on this side. Line crossings are considered positive when in the counter clockwise sense and negative in the clockwise sense. The test line itself is considered to be in the first quadrant, thus crossing takes place when the line actually crosses an umaginary place a trifle below the given test line. This distinction is made to discriminate between "less than" tests and "less than or equal to" tests.

CIRCLES

Circular segments cross the test line under two conditions: first, if they began on one side of the test line and terminate on the other side and are suitably far to the right, they obviously cross the test line, as shown in Figure 2. Suitably far to the right is defined to mean that the center of the circle lies within the shaded area shown in Figure 3, where the circular portion of area around the test point is included or not according to the sense of the sense of the circle and the up/down sense of the crossing.

If both ends of the circle are above or below the test line, the circular setment will made a net winding of zero unless its center happens to be close to the test point. Some cases with net zero crossings are shown in Figure 4.

If both ends of the circle are above or below the test line and the center of the circle is closer to the test point than the radius of the circle, a net winding may occur. Here again, we refer to the Display File Memo on circle geometry #1137 to compute what is going on. For the point P we shall use the test point itself, since it is known to be on the line and inside the circle. Thus we can compute CR(BPT) and compare its sign to that of R. If there is a match, a winding number of zero results. If there is no match, then the winding number is plus or minus one depending only on the sign of R.

DISPLAY FILE #1146

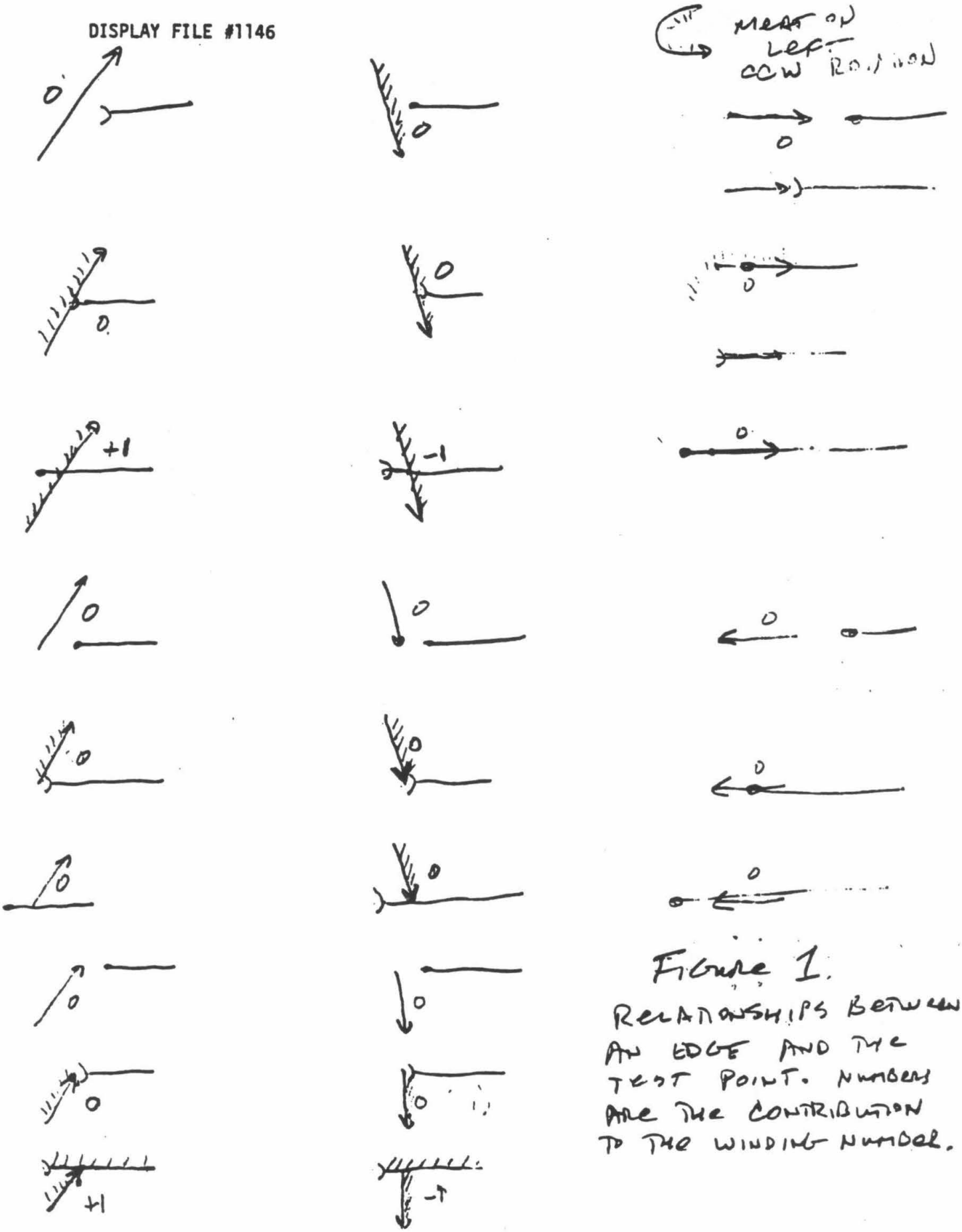


FIGURE 1.
RELATIONSHIPS BETWEEN AN EDGE AND THE TEST POINT. NUMBERS ARE THE CONTRIBUTION TO THE WINDING NUMBER.

DISPLAY FILE #1146

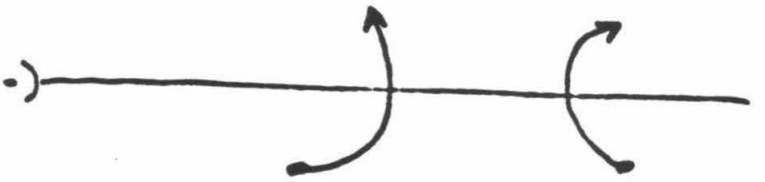


FIGURE 2 CIRCULAR SEGMENTS WITH ONE END BELOW AND ONE END ABOVE THE TEST POINT CROSS THE TEST LINE IF SUITABLY FAR TO THE RIGHT.

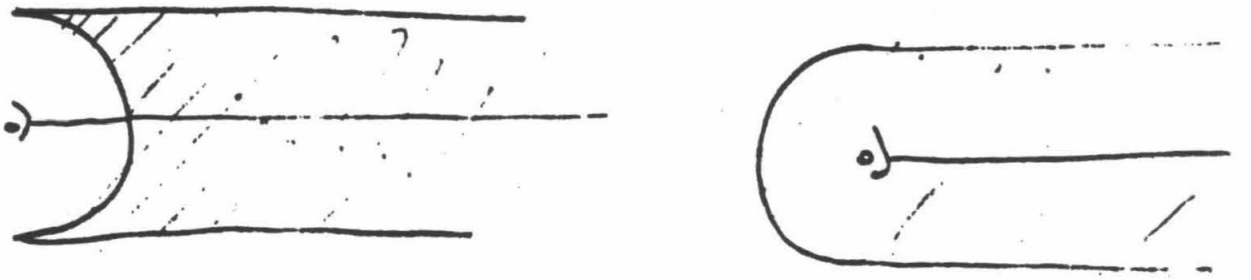


FIGURE 3. SUITABLY FAR TO THE RIGHT IS DEFINED AS HAVING CENTER WITHIN THE SHADED REGIONS DEPENDING ON DIRECTION OF ROTATION AND UP/DOWN SENSE OF CROSSING.

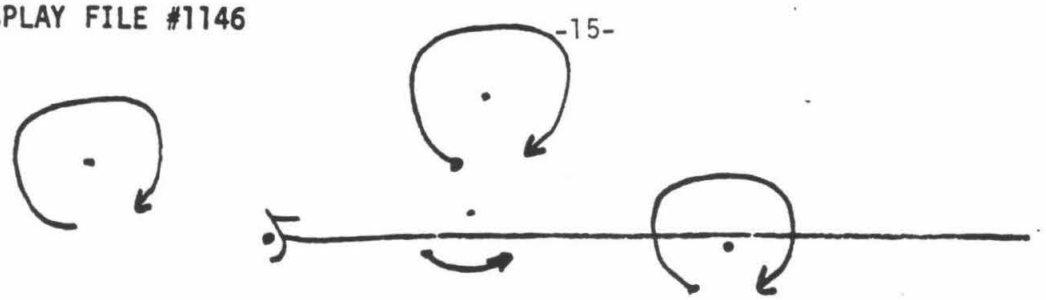


FIGURE 4 SOME SEGMENTS WITH ZERO NET CROSSINGS.

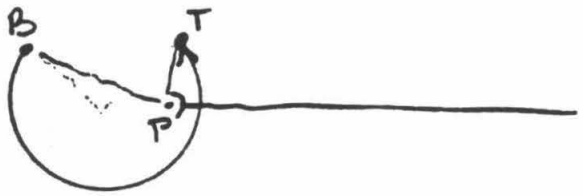


FIGURE 5 IF CR(BPT) MATCHES R IN SIGN THEN THE CIRCULAR SEGMENT DOES NOT CROSS THE TEST LINE. SEE ALSO DISPLAY FILE 1137, "GEOMETRY OF CIRCLES."

CALIFORNIA INSTITUTE OF TECHNOLOGY

TO DISPLAY FILE #1164 DATE November 23, 1977

FROM Ivan E. Sutherland IES

SUBJECT Self Intersections of Polygons

THE PROBLEM

A well-formed polygon may not have self-intersecting edges. In order to guarantee that a polygon is well formed, one must check all edges to discover any self intersections. The problem is to accomplish this check in less than N^2 time.

THE IDEA

The key idea is to sort the vertices of the polygon by angular position around its centroid. The order in which the vertices are connected by the edges is a good initial approximation to the required sorting order. Moreover, modifications to the polygon disturb the order of angular position around the centroid only slightly. Thus a simple interchange sorting procedure can be used both to establish the initial order of vertices and to maintain the ordered list through polygon modification.

Edges which intersect must share a common angular region as viewed from the centroid. An edge which extends from point A to point B may be crossed only by edges whose angular span includes some portion of the interval from A to B. Thus the number of edges against which a given edge must be tested is vastly reduced.

IMPLEMENTATION

One implementation of this notion utilizes a pair of pointers in each vertex. I shall call these pointers PREV and NEXT. If the vertex is not a part of a sorted list, PREV and NEXT will aim at their own vertex. If the vertex is a part of a sorted list, PREV will aim at the vertex clockwise from this one and NEXT at the vertex counter clockwise from this one.

A routine to extract a vertex from a sorted list is easy to write. It simply heals up the vacant space by changing the PREV and NEXT pointers of the adjacent vertex.

DISPLAY FILE #1164
Ivan E. Sutherland
Self Intersections of Polygons

A routine to insert a vertex in its correct position in a sorted list is also easy to write. Such a routine needs to know two things:

1. The name of the vertex being inserted
2. Starting vertex in the list near which insertion is to be accomplished.

Such a routine will search the given list forward or backward from the starting vertex as appropriate. Thus the amount of time taken for insertion will depend on the number of vertices between the vertex to be inserted and the starting vertex. When sorting a polygon ab initio, one should make new insertions adjacent to the most recently inserted vertex, thus making use of the fact that the sequence of vertices defined by the edges is approximately the correct sorting order. When checking that a sort is correct, one should start searching the list for the correct position of vertex V at the current location of vertex V, thus making use of the fact that the list is already approximately in sort.

IES:bc:1164

CALIFORNIA INSTITUTE OF TECHNOLOGY

TO Joint IC File (2 figures) #1247 DATE January 3, 1978

FROM Ivan E. Sutherland IES.

SUBJECT Intersections

This memorandum describes some algorithms for computing the intersections between lines and circles. These algorithms have been implemented in SIMULA as part of the classes "line" and "circle" in class "graphics". This memorandum is intended to document these routines.

INTERSECTION OF TWO LINES

Given two line equations:

$$a_1x + b_1y + c_1 = 0 \text{ and } a_2x + b_2y + c_2 = 0$$

we may write them in matrix form as:

$$\begin{bmatrix} a_1 & b_1 \\ a_2 & b_2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}$$

and observe that their intersection is found by left multiplying both sides of this expression by the inverse of the ab matrix.
The intersection is given by

$$\begin{bmatrix} x \\ y \end{bmatrix} = \frac{1}{D} \begin{bmatrix} b_2 & -b_1 \\ -a_2 & a_1 \end{bmatrix} \begin{bmatrix} -c_1 \\ -c_2 \end{bmatrix} \quad \text{where } D = \begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}$$

Thus the computation to be done is:

```
d:=a1*b2-a2*b1;
IF d = 0 THEN lines are parallel ELSE
x:=(b1*c2-b2*c1)/d; y:=(a2*c1-a1*c2)/d;
```

JOINT IC FILE #1247
Ivan E. Sutherland
January 3, 1978

CIRCLE AND LINE

There may be none, one or two intersections between a circle and a line. Referring to the diagram of Figure 1, we see the circle C and the line L which intersect at two points S and T. The point on the line closest to the center of the circle is called Q, and is a distance D from the center. The two intersections are each a distance E away from the point Q. The radius of the circle is R, and a unit vector called [a,b] is perpendicular to the line.

