CHAPTER THREE

POLYGONS

INTRODUCTION

So far our discussion has dealt only with lines and characters. The world might seem rather dull if it were made out of straight lines. It would be a world of stick figures and outlines lacking texture, mass, and solidity. How much more interesting is the world of patterns, colors, and shading. Unfortunately, much of the early graphics dealt only with line drawings. This was because the available display devices (plotters, DVSTs, and vector refresh displays) were line-drawing devices. Raster displays, however, can display solid patterns and objects with no greater effort than that involved in showing their outlines. Coloring and shading are possible with raster technology. Thus with the rise in popularity of raster displays has also come an increase in attention to surface representation. The representation of surface objects is important even for line-drawing displays. In Chapter 9 we shall look at the problem of not showing lines which would normally be hidden, such as the edges on the back side of an object. For this problem, knowledge of the edges is not enough; we must also know what surfaces are present so that we can decide what can and what cannot be seen.

In this chapter we will extend our system to include a new graphic primitive, the polygon. We shall discuss what polygons are and how to represent them. We shall learn how to determine if a point is inside a polygon. Finally, a method for filling in all inside pixels will be developed.

POLYGONS

We wish to be able to represent a surface. Our basic surface primitive is a *polygon* (a many-sided figure). A polygon may be represented as a number of line segments con-

nected end to end to form a closed figure. Alternatively, it may be represented as the points where the sides of the polygon are connected. The line segments which make up the polygon boundary are called *sides* or *edges*. The endpoints of the sides are called the polygon *vertices*. The simplest polygon is the triangle, having three sides and three vertex points. (See Figure 3-1.)

We can divide polygons into two classes: *convex* and *concave*. A convex polygon is a polygon such that for any two points inside the polygon, all points on the line segment connecting them are also inside the polygon. A concave polygon is one which is not convex. A triangle is always convex. (See Figure 3-2.)

POLYGON REPRESENTATION

If we are to add polygons to our graphics system, we must first decide how to represent them. Some graphics devices supply a polygon drawing primitive to directly image polygon shapes. On such devices it is natural to save the polygon as a unit. Other devices provide a trapezoid primitive. Trapezoids are formed from two scan lines and two line segments. (See Figure 3-3.) The trapezoids are drawn by stepping down the line segments with two vector generators and, for each scan line, filling in all the pixels between them. Every polygon can be broken up into trapezoids by means similar to those described below. In such a system, it would be natural to represent a polygon as a series of trapezoids. Many other graphics devices do not provide any polygon support at all, and it is left up to the software to break up the polygon into the lines or points which can be imaged. In this case, it is again best to represent the polygon as a unit. This is the approach we shall take. We shall store full polygons in our display file and investigate the methods for imaging them using lines and points.

What should a polygon look like in our display file? We might just place line commands for each of the edges into the display file, but there are two deficiencies with this scheme. First, we have no way of knowing which line commands out of all those in the display file should be grouped together to form the polygon graphical unit; and second, we have not correctly positioned the pen for drawing the initial side. We can overcome these problems by prefacing the commands for drawing the polygon sides by a new command. This new command will tell us how many sides are in the polygon, so we will know how many of the following line commands are part of the polygon. Upon interpretation, this new command will act like a move to correctly position the pen for drawing the first side.

We have not yet used the operation codes 3 or greater. We can therefore use these codes to indicate polygons. The value of the code will indicate the number of sides. We