Distance D may be obtained by plugging the coordinates of P into the normalized line equation.

D:=a*Px+b*Py+c; where [a,b] is a unit vector

D is positive or negative according to which side of the line the circle center lies on. In either case, the position of Q can be found as: Q= P + D[a,b] . E, the offset from Q, can be found by Pythagorean Theorem as

$$E = (R^2 - D^2)^{1/2}$$

The intersections S and T can be found from Q by noticing that one needs a vector perpendicular to [a,b] , namely either [-b,a] or [b,-a] as a unit vector in the direction of the line. Thus the total computation is:

```
D:=a*px +b*py + c;  ee: = r*r - d*d;
IF ee<0 THEN no intersections ELSE
e:=SQRT(ee);
sx:= px + a*d + b*e;
sy:= py + b*d - a*e;
tx:= px + a*d - b*e;
ty:= py + b*d + a*e;
```

Notice that the test for the number of intersections to compute is made on the basis of ee, the number whose square root is to be taken. This direct test ensures that even in the face of numerical inaccuracy, the routine will never attempt the square root of a negative number. One might instead have compared the magnitudes of D and R, a less desirable procedure in my opinion.

JOINT IC FILE #1247
Ivan E. Sutherland
January 3, 1978

INTERSECTION OF TWO CIRCLES

The two intersections of two circles are found in a similar way. Figure 2 shows two possible configurations of two circles. The centers of the circles are called P1 and P2. They are a distance D apart. Again a point Q lies midway between the two intersections. Q is a distance F away from the mid-point M, of the two circle centers. The intersections, S and T, are spaced a distance E away from Q.

Now this time we do not have any normalized vectors to use in computing the positions of Q and S and T. We must, therefore, make use of the vector P2-P1 whose length is D. We are, therefore, interested not so much in the values of E and F, but rather in the ratios E/D and F/D. The locations of the points can be found as:

$$S = (P1 + P2)/2 + (F/D)(P2 - P1) + (E/D)\ (perp(P2 - P1))$$

and similarly for T. Perp indicates a vector perpendicular to the given vector, which is easily found by interchanging coordinates and changing the sign of one of them.

Display File #1247
Ivan E. Sutherland
January 3, 1978

Now the distance from P1 to Q can be seen to be (F - D/2) for the case of figure 2a and (D/2 - F) for the case of figure 2b. The distance from P2 to Q is (F + D/2) in both cases. From these observations we can set up the basic equations:

$$(D/2 - F)^2 + E^2 = R1^2$$

$$(D/2 + F)^2 + E^2 = R2^2$$

It will turn out that we do not need to know which of the radii is larger because interchanging their roles merely changes the sign of F. Subtracting to eliminate E gives us:

$$(D/2 + F)^2 - (D/2 - F)^2 = R2^2 - R1^2$$

By expanding the squares and eliminating the terms which cancel we get:

$$2DF = R2^2 - R1^2 \quad \text{or} \quad F/D = \frac{R2^2 - R1^2}{2D^2}$$

This is rather convenient since it is easier to compute D^2 than to compute D.

Now is we add the two basic equations to solve for E we get:

$$(D/2 + F)^2 + (D/2 - F)^2 + 2E^2 = R2^2 + R1^2$$

Again expanding the equares and eliminating the terms in DF we get

$$(D/2)^2 + F^2 + E^2 = \frac{R2^2 + R1^2}{2}$$

But we are interest in E/D rather than in E and so we find

$$E/D = \left[\frac{R1^2 + R2^2}{2D^2} - (F/D)^2 - (1/4)\right]^{\frac{1}{2}}$$

And that, gentlemen, is all there is to it. Only one square root to take. Again, if the number inside the square root is negative it means that either the circles are too far appart to intersect, or that the difference in their radii is so large that one lies totally inside the other.

Display File #1247
Ivan E. Sutherland
January 3, 1978

Thus using the sign of the square root operation as a clue to how many intersections there are is straightforward. The algorithm is:

```
dx := x2 - x1; dy: = y2 - y1;

dd := dx**2 + dy**2;

fod := 0.5*(r2**2 - r1**2)/dd

eods := 0.5*(r2**2 + r1**2)/dd - fod**2 - 0.25;

IF eods<0 THEN no intersections ELSE

eod := SQRT(eods);

sx := (x1 + x2)/2 + fod*dx - eod*dy;

sy := (y1 + y2)/2 + fod*dy + eod*dx;

tx := (x1 + x2)/2 + fod*dx + eod*dy;

ty := (y1 + y2)/2 + fod*dy - eod*dx;
```

That's all there is to it. I wish it had been as simple to find as it is to describe.

IES:y1:1247

JOINT IC FILE #1247

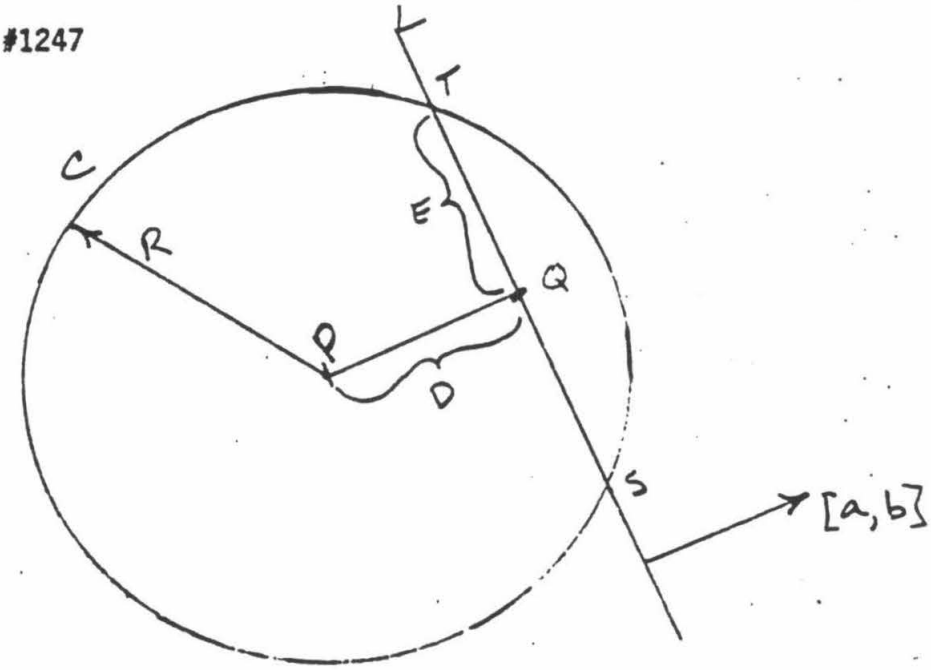


FIGURE 1 CIRCLE/LINE INTERSECTIONS.

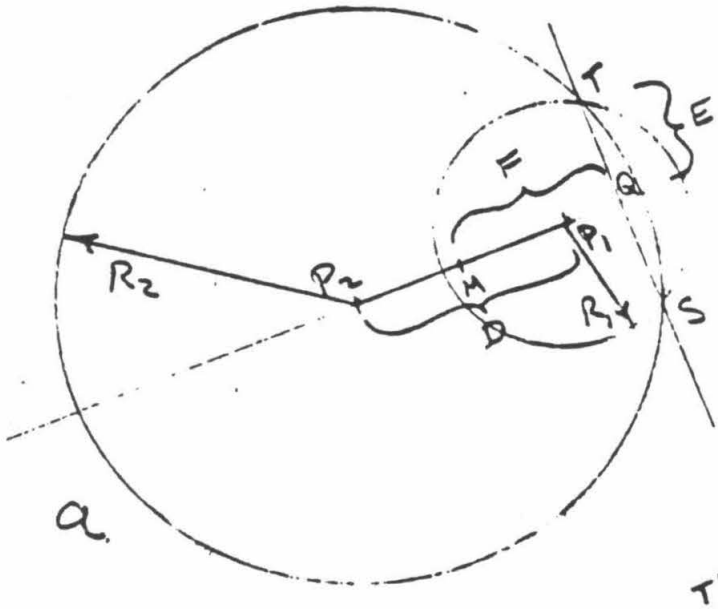
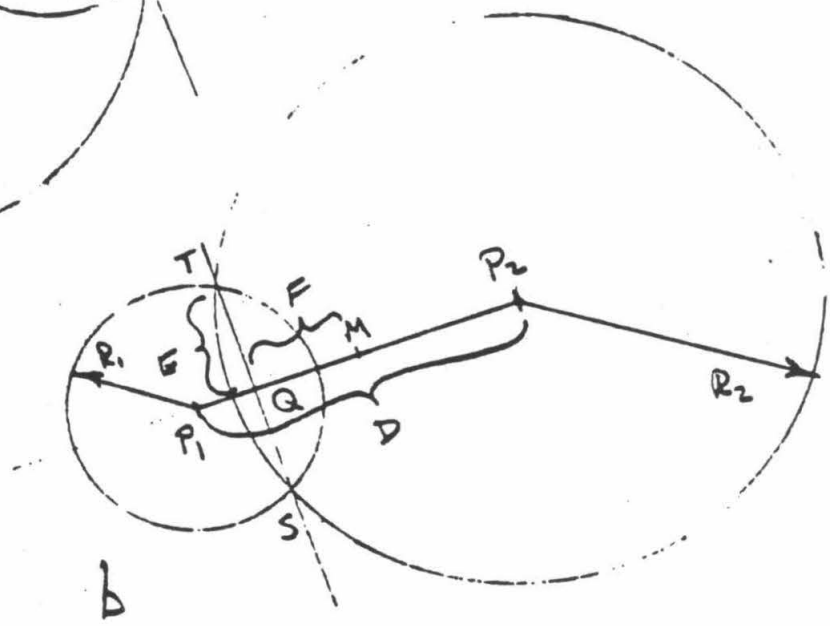


FIGURE 2
CIRCLE/CIRCLE
INTERSECTIONS



CALIFORNIA INSTITUTE OF TECHNOLOGY

TO Joint IC File #1263 DATE January 1, 1978

FROM Ivan E. Sutherland IES

SUBJECT Angular Comparisons

It is sometimes necessary to compare the angular relationship of points or directions. I have used two approaches to such angular comparisons:

a) A function monotonic in the angle for direct comparison
b) A cross product method to determine relative angular position. I have come to favor the cross product method although the angle method is still in extensive use. This memo documents both methods and points out the weakness of the angle method.

APPROXIMATE ANGLES

An approximate angle routine called AANGLE(x,y) may be used to compute an approximate angle for a vector x,y. The range of the function is $-1/2 \leq$ AANGLE $< 1/2$. AANGLE is zero when x and y represent a vector to the right i.e., when y is zero and x is positive. AANGLE is increasingly positive as the vector xy rotates counterclockwise about the origin. AANGLE assumes negative values when the vector xy rotates counterclockwise around the origin. Thus negative values of y produce negative AANGLE and positive values of y produce positive results.

The approximate angle function is monotonic as is the actual angle but is simpler to compute than the arc tangent function. I have used, in fact, the ratio of the smaller x and y magnitudes to the larger and then added or subtracted appropriate numbers to account for the quadrant. The function is, therefore, exact at 90 and 45 degrees directions - but only approximately equal to the actual angle in between. Because the function is monotonic, it can be used satisfactorily for comparisons.

An angle difference routine called ADIF(a,b) computes angular differences mod 1 in the range $-\frac{1}{2} \leq$ ADIF $< \frac{1}{2}$. ADIF computes a-b and then adds or substracte one or two as appropriate to reduce the answer to the correct range.

Angular Comparisons #1263
Ivan E. Sutherland
January 1, 1978

AANGLE and ADIF are such that the difference of the approximate angles of two vectors will be least, that is to say $-\frac{1}{2}$, when the two vectors point in opposite directions. The difference will be zero when the two vectors point in the same direction and will be greatest when the vector whose AANGLE is a point almost back but to the left of the other, i.e., when motion from B to A represents a shart left turn.

Each edge has two properties called FANGLE(p) and LANGLE(p). FANGLE and LANGLE take a point as an arguement. FANGLE computes the first AANGLE of the edge as viewed from the point in the counterclockwise sense. That is to say, looking from the point away from the edge and turning left, one eventually will see a part of the edge. FANGLE computes the approximate angle which one will be facing when looking at this extreme right hand end the edge from the point. LANGLE computes the last approximate angle, i.e., the angle to the extreme left hand end of the edge as viewed from the point.

It is not necessarily true that FANGLE $<$ LANGLE, because an edge generally to the west of a point may have a positive FANGLE and a negative LANGLE. Nor is it necessarily the case that FANGLE and LANGLE are less than 180 degrees apart, since circular edges may partly enclose the point.

It has turned out with experience that approximate angles are not very useful. One is usually interested in comparing them, and such comparisons inevitably must face up to the discontinuity between $-\frac{1}{2}$ and $+\frac{1}{2}$. This discontinuity is contrary to the whole notion of angular continuity around the origin. Moreover if one asks about the angular order of ab, one can only make a meaningful statement if one talks about the shorter angle between a and b. In none of the computations which I have done, is the shorter angle of any significance. One wants, rather, to make sequence comparisons to discover whether three vectors a,b, and c form a clockwise or counterclockwise set.

Angular Comparisons #1263
Ivan E. Sutherland
January 1, 1978

Or put another way, one wishes to know whether B is "between" A and C as one progresses counterclockwise from A to C. Because the cross product method approaches these questions directly, I believe it to be preferable.

THE CROSS PRODUCT METHOD

Given two vectors, x_1y_1 and x_2y_2, one can determine their angular relationship by computing their cross product. Because all vectors lie in a plane, only the z components of the cross product need be computed. It is formed as $x_1y_2 - x_2y_1$. The cross product will be positive if the shortest direction of rotation between vector 1 and vector 2 is counterclockwise and negative otherwise.

Given 3 vectors A,B,C, which appear in counterclockwise sequence, there will be three angles AB, BC, and CA defined. AB + BC + CA = 360°. Thus at most one of these three angles can exceed 180°; the other two will be less than 180°. Therefore at least two of the cross products A x B, B x C, and C x A will be positive. Similarly, if two or more of the cross products are negative, then the sequence is reversed.

One or more of the cross products might be zero. If one only is zero it may indicate that those two vectors are equal or opposite. If the two vectors are opposite, the other two cross products will have the same sign, a sign which correctly indicates the sequence. If the two vectors are identical the other two cross products will differ in sign, indicating that sequence is not defined.

Given 3 vectors A B C which appear in counterclockwise sequence, there will be three angles, AB, BC, and CA defined. AB + BC + CA = 360°. Thus at most one of these three angles can exceed 180°; the other two will be less than 180°. Therefore at least two of the cross products A x B, B x C, and C x A will be positive. Similarly, if two or more of the crossproducts are negative, then the sequence is reversed.

Angular Comparisons #1263
Ivan E. Sutherland
January 1, 1978

One or more of the cross products might be zero. If one only is zero it may indicate that those two vectors are equal or opposite. If the two vectors are opposite, the other two cross products will have the same sign, a sign which correctly indicates the sequence. If the two vectors are identical, the other cross products will differ in sign, indicating that sequence is not defined. If two cross products are zero, no sequence can be defined, because at least two of the vectors point in the same direction.

Thus a routine to check angular sequence can be very simple:

```
        ab:=ax*by - ay*bx;
        bc:=bx*cy - by*cx;
        ca:=cx*ay - cy*ax;
For **:= ab, bc, ca do begin
        If xx<o   THEN  seq:=seq - 1;
        If xx>o   THEN  seq:=seq + 1;
End of LOOP
```

Such a routine exists in graphics but has not yet been used for anything.

IES:yl:1263

CALIFORNIA INSTITUTE OF TECHNOLOGY

TO Silicon Engineering Project File #1316 DATE December 27, 1977

FROM Ivan E. Sutherland 9 45

SUBJECT A Design Rule Checker

This memorandum sets forth some ideas on a design rule checker. The proposed system will make use of the hierarchical nature of IC designs to substantially reduce the computation required for checking design rules. The system proposed will compare the given geometry with a given logical connection list so as to detect missing connections and erroneous connections as well as spacing violations.

INPUTS

There will be three forms of inputs required. The design rule checker will confirm that the three separate inputs are mutually consistent. The three inputs are: a) a proposed circuit geometry, b) a set of global geometric rules to which it should conform, and c) a description of the logical connections which the geometry is intended to implement.

The geometry of the circuit will be described in a hierarchical fashion. Each level of the hierarchy will consist of polygons, wires, and instances of sub-structures which in turn consist of polygons, wires and instances of sub-structures. Such a hierarchical description can not only be substantially shorter than a "fully initiated" description in which substructures are not explicitly represented, but also matches well the functional descriptions of circuits in which "registers", "busses", "adders" and other structures are composed of repeated elements. I propose to make extensive use of the hierarchical nature of the geometric description to reduce both computation time and memory requirements.

Each element in the circuit geometry will carry a logical name. Thus a wire might be labled "ground" or "bit23". Names will be local to instances so that two instances of a half adder cell, for example, might each contain a wire labled "sum". Names of logical elements within instances can be referenced at a higher level in the hierarchy by using a dot notation similar to that of SIMULA or PASCAL. Thus the two wires

labled "sum might be designated at the higher level of the adder as "bit3.sum" and "bit4.sum" if the names of the two half adders were "bit3" and "bit4" respectively.

Of course the names attached to the geometric description will be compared to those attached to the description of logical interconnections. This cross referencing will more than repay the effort required to attach specific names to geometric elements. Moreover, design rule violations can be identified both by geometric position and by the names of the elements in violation. For example, one might get a report such as: "ground and "power" too close at 4678,3422.

The global design rules to be checked will be described in terms of logical operations on polygons. Because the system proposed will be able to handle polygons with both straight and circular edges, the design rules can be specified in terms of fattening, thinning, union, intersection, and negation of polygons. For example, a minimum spacing rule of 10μ can be specified as "intersection(fatten(5))=none". I am not yet sure exactly how to specify all design rules in terms of fattening and thinning, union, intersection, and negation, but there is certainly a very interesting set of operations which these functions together make possible.

One hopes that the specification of the design rules is sufficiently simple that it can be gotten right without automatic checks. This is a weakness in the system. No check on the value, accuracy, or correctness of the global design rule specification is proposed. We accept as given that metal cannot be closer than _____ microns from other metal. Errors in the design rule input will show up as poor circuit yield without any advance warning.

The logical connection list input will be a series of statements of intent to connect. For example, it may include the notion that adderbit23.ground" and "ground" should be connected together. Similarly it specify that "adderbit5.input" and "adderbit55.sum" should be connected together. The logical connection list need only consider connections between sub-structures at one level in the hierarchical description; connection witin a sub-structure will be considered in a separate logical connection list.

Design Rule Checker #1316
Ivan E. Sutherland
December 27, 1977

The separation of the logical connection list into levels in the hierarchy not only reduces the storage requirement and computation time for dealing with the structure, but also conforms nicely with a hierarchical simulation system to confirm that the logical connection list in fact will give the desired performance. I see the logical connection list as a link between the design rule checking system and the logical and electrical simulation systems.

HOW IT WORKS

I have been working for some time on a geometric computation package for polygons with straight and curved edges. As this geometric package is now beginning to work it is possible to see how to use it to achieve the functions proposed in this document. I shall digress, therefore, into a little description of the geometric computation package.

Polygons are described with straight and curved edges. A function called fatten(d) is defined for each polygon which constructs another polygon which is "d" larger than its predecessor. A fattened square develops rounded corners. If the parameter "d" is negative, fatten(d) makes the polygon smaller.

Now the hard part of fattening and thinning polygons is that these operations may totally change the topology of the polygon. A polygon with a pronounced C shape and a narrow gap may turn into an O shape when fattened a little, or even turn into a box without a center operning when fattened a lot. Similarly, a dog-bone shape may turn into two disjointed pieces when thinned. I believe that my geometric routines will compute all these changes correctly. Although we do not yet know how fast the algorithms will be, we at least know that they grow as something on the order of N log N rather than as N^2 since they use sorting in careful ways.

The same routines which handle all of the "self intersections" of fattened polygons can handle the edge intersections of different polygons so as to compute a single polygon which is the union or intersection of two such polygons. Because polygons are normally traced counterclockwise, negation of polygon amounts to reversing the direction in which it is traced. We are thus already armed with a good battery of

computation routines for what is to follow; in fact, it is the existence of these routines which lead to the overall system description set forth here.

Now consider the metal layer of some sub-circuit, say the half adder which I have been using as my example. Suppose that metal runs must be separated by a distance "s". Within the half adder, we can check this rule by expanding all of the metal areas by s/2 and then checking that there is never any overlap between polygons which carry different names. Suppose that the half adder passes this test. We must now make sure that none of the wiring around the half adder "violates its space". Instead of checking all of the wiring around the half adder against the individual wires within it, we can test them against a blob which represents "its space". Such a blob can be computed by the fattening of all the half adder's metal areas by "s," taking the union of the resulting polygons, and then thinning again by "s." By considering each sub-symbol as a "blob" at the next higher level in the hierarchy, we should be able to save computation time and memory space.

If each sub-circuit is represented as an outline blob, how can interconnections between them be computed? Suppose that a wire violates the space of a sub-circuit. Then the intersection area of the wire and the blob of the sub-circuit can easily be computed. At the level of the sub-circuit, this intersection can be compared with the different wires of the sub-circuit to see which one it connects to. Thus one can discover that there is area in common between a wire at the higher level and the wire inside the sub-circuit. Similarly, area which is shared by two sub-sircuits can be compared with each to discover direct connection between sub-circuits.

If one uses such a technique of discovering at each level of the hierarchy, the areas shared by more than one circuit element, and then accounting for them by examination of lower level circuits, one can detect the circuit connectivity. Obviously, this connectivity is compared to the given wiring list and any discrepancies reported. Such reports can include both connections discovered in the geometry which were not predicted in the

circuit description and connections in the circuit discription which are missing in the geometry.

Now it is often the case that the sub-circuits which I design have spaces left in them for wiring at a higher level in the hierarchy. Such spaces pass through the sub-circuits. The blob for a sub-circuit with such a space in it will be two disjointed pieces. It will be discovered that the wire passing through the space between such pieces does not violate the space of the sub-circuit.

It may also be convenient to describe explicitly the connection points of sub-circuits. Such an explicit description might describe a sub-circuit as certain regions which are not to be violated ever, and certain ones to which connections may be made. In effect, the sub-element might preserve some of the character of its wiring geometry, at least for those words which will be of importance when the sub-circuit is used. Such a description would simplify the lookup of connections between circuits. I would prefer to avoid describing explicit connection points on sub-circuits; I do not believe we need to force people to say <u>where</u> they will connect to the ground wire of a circuit, for example, but merely to specify that they will do so somehow.

USE OF THE HIERARCHY

Discovering which parts of one polygon overlap which parts of another requires one to compare all parts of one with all parts of the other. The comparisons are done in my programs by using a sorted list of elements and an active list of elements. Elements are moved from the sorted list to the active list. As each element is moved into the active list it is compared to all elements already in the active list to check for overlaps. The total number of comparisons required is the number of elements in the sorted list times the average length of the active list. If one simply accumulates elements in the avtive list, the total number of comparisons will be $n^2/2$. If one culls the active list, removing from it all element which have been

"passed " in the sorted list, one can reduce the length of the active list considerably. The nature of the sort used for the sorted list and the nature of the cull must, of course, be related. I have implemented a sort by angular position from a centroid, but sorting by radial distance from the centroid, by north-south location or by other criteria might equally well be used.

Now the sorting process can be accomplished simply by using the hierarchy. A sub-element may be placed, in toto, in the sorted list and expanded only when actual comparison of its elements with those of the active list is required. It is harder to see how to cull all of a sub-structure from the active list, since elements in the active list will have been expanded, but perhaps it is possible also. Thus the hierarchical nature of the structure should be useful internally in the geometric computation as well as being useful to reduce the number of elements with which the programs must deal.

CONCLUSION

I am proposing that we proceed on a design rule checker such as I have described here. We have some basic geometric routines. We need to build the hierarchical structure on top of them. I believe we have a good chance of doing something really interesting and useful.

IES:slr:1316

CALIFORNIA INSTITUTE OF TECHNOLOGY

TO Silicon Engineering Project File #1322 DATE January 19, 1978

FROM Ivan Sutherland
Bob Sproull

SUBJECT An Overlap Detector

The design rule checker needs a polygonal overlap detector. Obviously such a program is similar to the hidden surface routines with which we are familiar. The integrated circuit polygonal material, however, has three special characteristics which we should be able to use to simplify and speed up the design rule checking program:

1. The depth number is low, i.e., the number of polygons which overlap is very small, usually a maximum of two.
2. Many of the lines are horizontal or vertical, and polygons tend to be long and thin.
3. There is a symbol structure which might be used to partition the task.

Given these facts about the polygons to be tested, one might well proceed as follows:

a) Sub-divide the picture using a Warnock-type algorithm with horizontal and/or vertical cuts. During partitioning, one would treat symbols and polygons and wires merely as minimum bounding boxes, simply listing those which appear in each sub-window. The Warnock partitioning step serves to focus attention locally on the tasks to be performed. Warnock's subdivision would terminate whenever areas were found for which a reasonable subdivision edge is unavailable, e.g., when 2 or more minimum bounding boxes completely surround the subdivided area. Obviously, areas with none or only a single minimum bounding box require no further processing. The early stages of Warnock's subdivision might be done on a disk-to-disk basis if core memory space is not available for the entire problem set.

b) For each sub-area in which 2 or more polygons are present, select from those polygons the edges which lie within the area in question. Compute all intersections of such edges and record those intersections by subdividing the edges at the intersections thus discovered. Notice that the edges need not be clipped to the area in question - merely selected on the basis of penetrating the area.

c) For areas which contain one or more polygons and one or more symbols, clip the polygons to the minimum bounding box of the symbol. This clipping step is important because it may later on permit us to recognize the similarities between the interactions of polygons and various instances of the same symbol. Record the clipped polygons transformed into the symbol coordinate sustem as interacting with that instance of the symbol.

For each such polygon-to-symbol interaction, examine all previous interactions to search for similar problems. Thus, for example, a ground wire which extends across a row of symbols would be clipped into sections whose length matches that of the symbol and these sections would appear as identical problems in each instance of the symbol. One might prefer to do this step using the "blob" of the symbol rather than its minimum bounding box, but early recognition of problems as being identical is probably worthwhile.

d) For areas in which two or more symbols overlap but no polygons, record those areas in both polygons as problem areas. Again, search for identical problem types, since a row of symbols which overlap is relatively easily recognized as a collection of identical overlapping areas.

e) In each symbol which has problems, expand the symbol and address those problems along with potential design rule problems of the symbol itself by recursive call to the procedure already described.

An Overlap Detector
Ivan Sutherland & Bob Sproull
January 19, 1978

SUMMARY

1. The horizontal and vertical subdivisions called for in the Warnock algorithm subdivision process capitalize on the many horizontal and vertical lines which appear in circuits. The Warnock algorithm will be all that is required over the many areas of the chip in which nothing much overlaps anything else. The low depth number of the data ensures that the Warnock algorithm will be effective.

2. The search for identical problems associated with similar symbols can make the task of examining the content of symbols substantially simpler. It is made practical by the low depth number; only a few players appear in any problem area.

3. The use of unexpanded symbols is a major factor which we must use to improve the performance of the system.

IES/slr:1322

Reference List:

DF	1129	Broadening Polygons	Nov. 9, 1977
DF	1137	Geometry of Circles	Nov. 9, 1977
DF	1146	Computing Wrap Number	Nov. 9, 1977
DF	1164	Self Intersections of Polygons	Nov. 23, 1977
JIC	1247	Intersections	Jan. 23, 1978
JIC	1263	Angular Comparisons	Jan. 1, 1978
SSP	1316	A Design Rule Checker	Dec. 27, 1978
SSP	1322	An Overlap Detector	Jan. 19, 1978

Contents

This memorandum is intended to describe the polygon package on which Jim Rowson and I have been working for the past several months. This memo follows the order of the actual software packages and is intended to be a guide to the subroutines involved. Some theory and data definitions are included and reference is made to other memos in which more theory can be found.

DATA REPRESENTATION:

In the polygon package each POLYGON is represented as one or more SHEETs. Each SHEET is a closed ring of alternating EDGEs and VERTEXes. Each EDGE refers to an EQUATION which is either a CIRCLE or a LINE depending on whether the EDGE is straight or curved. Each VERTEX refers to a POINT with X and Y coordinates. Many VERTEXes may refer to the same POINT; indeed during processing POINTs with identical coordinates are discovered and merged. These relationships are shown in Figure 1.

When it is important to know which EDGEs begin or terminate at a particular POINT, edge references, called "EDGREFs", are added to the structure as shown in the bottom of Figure 1. Because straight EDGE P begins at POINT 1 and straight EDGE R terminates at POINT 1, POINT 1 acquires two EDGREFs as shown. Similarly, POINT 2 aquires two EDGREFs which show that EDGE P terminates at it and that EDGE Q begins at it. When several SHEETs cross at a single POINT that POINT aquires more than two EDGREFs which serve to link the separate SHEETs together.

THE PROGRAMS:

The programs which work on these structures, and this memorandum which describes them both follow the outline suggested in Figure 1. The program is broken into sections as shown below; this memo will follow the same format.

RINGS.SIM — Provides the basic data forms shown by horizontal arrows in Figure 1. It is rather like SIMSET, but for rings rather than for lists.

ARITH.SIM — Provides the basic arithmetic capability. Defines POINTS, CIRCLES, and LINES.

EDGES.SIM — Provides for line and circle segments called EDGES, and defines VERTEXes.

EDGREF.SIM — Defines a number of types of EDGREFs.

SHEETS.SIM — Defines the operations which can be performed

on closed sets of EDGEs, e.g. area, minimum bounding box, perimeter, etc.

POLYS.SIM Manipulates sets of EDGEs to find self intersectins, overlapping areas, etc.

NOTATION:

When a routine appears in a class it may do something to an element of that class. I call such an element the "subject". The routine may also do things with other elements which are passed to it as parameters. I call such elements "parametric" elements. Thus, for example, the OUT procedure in RINGs removes the subject element from whatever ring it is in and returns a pointer to some other element in that ring if such an element exists.

DEBUGGING:

A number of routines called DEBUGxxxx(A,B,C . . . .) are used to provide intermediate print outs during debugging. All such routines begin with the letters DEBUG. All of them print out on SYSOUT. Had I to do it all again, I would provide a debug output file separate from the regular output file and separate from SYSOUT. DEBUG routines analyze various troubles and print out various data types and numerical values. A series of routines called DEBUGNODE can be used to analyze which type of node one has got. DEBUGNODE routines appear in several places as the variety of nodes increases. Nearly every class has a DEBUG routine which prints out pertinent information about subject elements of that class. They will be described as I come to them.

RINGS.SIM

Like SIMSET, RINGS.SIM provides mechanisms for inserting and deleting elements from sets. Unlike SIMSET, RINGS.SIM deals with elements which form sets without a unique first or last element. The "head" of a ring is considered quite separate from the ring members. This difference from SIMSET simplifies representation of closed collections of EDGEs, and simplifies representation of VECTORs which are sorted sequentially around a POINT. It suffices also for unordered lists of things such as the SHEETS of a POLYGON. Unlike SIMSET, the pointers in RINGS.SIM are externally accessible. Hidden pointers would be an improvement, I think.

RINGS.SIM defines two major classes: RINGER and DOWNRINGER. A RINGER is a member of a ring set. It has a forward and a backward pointer called SUC and PRED respectively, which point to other RINGERs. A DOWNRINGER is a RINGER with an additional pointer called DOWN which points to a ring of RINGERs. In Figure 1, the PRED and SUC pointers come out at the sides and near the top of the boxes. The DOWN pointer comes out the bottom of the boxes. Other pointers come out in other places. Thus it is evident that POLYGONs, SHEETs, and POINTs are represented with DOWNRINGERs while VERTEXes, EDGEs and EDGREFs are represented with simple RINGERs.

Class RINGER has four procedures. OUT removes the subject RINGER from its ring. PUTBEFORE and PUTAFTER put the subject RINGER into the same ring as the parametric RINGER, adjacent to it. PUTINTO puts the subject RINGER into the last position in the ring referred to by the parametric DOWNRINGER. OUT returns a pointer to another ringer in this same ring if any such ringer exists. PUTINTO, PUTBEFORE and PUTAFTER all return a pointer to the subject ringer.

Now two comments. First, OUT is never used in any of the remaining code. It turns out that a RINGER is always in the ring of some DOWNRINGER, but OUT has no way of knowing which one. Were the DOWN pointer of the DOWNRINGER pointing to the subject RINGER when OUT was called, the DOWNRINGER would loose track of its ring. Instead of using OUT in Class RINGER, the rest of the code uses TAKE in Class DOWNRINGER which arranges the DOWN pointer properly.

Second, PUTBEFORE and PUTAFTER have been carefully written so that they can put a whole ring into another ring. If the subject RINGER happens to be a member of a ring, that entire ring is inserted adjacent to the

parametric RINGER. Order is preserved. The subject ringer comes out in the adjacent position with the rest of its ring, in order, adjacent to it but on the opposite side from the parametric RINGER. This feature was introduced for manipulating multiple intersections of circles, but has subsequently fallen into disuse. PUTBEFORE and PUTAFTER are, in fact, used only to insert single elements into rings.

Class DOWNRINGER is the equivalent of the SIMSET Class HEAD. A DOWNRINGER refers to a ring of RINGERs as well as being a RINGER in its own right in some other DOWNRINGER's ring. In a tree composed of DOWNRINGERs, specific pointers, SUC, PRED, and DOWN exist for "next brother", "previous brother", and "first son".

Within Class DOWNRINGER the integer procedure CARDINAL tells how many members there are in the DOWN ring. Boolean procedure EMPTY is true when the DOWN ring is empty, i.e. when DOWN==NONE. The procedure TAKE is used to remove the parametric RINGER from the DOWN ring of the subject DOWNRINGER. It fixes up the DOWN pointer so as not to loose the ring and makes DOWN==NONE when the ring becomes empty.

Rings have no beginning and no end. Nevertheless it is often useful to visit each member of a ring exactly once. The DOWN pointer which identifies the ring can be used as a once around marker. The two procedures NEXT and ENUMERATE are used to generate the members of a ring in sequence. NEXT(r) returns the "next" RINGER in the ring. NEXT(NONE) is defined to be the RINGER pointed to by DOWN. NEXT returns NONE if the next element would have been pointed to by DOWN. ENUMERATE calls the parametric procedure once for each member of the ring, passing the RINGERs involved to the procedure as parameters.

One comment. CARDINAL and EMPTY have not proven very useful. It seems simpler to tell when a ring is empty by checking that DOWN==NONE. In the polygon code the actual number of elements in the ring is not very important. One element in the ring can be detected by PRED==THIS RINGER. Two elements can be detected by PRED==SUC. Three or more need to be counted.

Debugging aids in RINGS.SIM

DEBUGTEXTL(say) prints out whatever text you put in "say" as a line on the debug listing. It often appears to herald entrance to a subroutine. DEBUGONE, DEBUGTWO, DEBUGTHREE and DEBUGSIX are all used to print out numeric values with a comment. READINCUE, READINTEGER, and READINREAL are used to provide cues for manual input when

the input device is SYSIN but not when reading from a file. They return the value obtained. All of these debugging aids have nothing to do with RINGS.SIM; they appear in RINGS.SIM simply as a clerical convenience.

ARITH.SIM

ARITH.SIM contains the basic arithmetic elements. It defines Class POINT, Class EQUATION with its subclasses LINE and CIRCLE, and it defines minimum bounding boxes in Class BOX. Scalar and vector quantities are defined as Class VECTOR and Class SCALAR, both subclasses of Class SORTABLE which contains the basic sorting mechanisms. All of the hard arithmetic stuff is here: intersections, minimum bounding boxes, and sorting.

An un-classed (or is that unclassy?) procedure called ROTSEQ appears at the start of ARITH. It takes six parameters which are the X and Y components of three vectors. It returns +1 if the three vectors have a counterclockwise rotation sequence, -1 if they have a clockwise rotation sequence, and 0 if the rotation sequence cannot be determined because some of the vectors have identical directions. This procedure implements the ordering procedure described in JIC memo #1263, "Angular Comparisons". It appears again in slightly different form in Class VECTOR. The unclassy version is not used; it remains only for you to see it in pure form.

CLASS POINT:

POINTs have two real coordinates called X and Y. Because POINTs are represented as DOWNRINGERs, they also have the DOWN pointer which is sometimes used to refer to a set of EDGREF blocks as shown at the bottom of Figure 1. To simplify debugging, each POINT also contains an index number called COUNT which is automatically incremented as POINTs are created.

There are several procedures in Class POINT. The procedure READIN gets two real values for the X and Y coordinates. PRINT prints out not only the coordinates, but also the index number and cardinality of the POINT. The cardinality is important because it tells how many EDGREFs there are and thus how many EDGEs are known to begin or terminate at this POINT. DEBUG heralds a POINT and prints its coordinates.

INTOBOX expands the parametric minimum bounding BOX to accomodate the subject POINT. The alternative of having a BOX know how to expand itself to accomodate a POINT is possible, but leads to excessive knowledge by BOXes of different data types. Maybe I should have made an XMIN, XMAX, YMIN, and YMAX procedure for each data type and let BOX call them appropriately. Anyhow, poor Class BOX gets pushed around by everyone else.

Points can be EQUAL if they are identical or if their X and Y coordinates are equal. I have avoided making a tolerance test for equality, but we may yet be forced to do so. A special kind of equality test, ABSORBS is used to combine POINTs which have been generated separately but which happen to have identical coordinates. It is particularly important to detect and combine POINTs in separate SHEETs which happen to have equal coordinates. Note that the ABSORBS procedure does not attempt to combine the DOWN ring; the DOWN ring should be empty when POINTs are combined.

PDIST computes the distance between two POINTs by the usual square root method. The only place I can think of this being used is in the perimeter computation, but there the square root could be avoided by using the fact that line equations are already normalized. I suspect that the distance computation is one of those things that got put in for completeness and is not used!

The procedure CROSS, which takes two parametric POINTs, is an ugliness left over from a previous generation of programming. It is used only in the WRAP computation for circle segments. WRAP should be revised and CROSS should be deleted. CROSS computes the z component of the cross product of the displacement vectors from the subject POINT to two parametric POINTs. WRAP discovers which way a circular arc wraps a POINT by comparing the sign of this cross product with the sign of the radius of the circle. Ugly.

DISPLACEDBY is used during the inflation process to get a new POINT inside or outside the subject POINT. The new POINT is a parametric distance d in the direction indicated by a parametric vector v.

LINETO is used by the polygon construction routines to generate the parameters of a LINE between the subject POINT and a parametric POINT. If one uses LINETO for generating a LINE between two equal POINTs, the normalization routine in Class LINE will be given a too challenging task and will complain. This should be fixed by arbitrarily defining some direction, say vertical, for LINEs of zero length.

Three unimplemented procedures, LINEFROM, VECTORTO, and VECTORFROM, are not anywhere used, but might easily be defined. For now we should save the space.

CLASS EQUATION, CLASS LINE and CLASS CIRCLE:

There are two kinds of EQUATIONs, LINEs and CIRCLEs. Class EQUATION is merely a holding Class for all of the virtual procedures that will appear in LINE and CIRCLE. EDGEs, of course, will each reference an EQUATION, whose type will determine whether the EDGE is straight or curved. In what follows I will describe the procedures which appear both in LINE and in CIRCLE and note any differences.

Class LINE defines three real quantities, a, b, and c. These are the coefficients of the line equation a*x + b*y + c = 0. The first two, a and b, are normalized so that a**2 + b**2 = 1. The NORMALIZE procedure in Class LINE is called as LINEs are created to do this normalization. The coefficients a and b form a unit normal which points outward from straight EDGEs. The "value" of a POINT with respect to a LINE is a signed quantity. The value of a POINT {x,y} with respect to a LINE {a,b,c} is computed as a*x+b*y+c. The outward pointing convention for a,b insures that the value of a POINT is positive when the POINT is outside the LINE. The distance from a POINT to a LINE is the absolute magnitude of the value. See PVAL and CCVAL four paragraphs below.

Class CIRCLE also defines three real quantities, x, y, and r. The first two, x and y, are the coordinates of the center of the circle. The third, r, is the radius. Positive r indicates that the circle goes around counterclockwise; negative r indicates that the circle goes around clockwise. Thus r is used to indicate rotation direction in the mathematical sense. Because polygons are also traversed counterclockwise, positive radius CIRCLEs represent round corners and negative radius CIRCLEs represent fillets. The "value" of a POINT with respect to a CIRCLE is a signed quantity. The value of a POINT inside a positive radius CIRCLE is negative, and can assume a least value of -r when the POINT is at the center of a CIRCLE. The value of a POINT at the center of a negative radius CIRCLE is positive. This convention is entirely consistent with the value convention chosen for LINEs.

DEBUG and PRINT simply print out some parameters of the LINE or CIRCLE, distinguishing between them.

INTERSECT computes the intersection POINT(s) of the subject EQUATION with the parametric EQUATION. Because intersections involving CIRCLEs can produce two answers, all intersection procedures return two answers, one of which may be NONE if there is only one intersection, or both of which will be NONE if there are no intersections. The LINE/LINE computation appears only in Class LINE, the CIRCLE/CIRCLE computation appears only in Class CIRCLE and the LINE/CIRCLE intersection computation appears in both classes. The

resulting POINTs are returned by means of NAME variables in each procedure. The coding of these procedures follows the geometry described in JIC memo #1247, "Intersections".

EQUATIONs can be COLINEAR. Because EDGEs which face in either direction may overlap, COLINEAR accepts either identical equation parameters or the negative values. A special form of COLINEAR called ABSORBS is intended to combine LINEs and CIRCLEs when multiple instances of them are found with the same values. Just as POINTs have EDGREFs associated with them, I have been thinking of having vertex references, VERTREFs, associated with LINEs and POINTs. Such VERTREFs might be sorted in order along the LINE or CIRCLE. VERTREFs are not implemented at present, and the ABSORBS procedures for LINEs and CIRCLEs are not in use.

POINTs may be compared to LINEs and CIRCLEs to get a signed measure of the distance from the POINT to the LINE or CIRCLE. Because this is a signed value, the procedures which obtain it are called PVAL and CCVAL, rather than PDIST or CCDIST which would produce unsigned distance measures. PVAL and CCVAL plug in the X and Y coordinates of a parametric POINT and the center of a parametric CIRCLE, respectively, into the error equation for the subject curve.

The distance between a LINE and a BOX is computed by BVAL in Class LINE. BVAL selects the corner of the BOX nearest the LINE and computes its value as plugged into the LINE equation. If BVAL returns zero, the LINE passes through the BOX. If BVAL returns a positive number the BOX lies outside the LINE by that minimum separation. If BVAL returns a negative number the BOX lies inside the LINE by that minimum separation.

The BOX to CIRCLE relationship is computed by BDIST in Class CIRCLE. BDIST computes the minimum distance between the BOX and the CIRCLE, choosing the side or corner of the BOX closest to the circle center. BDIST returns a positive number when the CIRCLE is entirely outside the BOX, and zero otherwise. Neither BDIST nor BVAL are in use at the present. BVAL used to appear in the polygon clipping procedures and should again be used there. BDIST is included for completeness only.

One can measure the positions of POINTs along LINEs and CIRCLEs. Such a measure is valuable for determining if a POINT is internal to an EDGE. For a LINE, this measurement along the LINE is a SCALAR computed by PORD in Class LINE. For a CIRCLE, this ordering is a VECTOR as computed by PORD in Class CIRCLE. In either case the sequence of the PORD values for the end POINTs of an EDGE and some test POINT can be used to tell whether the POINT is interior to the EDGE.

This code should probably be streamlined, since it is hardly worthwhile to set up a new data structure each time one tests whether a POINT is internal to an EDGE. The internal computations for EDGE might well be expanded to do the requisite computations.

DISPLACEDBY in Class LINE and Class CIRCLE are used by the inflation routines to generate EDGEs outside or inside the given EDGEs. The computations for displacing LINEs and CIRCLEs are particularly simple.

CLASS BOX:

A BOX is a minimum bounding box for some figure. Class BOX defines the left, right, bottom and top coordinates of the box. Class BOX initializes these values, so that the first thing put into the BOX will establish a meaningful size. BOXes PLOT as an outline, and PRINT their values. INBOX puts the values of the subject BOX into the parametric BOX. Notice that all expansion of BOXes is done from outside them. I suppose one might want to revise this to make BOXes able to expand themselves. One would then have to provide separate MINX, MINY, MAXX and MAXY routines for each thing which might be put into a BOX. Well, this is the kind of representation problem we have! Notice please that Class LINE and Class CIRCLE cannot put themselves into a BOX. Only EDGES can, since only the EDGEs have finite extent to make putting in a BOX meaningful.

CLASS SORTABLE, CLASS SCALAR and CLASS VECTOR:

There are two kinds of sortable items, SCALARs and VECTORs. SCALARs are sorted by value in the obvious way. VECTORs are sorted by their direction around the origin. The SUC pointer in such a sorted list of VECTORs points to the next VECTOR found in the counterclockwise direction, i.e. the direction of mathematical advance around the origin. Class SORTABLE implements the sorting procedure PUTINSORT which is used for both SCALARs and VECTORs.

PUTINSORT is a simple insertion sort mechanism. It picks a place in the sorted list and calls a routine called SORTSEQ which returns +1, 0 or -1, which indicate respectively: try again further on in the list, put the new item here, and try earlier in the list. Several different versions of SORTSEQ exist for different kinds of things being sorted.

A SCALAR has only a value. The usual DEBUG and PRINT routines are there to illuminate the value. The SORTSEQ routine, like others, tests whether the subject SCALAR lies properly between the two parametric SCALARs. Notice that

because a sorted list of SCALARs is represented as a closed ring, it is possible that SORTSEQ will be asked to fit a new SCALAR into the position between the end and the beginning of the list. The new value correctly fits here only if it is larger than the largest value or smaller than the smallest. This point caused me no little grief in debugging.

Class VECTOR is used to represent two component vectors. The components are called DX and DY. They are left unnormalized. The usual DEBUG and PRINT routines are available. ROTSEQ in Class VECTOR again implements the function described in JIC memo #1263, "Angular Comparisons", to find out whether the direction of rotation in passing sequentially around three vectors is clockwise or counterclockwise. ROTSEQ is used only by the part of Class EDGE which is seeking to find whether a given POINT lies inside a circular EDGE. It might be better to put the code directly in where it is used.

SORTSEQ for VECTORs is based on the same ideas as ROTSEQ. It indicates only whether the VECTOR fits, by returning a zero, or does not fit, by returning a one. It never recommends looking backward in the sorted list, since there is no metric to use. SORTSEQ in Class VECTOR is not, in fact, used, since a more refined form of sort sequence exists in Class EDGREF, the only subclass which is actually sorted.

The LENGTH of a VECTOR can be computed by the obvious square root routine. DOT and CROSS compute respectively the dot product and the z component of the cross product of the subject VECTOR and the parametric VECTOR.

EDGES.SIM

EDGES.SIM defines Class EDGE and Class VERTEX from which SHEETs are made. Because EDGEs can be either straight or curved and the distinction is made only on the basis of the type of the equation referenced, many of the routines in Class EDGE have two parts, one for straight EDGEs and one for circular EDGEs. This ugliness might be avoided by making two sub-classes.

EDGEs are said to have a "beginning" and a "termination". These words are used to avoid the words "start" and "end" which are preempted by the programming language. Thus in Class EDGE you will find procedures BP and TP which find the Beginning POINT and Termination POINT of the subject EDGE respectively. The prefixes B and T are used in other places as well.

In EDGES.SIM we see the first of the DEBUGNODE routines which analyzes the type of the node that has been printed. Then in Class EDGE itself we find the usual DEBUG routine and a PRINT routine which depends on the print routine for the EQUATION on which the subject EDGE rests.

CLASS EDGE:

There are three kinds of plotting routines. PLOT is used to plot the subject EDGE as one of a sequence of EDGEs. It assumes that the pen has been moved to the beginning of the edge. PLOTME is used for isolated edge plotting. It moves the pen to the beginning POINT, plots the subject EDGE and then lifts the pen. I can't remember why I put it in, which suggests that it is probably no longer needed. Finally BOXPLOT plots the subject EDGE and its minimum bounding box. I put it in to test the INTOBOX procedure which follows.

INTOBOX for EDGEs is somewhat complicated by my desire to get the computation right for curved EDGEs. The basic idea is that if a curved EDGE straddles the X or the Y axis, then it will have an extreemum on that axis. If it doesn't straddle the axis and it goes the long way around, see procedure ISLONG, then it has two extreema on that axis.

Putting a straight EDGE into a BOX is much simpler. Straight EDGEs merely put the terminal POINT in the BOX, since the entire ring of EDGEs will be put into the BOX in sequence.

INBOX tests if an EDGE might conceivably be in a BOX. It treats circular EDGEs as complete circles and thus will sometimes announce them to be inside BOXes in which only their extensions lie. INBOX is not at present used for anything.

An EDGE is said to be "degenerate" if its beginning and terminal POINTs are equal. A routine called DEGENERATE returns TRUE if this is the case. Another routine called CLEAN is used to delete degenerate EDGEs. If CLEAN returns FALSE, the subject EDGE has been removed. CLEAN is used by a procedure in Class SHEET to eliminate unnecessary EDGEs. Class VERTEX has a CLEAN procedure also to eliminate unneeded VERTEXes. Notice that CLEAN needs the SHEET as a parameter lest SHEET.DOWN end up pointing to the deleted parts.

The intersections of EDGEs are found by the INTERSECT routine. INTERSECT calls on the INTERSECT routine of the EQUATIONs on which the EDGEs rest and then checks that the POINT or POINTs returned are interior to the EDGEs. Like the INTERSECT routines in Class CIRCLE and Class LINE, these INTERSECT routines return any valid intersection POINTs in NAME variables. They also return TRUE if there are any intersections. I believe that some effort devoted to simplifying the subroutine structure of the EDGE INTERSECT routine would be well spent. This routine is the heart of the EDGE/EDGE comparisons in the self intersection routine and is heavily used. Actual measurements of time here would be helpful.

A CLIP routine is included to clip EDGEs against infinite straight lines. This routine is not at present being used. It was the first of the intersection routines written and is included mainly for completeness.

UPCROSS and DOWNCROSS are routines called by WRAP to determine whether an EDGE might contribute to the wrap number of a POINT. They test whether the subject EDGE passes from below the parametric POINT to above it and vice versa. See WRAP described later on.

ENDPOINT is used by the INTERSECT routine to find out if a POINT which has been generated as an intersection is at the end of one of the intersecting EDGEs. CONTAINS tests the parametric intersection POINT to see if it is interior to the subject EDGE. Both ENDPOINT and CONTAINS might be combined into a new version of INTERSECT. PASSES and CONTAINS are identical except for their treatment of end POINTs. PASSES is FALSE if the parametric POINT is an end POINT, but CONTAINS is TRUE if the parametric POINT is an end POINT. Given that we have an ENDPOINT routine it may

not be necessary to have both PASSES and CONTAINS. Again, much revision is possible.

ISLONG tests to see if the subject circular EDGE has more than 180 degrees of extent. It does this by computing a cross product of the beginning and termination vectors for the circular EDGE. If the cross product is zero, it tests the dot product to separate the zero degree case from the 180 degree case. A 180 degree arc is long, a zero degree arc is not.

Wrap Number

A POINT may be either inside or outside of a SHEET. The relationship between the POINT and a SHEET of EDGEs is described by the "wrap number". If a SHEET "wraps" a POINT once, then the EDGEs of the SHEET surround the POINT once in the counterclockwise direction. A wrap number of -1 indicates that the SHEET surrounds the POINT in the clockwise direction. Wrap number of zero indicates that the POINT is outside the SHEET. Wrap numbers higher than one are possible if the SHEET crosses itself to go around the POINT twice, as in the center of a five pointed star.

The wrap number of a POINT on the EDGE of a SHEET is undefined. Whenever the wrap number is computed we can be sure that the POINT being tested is not on any EDGE of the SHEET. Thus the WRAP subroutine in Class EDGE can be a bit casual about its treatment of EDGEs which pass through the test POINT. This observation obviates much of the concern of DF memo #1146, "Computing the Wrap Number".

There are two ways to compute wrap number. One can integrate the angle change of EDGEs as seen from the test POINT. This method, known as the SIGMA-DELTA-ALPHA method is somewhat harder to compute than the line crossing method implemented here. The line crossing method counts how many times a semi-infinite line from the test POINT crosses EDGEs of the SHEET. The procedure WRAP in Class EDGE computes an integer which is the contribution of the subject EDGE to the wrap number. This contribution for a single EDGE is always +1, -1, or 0.

WRAP uses the subroutines UPCROSS and DOWNCROSS to test the general direction of progress of the EDGE. It makes additional tests on the PVAL of the POINT with respect to the EDGE to see whether the EDGE passes on the proper side of the POINT. The semi-infinite line for which crossing is being tested lies to the right of the POINT. If the beginning or terminal end of the LINE has the same Y coordinate as the test POINT, then it is declared to be "above" the semi-infinite line; this can be seen in the

"equals" tests in UPCROSS and DOWNCROSS. Notice that the cases where the PVAL of the POINT is zero, i.e. where the EDGE passes through the POINT, can be treated fairly casually, since they should never occur.

The CIRCLE part of WRAP is a last-minute substitution from a previous incarnation of the program. It should be rewritten. It is the sole user of CROSS in Class POINT, a use which should be discontinued, since POINTs have no inherent cross product function.

The LINE part of WRAP depends on the fact that EDGEs progress along LINEs in such a way as to have the EQUATION components [a,b] be an outward pointing unit vector. EDGEs whose beginning and terminal POINTs have gotten reversed somehow will not work in WRAP. Whether this is a defect or not I do not know.

All EDGEs have a curvature which is their reciprocal radius. Curvature has the nice property that it relates CIRCLEs and LINEs properly. Imagine several EDGEs progressing generally upward from a POINT with the same beginning direction, as shown in Figure 2. The center one is straight, the one on the left has positive radius and thus curves towards the left, the one on the right has negative radius and thus curves towards the right. The curvature of the center edge is zero. EDGEs with larger and larger positive curvature lie to the left of the straight EDGE with tighter and tighter turns. Negative curvature EDGEs lie to the right of the straight one; as the curvature gets more negative, they curl ever more tightly away from the straight EDGE. EDGEs placed into sorting order by curvature in such a bundle will form a shief whose order corresponds naturally to the obvious geometric order of the EDGEs. The procedure CURVATURE merely computes reciprocal radius, inserting zero for straight EDGEs.

MAXY and MINY compute the maximum and minimum Y extent of the subject EDGE respectively. They wory correctly about curved EDGEs just as INTOBOX did. By checking whether the EDGE straddles the X coordinate of its center, they detect internal maxima for the edge.

The AREACONT and PERIMCONT routines compute the area and perimeter contributions of an EDGE respectively. For curved EDGEs, both routines make reference to the SECTORANG routine to compute the angular extent of the circular EDGE in radians. The area contribution formula for straight EDGEs involves computing the area of the trapizoid between the EDGE and the X axis as shown in Figure 3a. The area of this trapizoid for a straight EDGE is (X2 - X1)*(Y2 + Y1)/2. Actually the area contribution as computed in Class EDGE is

twice that, since the division by two can be done after accumulating the contributions from all EDGEs in a SHEET. For curved EDGEs, the area contribution is the area under the straight line segments, shown dotted in Figure 3b, joining the beginning and termination of the circular EDGE to its center, plus the area of the "piece of pie" defined by those imaginary line segments and the CIRCLE boundary itself. After algebraic simplification you get the computation shown. Again a double value is computed.

The perimeter contribution is substantially simpler. For straight EDGEs it is just the length of the EDGE. For curved EDGEs it is the absolute value of the radius times the sector angle. Notice that in SECTORANGLE, the sector angle is computed as the arctangent of the ratio of the cross product and dot product of the beginning and termination sides of the angle. By using the dot and cross product, the computation gets away with a single arctangent computation. SECTORANGLE is not used in the plotting routines; it probably should be.

The four procedures BP, TP, BV and TV find the Beginning or Terminal POINT or VERTEX of the subject EDGE by following the topology of the structure. Since each EDGE in a SHEET is preceeded and followed by a VERTEX, BV and TV are especially simple. BP and TP have two steps to go as can be seen in Figure 1.

BEGNORM and TERMNORM find the outward pointing normals to an EDGE at its beginning and its termination. The corresponding routines DBP and DTP, which are acronyms for displaced beginning POINT and displaced terminal POINT, each return a POINT displaced perpendicularly to the EDGE a parametric distance d from its beginning or terminal POINT respectively. DBP and DTP are used in the inflation routine. The same computations as are found in BEGNORM and TERMNORM will also be found in the initialization procedures for vector edge references in Class VEDGREF.

The self intersection routine cuts EDGEs wherever intersections are found between them. Two new VERTEXes are created for each such pair of EDGEs. The new VERTEXes rest on the same identical intersection POINT. Because the two new halves of the EDGE might themselves intersect with other EDGEs, they must be treated rather carefully. CUTAT checks that the parametric POINT is internal to the subject EDGE, refusing to cut if the POINT is not internal. CUTAT must create two EDGEs where only one existed before. The first is made by modifying the subject EDGE to give it a new terminal VERTEX. CUTAT creates and returns the other new EDGE. CUTAT returns NONE if no cut was possible.

Improvement in performance is possible here. CUTAT checks that the parametric POINT is interior to the EDGE. Since the intersection routine has already made this check it might be omitted in CUTAT except that avoiding cuts at the beginning and termination POINTs of the EDGE is probably important. INTERSECT permits intersections to be at the end of one EDGE, but not both, but CUTAT must avoid any end cuts. Some work done here could improve performance.

CLASS VERTEX:

Each VERTEX can know the SHEET to which it belongs. These pointers, called MYSHEET are used by the topology routines to mark where work has been done. They are otherwise unused. They can be seen in Figure 1 as the upward pointing arrows out of the VERTEX blocks.

The usual DEBUG and PRINT routines exist for VERTEXes. Notice that PRINT for a VERTEX prints the "bend" of the VERTEX. This is a number which is 1 for a leftward tangent direction change, -1 for a rightward tangent direction change and 0 for a continuation of direction through the VERTEX. BEND is 2 if the EDGEs at this VERTEX make a 180 degree direction change i.e. if they double back.

The procedure KILLEDGREFS is used to delete all of the edge references (EDGREFs) associated with the POINT which belongs to a VERTEX. It could probably be simplified to merely make the DOWN pointer of the POINT be NONE, but I am not sure how the garbage collector treats isolated rings.

CLEAN, as you would think, checks to see if the preceding and following EDGEs have identical line equation, combining them if equal, via the call on ABSORBS. If the EDGEs are colinear, CLEAN deletes one of the EDGEs and the subject VERTEX! Note that the loop generator which is calling CLEAN will have to accomodate this deletion. Notice also that CLEAN needs the parametric SHEET as context in which to "TAKE" unneeded parts lest SHEET.DOWN end up pointing to deleted parts.

EDGREF.SIM

At the beginning of EDGREF.SIM you will find another DEBUGNODE routine which knows about more types. The order of the tests in the INSPECT statement is important since only one of them will be performed. Thus superclasses appear after their subclasses in the sequence. EQUATION, for example, comes after LINE and CIRCLE.

In this section of the code two major classes are defined, SEDGREF and VEDGREF. These are scalar EDGREFs and vector EDGREFs respectively. Scalar EDGREFs are used for the Y sort in the self intersection routines. Vector EDGREFs are used to sort around POINTs for the topological analysis which finds unneeded SHEETs when POLYGONs intersect. Two subclasses of VEDGREF exist, one for referring to the beginning of an edge and one for referring to the termination of an edge. Obviously they are called BVEDGREF, and TVEDGREF respectively.

CLASS SEDGREF:

Aside from the usual DEBUG routine, there is only one routine in SEDGREF. This routine, called CROSSCUT, is the working part of the edge/edge comparison in the self intersection computation. CROSSCUT has been very conservatively written to test everything in sight all possible ways to avoid errors of ommission. It can probably be simplified a lot by a more careful analysis.

CROSSCUT has three parts, an "end POINTs" part, an "intersection" part and a "colinear" part. The end POINTs part compares the end POINTs of the EDGEs to see if any of them can be ABSORBed by others. This part guarantees that any VERTEXes that have common values will use identical POINTs so that the EDGREFs at that POINT may refer to all EDGEs that meet at that POINT. There is probably about twice as much testing in the POINTs part as necessary.

The intersection part computes the intersection(s) between the two EDGEs and then cuts each EDGE into as many as three parts. The cutting process is slightly complicated, because one does not know after the first cut whether the earlier or later part is the one cut by the second intersection. Fortunately CUTAT will ignore instructions to cut at an empty POINT or at a point not interior to the EDGE, and so one simply does all possible cutting.

The colinear part cuts each EDGE at the beginning and teremination POINTs of the other EDGE. Again the cutting is complicated because after cutting at the beginning POINT one

is not sure whether the first or second such part will be cut by other POINTs. The routine simply cuts at every POINT in sight and lets CUTAT filter out the irrelevant cuts.

CLASS VEDGREF:

The main thing in Class VEDGREF is the complete version of SORTSEQ used to determine where VEDGREFs go in an angularly sorted list. Each VEDGREF has a reference to an EDGE and the sorting is to place the EDGEs in angular sequence around the POINT at which they meet.

The general strategy is to distinguish EDGEs first on the basis of their tangent directions. If tangent directions are identical, EDGEs are distinguished by their curvature. If they have identical curvature they are colinear and thus presumably identical. Identical EDGEs must still be sorted in a consistent way at both ends. Thus SORTSEQ will look at the sorting done at the other end of the edge, if any, and make a matching sort at this end.

Now, while the beginning and terminal normals, BEGNORM and TERMNORM, computed in Class EDGE were always outward pointing, vector EDGREFs are attempting to model the direction in which EDGEs enter a POINT. Again normals are used, but to be consistent, terminal EDGREFs must point in the inward direction. Similarly, the curvature of terminal references must be reversed. Again sorting is done mainly on the basis of the cross product method. It is interesting to note that when cross products do not discriminate adequately, differences in curvature can play the identical role. This routine should be the subject of a separate SSP memo shortly; its development was hard and the ideas in it are rather nice.

Beginning EDGREFs (BVEDGREFs) and terminal edge references, (TVEDGREFs) differ only in minor details. Their debug routines print B or T respectively. Their OTHEREND routines find the matching EDGREFs at the other end of their EDGE by searching the ring of references at the other end POINT. Which end is searched must obviously be different. Finally, Class TVEDGREF initializes with the inward pointing vector rather than the outward pointing vector.

SHEETS.SIM

CLASS SHEET:

Here are the programs which deal with a single set of connected EDGEs. A SHEET is a DOWNRINGER with a COUNT. The COUNT is the computed wrapnumber for POINTs just to the left of the EDGEs of the SHEET. Thus a COUNT of one identifies SHEETs which separate empty space on the right side of their EDGEs from singly wrapped space on their left. Both SHEETs of an anulus have COUNT of one.

There are two kinds of debugging aids for SHEETs. Each is intended to shed light on the propriety of the sorting order of the EDGREFs at the VERTEXes of the SHEET. DEBUGE simply prints out the EDGREFs in order around each POINT at each VERTEX. If one calls DEBUGE for each SHEET of a polygon, one gets a redundant listing because each POINT is listed again for each VERTEX which references it. DEBUGL is intended to show how vertices are rearranged during the topological search for new SHEETs. For each TVEDGREF at a POINT it prints out the EDGEs which connect to that VERTEX.

The READIN procedure for SHEETs is able to make SHEETs with at least two VERTEXes and only straight EDGEs. The two VERTEXes must be different. SHEETs with curved EDGEs are made by inflating such input SHEETs.

The PRINT procedure for SHEETs makes a heading for them which matches the PRINT procedures of their constituant parts. SHEETs are printed out as an alternating set of EDGEs and VERTEXes by the inspect statement at the end of the SHEET.PRINT.

Three kinds of plotting are available for SHEETs. PLOT simply plots the EDGEs of the SHEET in whatever color has been established by the POLYGON plotting procedure. TPLOT, for test plot, selects a color for the SHEET according to the wrap number of the SHEET. As presently established, SHEETs with wrap number of one are plotted in black, SHEETs with wrap number zero or less are plotted in red, and SHEETs with wrap number two or more are plotted in green. SHEETs with zero area are plotted in blue. Only the black sheets are meaningful output. EDGBOXPLOT plots each EDGE with a private minimum bounding box; it was written to check the INTOBOX routines.

The CLEAN procedure for SHEETs at present inspects each EDGE or VERTEX and CLEANs it. Because the CLEAN procedure for an EDGE or VERTEX may delete it, the looping structure in SHEET.CLEAN uses two variables, r and rr. The rr pointer moves forward around the sequence ahead of the r pointer.

If a SHEET element is found to be OK because its CLEAN procedure returns TRUE, then r is advanced to that element. The rr pointer will not be meaningful if the CLEAN procedure deleted the element to which it points. This is an example of a general difficulty with generators when the procedures being called change the number of elements in the list being generated. A similar difficulty might well appear in INFLATE, but does not.

Each POINT in the structure may have a ring of EDGREFs to tell about the EDGEs which terminate or begin there. The MARKVERTICES procedure generates these EDGREFs and places them into sorted order at each VERTEX. The PUTINSORT procedure used will be found in Class SORTABLE in ARITH.SIM, but will use the SORTSEQ procedure for vector EDGREFs found in Class VEDGREF in EDGREF.SIM.

WRAP takes a single POINT as a parameter. It computes an integer which is the wrap number of the parametric POINT for the subject SHEET. Because of the behavior of the individual EDGE contributions to the wrap number, WRAP provides uncertain answers if the POINT lies on an EDGE of the sheet. WRAP computes the wrap contribution of the subject SHEET to the parametric POINT's total wrap count. Contributions from several SHEETs will be accumulated in Class POLYGON. The COUNT variable of the subject SHEET was computed by calls on WRAP for other SHEETs.

AREA and PERIMETER merely add up the area contributions and perimeter contributions of individual EDGEs. The factor of two in the area sumation was noted in AREACONT in Class EDGE.

NEXTEDGE and NEXTVERTEX are generators which, like NEXT in Class DOWNRINGER make it easy to make loops involving all elements of a SHEET. Because EDGEs and VERTEXes alternate in sheets, these two generators must skip over elements of unwanted types. The DOWN pointer of the SHEET may point to either an EDGE or a VERTEX.

Poor Class BOX gets putinto all the time. PUTINTOBOX expands the parametric BOX to hold the subject SHEET. Polygons have minimum bounding boxes and thus PUTINTOBOX is used to enter a new SHEET into a POLYGON. After expanding the box, the subject SHEET is putinto the POLYGON's DOWN ring.

There are three procedures which generate new SHEETs corresponding to modified versions of old SHEETs. These are REVERSED, INFLATED, and CLIPPED. Corresponding generators in Class POLYGON will call these procedures for each SHEET. REVERSED, which is not implemented as of February 1978, is

intended to reverse the direction of the SHEET. Reversal of direction is important for subtracting one POLYGON from another. REVERSE will generate an identical set of EDGEs and VERTEXes, but in the reverse order. It will also produce a new set of EQUATIONs which are reversed in sign. INFLATED expands or contracts the SHEET by the real parameter, d. INFLATED uses the DISPLACEDBY routines, previously noted in Class EDGE and Class POINT to compute an entirely new SHEET of larger or smaller dimensions. CLIPPED produces an entirely new SHEET which is the part of the subject SHEET on the inside of the parametric LINE. The normal vector for the LINE points towards the part of the SHEET which will be saved. CLIPPED may produce very strange results for POLYGONs with more than one SHEET, or for SHEETs which cross the clipping line many times; it is well to follow the clipping procedure with the selfintersection procedures.

CLASS TRACK:

Class TRACK represents the centerline of a track whose width is given. Such a track is represented as an alternating series of VERTEXes and EDGEs beginning and ending with a VERTEX. Class TRACK knows only how to convert itself into a SHEET by tracing itself forwards and backwards; note the double use of WHILE in the loop generator. The resulting proper SHEET is then inflated. Presumably the capabilities of Class TRACK will be expanded as time goes on.

POLYS.SIM

Here we have the polygon handling stuff and the major hardworking procedures in the package. I shall outline them in order of appearance, saving some comments on performance for the very end.

POLYS.SIM starts off with another, more capable, node analizer called DEBUGNODE. Then there are three ordinary generators which call corresponding routines in Class SHEET: READIN, PRINT, DEBUGE.

CLASS POLYGON:

POLYGONs can be plotted in a variety of ways. PLOT simply plots the SHEETs of the POLYGON in whatever is the going color. GPLOT plots only the good SHEETs, i.e. those with non-zero area and a wrap count of one. TPLOT calls the TPLOT routine in each SHEET which, in turn, chooses a color appropriate to that SHEET's wrap count and area. BOXPLOT plots the POLYGON and its minimum bounding BOX. EDGBOXPLOT plots each EDGE and the minimum bounding BOX for that EDGE. EDGEBOXPLOT was written to debug the minimum bounding box routines.

CLEAN simply calls on the CLEAN procedure of each SHEET. It may be that CLEAN will eventually do more than this, for example delete zero area or bad wrap number SHEETs, but my confidence level is not yet high enough for that.

FIXEDGREFS first deletes any remnant EDGREFs and then builds new ones by calls on SHEET.MARKVERTICES.

WRAP, AREA and PERIMETER are merely generators and accumulators for the sub computations performed at lower levels.

NEXTSHEET is another generator which visits all of the SHEETs in a POLYGON exactly once. NEXTEDGE is supposed to visit all of the EDGEs in the polygon exactly once, but has fallen into disuse and may not be correct.

CENTROID computes a POINT which is the centroid of the polygon. It uses NEXTEDGE and thus may not be correct. I used to have a use for the centroid of polygons, I no longer have.

REVERSED, INFLATED, and CLIPPED are merely calls on the lower level routines for similar purposes. Note that the new SHEETs which are produced are placed into the new POLYGON with the PUTINBOX routine in Class SHEET, thus

updating the minimum bounding BOX for the new POLYGON. At one time CLIPPED compared the test line with the minimum bounding box of the POLYGON to be clipped. This code should be put in again.

Self Intersection Procedure

Here is the big news. The SELFINTERSECTIONS routine in Class POLYGON proceeds in three stages. First it sorts the EDGEs by southmost Y value. Then it moves the EDGEs into an active list by: Second, culling from the active list any whose northmost Y value is less than the southmost Y value of the newly entering EDGE, and third, testing the newly entering EDGE against all EDGEs in the active list.

Because the process of comparing new EDGEs with those in the active list may cut the new EDGE into many parts, a sublist called "newcut" is kept to hold such parts. Thus as EDGEs move from the sorted list into the active list, parts of them may end up on the newcut list. The newcut list is always exhausted by comparing its parts against the active list before another EDGE is taken from the sorted list. Because the newcut list is not sorted by Y value, the culling step is omitted for EDGEs coming from the newcut list.

The sorting process makes use of scalar edge references (SEDGREFs); remember them? In the sorting loop a new SEDGREF is made for each EDGE. The SEDGREF value corresponds to the minimum Y value of the EDGE. These SEDGREFs are PUTINSORT using the routine in Class SORTABLE. In order to put subsequent elements into the same sorted list it is necessary to fill SORTED.DOWN if it is empty. Thus the second statement in the procedure is used only for capturing the first sorted element into the sorted list. The final statement moves the SORTED.DOWN pointer to the smallest list element. This is probably a bad plan. The sorting procedure depends to some extent on the fact that adjacent EDGEs in a SHEET will be placed near to each other in a Y sorted list. It would be better to have a separate search loop after sorting is complete to find the smallest element and then point SORTED.DOWN at it.

Fixsheets Procedure

The sub procedure MATE matches an EDGE which begins at a POINT to an EDGE which terminates at the POINT. MATE works by counting the intervening EDGEs so as to match up EDGEs which belong to the SHEETs with equal wrap count. Operation of MATE is shown in Figure 4. MATE also rearranges the EDGEs and the VERTEXes to group EDGEs which belong in a single SHEET. MATE rearranges four EDGEs and

two VERTEXes so that the mates get a common VERTEX, and the rejected EDGEs end up sharing the rejected VERTEX. MATE returns a pointer to the new VERTEX.

The sub procedure REARANGE is mainly a multiple call on MATE. However, it also marks the newly mated VERTEXes with MYSHEET marks corresponding to the SHEETs they are in, generating new SHEETs as necessary to accomplish this. It puts the proper wrap number counts in any such SHEETs as are generated. Note that there is no reason to rearange the VERTEXes at a POINT which has only two EDGEREFS.

The sub procedure TRACE follows a string of EDGEs until it closes on itself. In effect, TRACE does a "countour plowing" operation, always following the EDGE with the same count. Whenever TRACE gets to a new complex VERTEX, it will rearange it to establish the proper path. New sheets may be generated as a result of the rearangement. Such new SHEETs are put on a "pending" list by REARANGE and will be TRACEd out later. One error message exists in TRACE. It may happen that TRACE comes to a VERTEX where a previous visit generated a pending SHEET for the path now being TRACEd. This can happen if two SHEETs touch in more than one place. If the assigned wrap count for the new SHEET fails to correspond to the wrap count of the "countour" being followed, TRACE gives the "unequal sheet count" error. This is very bad news and should never happen.

I got MATE, REARANGE and TRACE to work properly quite early on. With them alone it is possible to keep track of the relative wrap number of a number of SHEETs which touch. It is not possible to find the absolute wrap number without resort to more powerful means. Most of the "main" routine is concerned with establishing the absolute wrap number of new SHEETs.

Because the absolute wrap number for a SHEET depends on its surround, it is necessary to invoke the "WRAP" routines to get started on each disjoint set of SHEETs. Fortunately, because these SHEETs are disjoint, any VERTEX of them will be separate from the EDGEs of the rest of the figure and thus the limitation of WRAP to non-edge POINTs can be met. Unfortunately, a wrap computation for each SHEET which involves every other SHEET will be an N**2 computation. This leads us to the definitions of polygons and sheets espoused in Appendix A.

Here is how MAIN works. It is given an "input" list of SHEETs. First, it sets up a "pending" sheet list and a "completed" sheet list. It then clears out the MYSHEET pointers of all VERTEXes because they will be used as visitation markers by the TRACE routine. Next it finds the

northmost EDGE of the input SHEETs. SHEETs which have passed to the completed list are ignored, since they are no longer needed. The northmost edge is guaranteed not to have any unvisisted surrounding sheets. Actually the northmost edge with least vertical extent is found, but the least vertical extent is merely a heuristic.

Now there are two cases identified in the code as 4a and 4b. If the northmost EDGE is curved, its northmost VERTEX may be lower than its top. This is case 4b. If the northmost EDGE is straight, or if it is curved but has a top VERTEX, we examine the top VERTEX. This is case 4a.

Now if the VERTEX is northmost, we can compute its wrap number for all the completed SHEETs. This gives us the absolute wrap number just outside the collection of EDGEs we are about to examine. We now generate a fake EDGE entering the VERTEX from above. We sort this fake EDGE into the set of EDGEs around the VERTEX. The EDGE just before it in the sort sequence must be the one which enters the VERTEX from the right. If it terminates at the VERTEX, then the wrap number of the good stuff to its left is one more than the absolute wrap number just computed. But if the right EDGE begins at the VERTEX, then the wrap number to its left is the "outside" wrap number; this is the case for negative area clockwise holes in things.

Now consider the case, 4b where the top EDGE is curved. It may be a duplicate EDGE. If so, EDGEs with identical slope and curvature will exist at its terminal POINT. We can then choose the outermost of these as the really top top EDGE. Notice that if the top EDGE has positive radius it must be progressing to the left, and thus its wrap number will be one more than the outside wrap number. If it is a negative radius EDGE, then its wrap number will be the same as the outside wrap number, because it is progressing rightward.

Anyhow, out of all that we get an edge and its absolute wrap number to begin with. The rest is easy. Simply TRACE the EDGE to find the rest of the SHEET in which it belongs. TRACE will set up pending SHEETs for all of the other SHEETs that touch it. We can then TRACE all the pending sheets, and so on until the PENDING list is exhausted. We must then go back and start again on the input list by finding a northmost edge.

APPENDIX A
A BIT OF THEORY ON THE MEANING OF POLYGONS

Let us consider the work required to do the POLYGON overlap computation. There are two places where large effort is required. First, in the self intersection computation, and second in the overlap topology computation. The self intersection computation requires that one find all possible intersections of EDGEs. I have implemented this computation with a Y sort of EDGEs to limit the number of edge/edge comparisons performed. Better performance can no doubt be obtained by sorting the polygons prior to entering the self intersection computation. I have written a Warnock type polygon sorter, but it is not yet debugged nor put into service. In either case one expects N log N effort growth at best.

The topology computation is linear in the number of EDGEs provided that the EDGEs all touch each other. For each set of SHEETs that do not touch each other, the topology computation must compute a fresh "wrap number" which depends on all EDGEs previously processed, an N**2 work growth. This computation can be avoided if auxiliary information is available to guarantee that SHEETs are "unrelated", i.e. that they are disjoint and do not surround each other. This consideration leads me to the proper way to think of POLYGONs and SHEETs.

A SHEET is a closed ring of EDGEs separated by VERTEXes. A POLYGON is one or more SHEETs which are topologically related. It is essential that any SHEET which is completely surrounded by another sheet appears in the same polygon with it. It is helpful in terms of reduced computation time, but not otherwise necessary, for SHEETs which are disjoint to appear in separate POLYGONs. My programs presume that if any SHEET "A" is not actually touched by another SHEET, "B", then either A does not surround B or A and B appear in the same POLYGON.

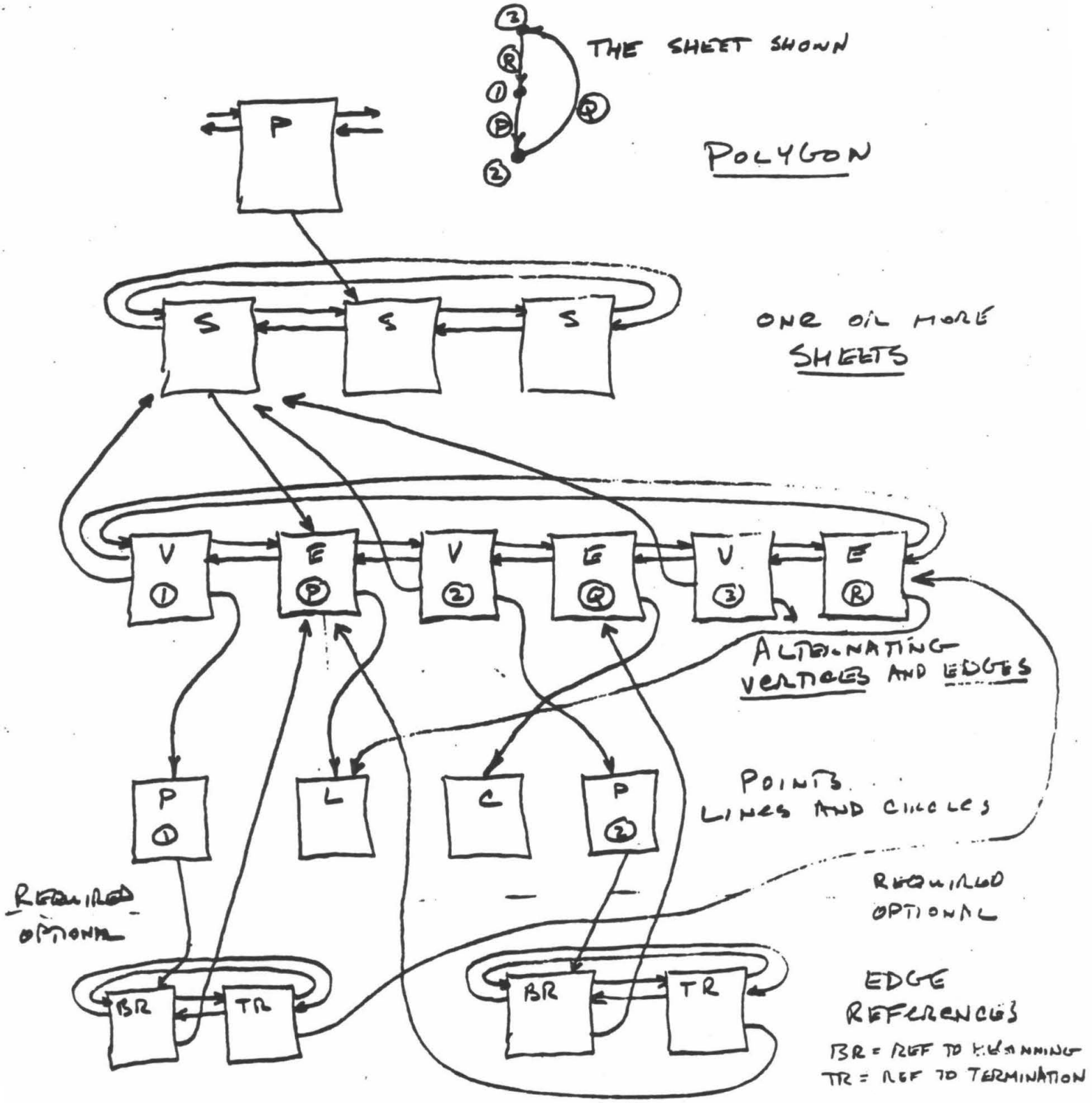


FIGURE 1. ELEMENTS OF THE DATA STRUCTURE.

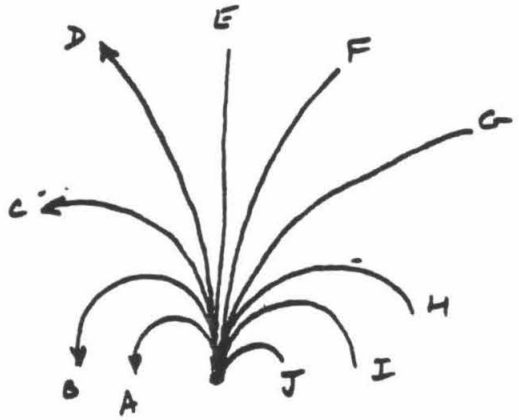


FIGURE 2. CURVATURE OF EDGES FOLLOWS A CONVENIENT ORDER. E HAS CURVATURE = 0. A, B, C, & D HAVE POSITIVE CURVATURE WITH A THE LARGEST. F G H I & J HAVE NEGATIVE CURVATURE WITH J THE MOST NEGATIVE.

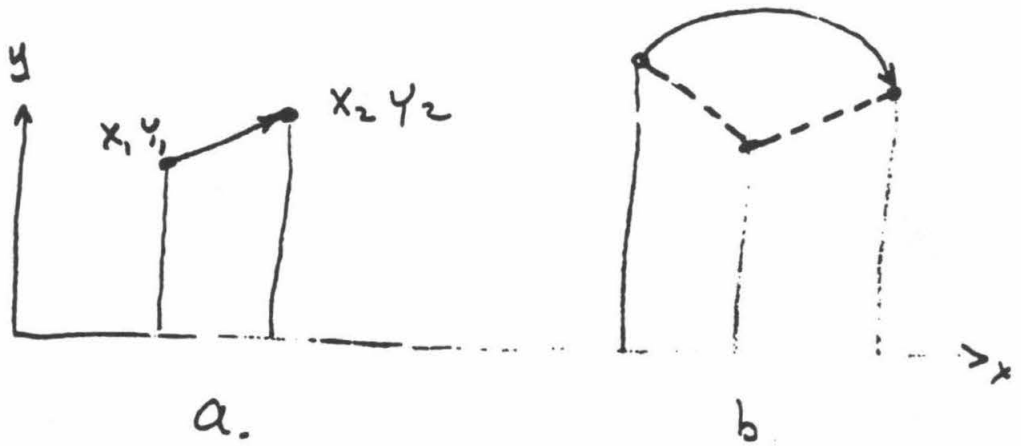


FIGURE 3. AREA COMPUTATIONS.

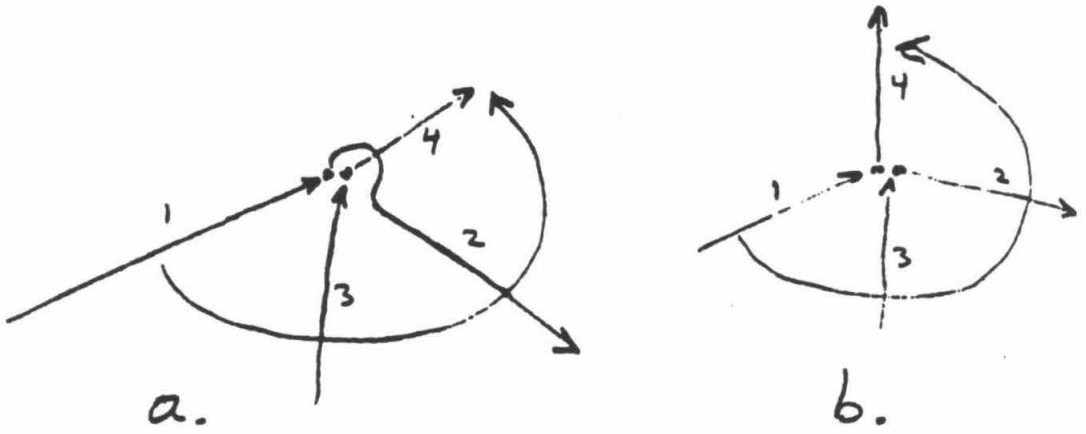


FIGURE 4. IF FOUR EDGES MEET AT A SINGLE VERTEX, MATE WILL MATCH THEM TOGETHER. HERE THE SINGLE VERTEX IS DRAWN AS TWO DOTS TO SHOW THE TOPOLOGY. INPUT a) TO MATE WILL PRODUCE OUTPUT b). WHERE EDGE 2 FOLLOWED EDGE 1 IN a), EDGE 4 FOLLOWS EDGE 1 IN b). 4 IS KNOWN TO MATE WITH 1 BECAUSE ONE EDGE (#3) ENTERS THE VERTEX AND ONE EDGE (#2) LEAVES IT AS ONE PROGRESSES COUNTERCLOCKWISE FROM 1 TO 4.

indicating the center line just as in vector mode. The END WIRE command terminates the current wire, but leaves the terminal in WIRE mode.

3.3 Read the CAP,CCP

```
C8 C7 C6 C5 C4 C3 C2 C1 C0
 0  0  1  0  1  X  X  X  X


    C4 C3 C2 C1 C0
     1  1  0  0  1   Read CAP

    C4 C3 C2 C1 C0
     1  1  0  1  0   Read CCP
```

The screen coordinates corresponding to the CAP or the CCP is returned in the following format:

byte 0	ESC	(octal 33)
byte 1	X0-X4	X0-X4 + o40
byte 2	X5-X9	X5-X9 + o40
byte 3	Y0-Y4	Y0-Y4 + o40
byte 4	Y5-Y9	Y5-Y9 + o100
byte 5	CR	(octal 15)

This format corresponds to the following sequence: <ESC><X coordinate><Y coordinate> <CR> where the X and Y coordinates are specified exactly as in Section 2.2, "X/Y Coordinate Data". This includes the addition of o40 to each byte to avoid transmitting control characters, and the inclusion of an X/Y flag bit. This flag bit is why byte 4 has o100 (o40 + o40) added in.

```
C4  C3  C2  C1  C0
 1   1   0   1   1     Read CCP on mouse button change
```

the screen coordinates corresponding to the CCP, a three bitmouse button switch code, and a 16 bit auxillary switch register code.

The following format is used:

byte 0	ESC	(octal 33)
byte 1	X4-X0	(X4-X0) + 040
byte 2	X9-X5	(X9-X5) + 040
byte 3	Y4-Y0	(Y4-Y0) + 040
byte 4	Y9-Y5	(Y9-Y5) + 0100
byte 5	S1-S0,B2-B0	(S1-S0,B2-B0) + 040
byte 6	S7-S2	(S7-S2) + 040
byte 7	S13-S8	(S13-S8) + 040
byte 8	S15-S14	(P2-P0, S15-S14) + 0100
byte 9	CR	OCTAL 15

X9-X0	X-COORDINATE OF CCP
Y9-Y0	Y-COORDINATE OF CCP
B2-B0	MOUSE KEY BUTTON CODE.
P2-P0	PREVIOUS MOUSE KEY BUTTON CODE.
S15-S0	AUXILLARY SWITCH REGISTER CODE.

Note that byte 4 has octal (0100) added as the x/y flag as noted in "Read CCP" description. Byte 8 has a similar octal (0100) addition so that a common routine may be used to assemble each of the two 4-byte fields.

3.4 Load Memory *

C8	C7	C6	C5	C4	C3	C2	C1	C0
0	0	0	0	1	1	1	0	1

All subsequent data words are loaded directly into the LSI/11 using a format similar to DEC's absolute loader. Data is interpreted in 3-byte frames, taking one 16 bit word/frame.

4. Notes on the complete protocol

The protocol I am modifying was developed as a joint project between Science Applications Inc. (SAI) and Information Technology Ltd. (ITL). The intention was to produce a protocol for graphics that is compatible with most ASCII host systems, and a number of micro-computer based terminals. All displays were assumed to be bi-level, so the major emphasis has been to provide fancy line drawings and character fonts. Further features include the ability to load and store pictures in the terminal, the ability to load and execute programs in the terminal, and methods of dealing with equipment connected to the terminal either as input

or output devices.

5. Conclusions

This proposal describes a way of getting color graphics available from the DEC-20 using the bitmap display which has been built. While there are many features which can be added, this document should describe a basic set. Please feel free to make comments and suggestions.