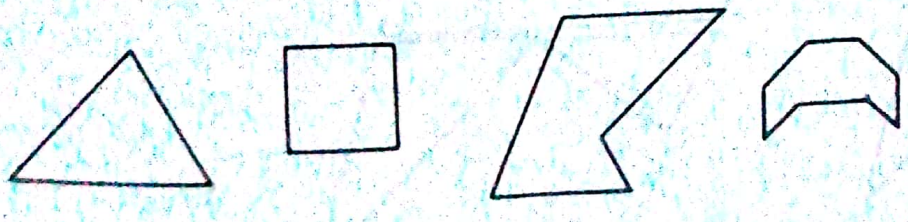


FIGURE 3-1

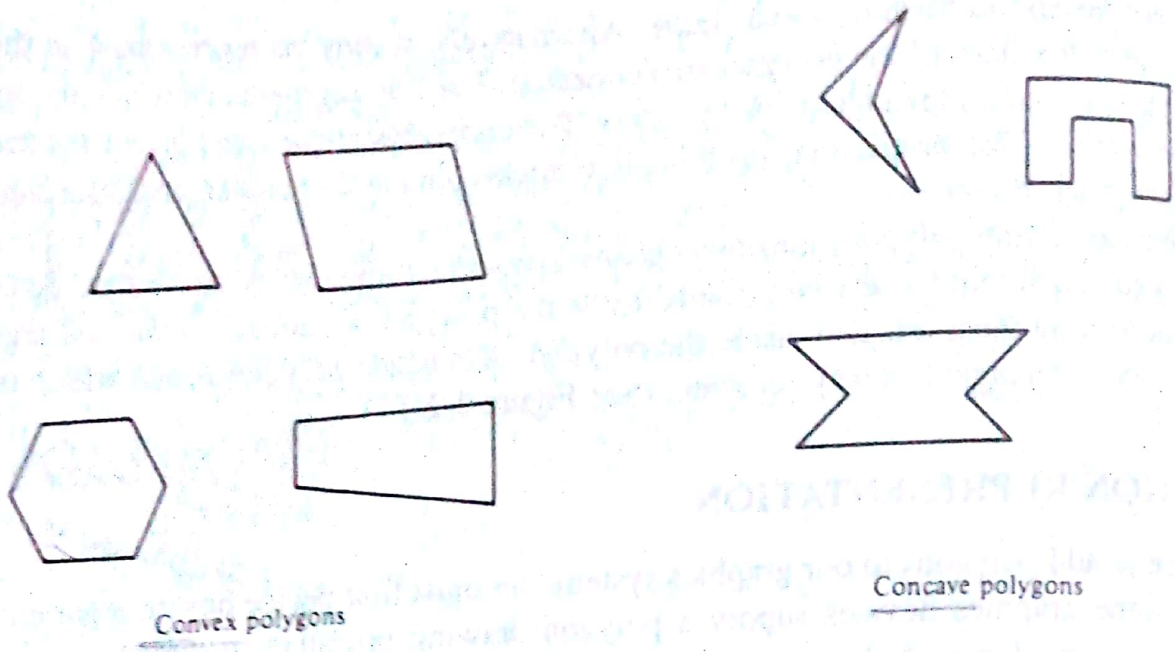


FIGURE 3-2
Convex and concave polygons.

will therefore be limited to polygons with no more sides than the maximum possible opcode. The X and Y operands of the polygon command will be the coordinates of the point where the first side to be drawn begins. Since polygons are closed curves, it will also be the final endpoint of the last side to be drawn. Upon execution, the polygon instruction (opcode 3 or greater) will signal that the following instructions belong to a polygon, but will otherwise behave as a move command. (See Figure 3-4.)

ENTERING POLYGONS

We can now consider algorithms for entering polygons into the display file. The information required to specify the polygon is the number of sides and the coordinates of the vertex points. Arrays can be used to pass the coordinates of all vertices to the

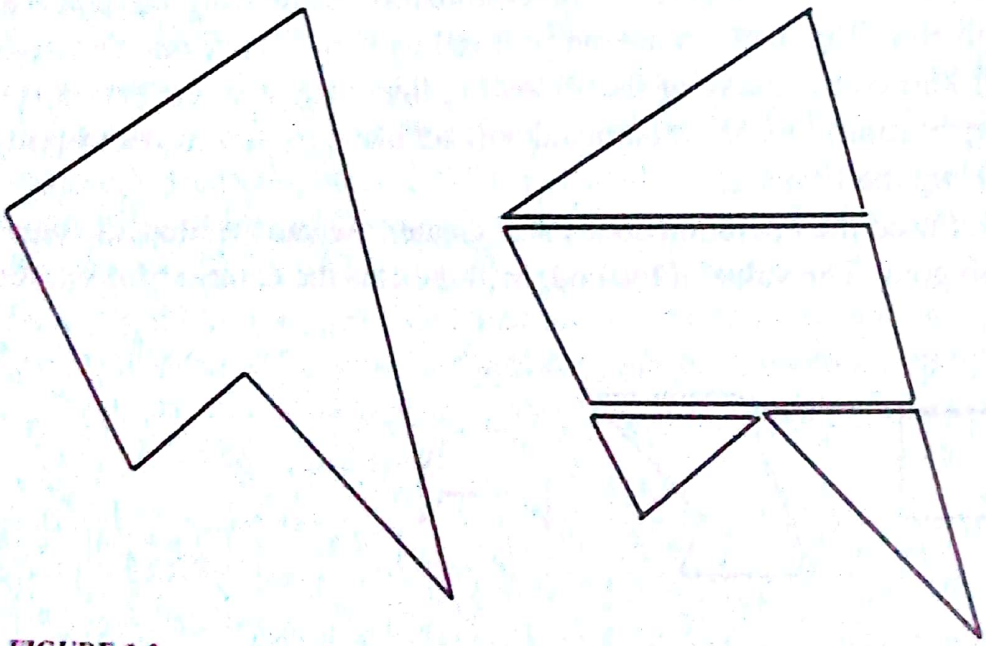
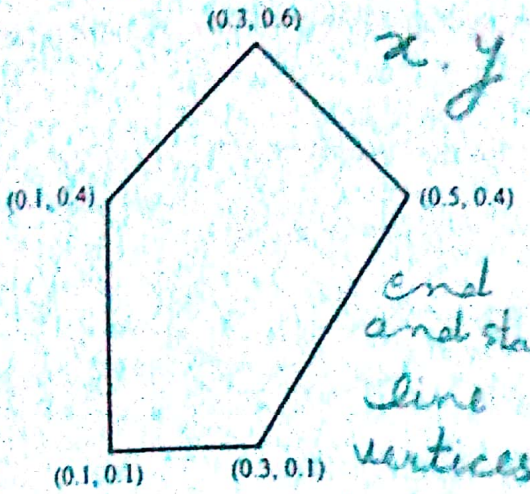


FIGURE 3-3
A polygon can be drawn as a series of trapezoids



DF-OP	DF-X	DF-Y
5	0.1	0.1
2	0.3	0.1
2	0.5	0.4
2	0.3	0.6
2	0.1	0.4
2	0.1	0.1

FIGURE 3-4
A polygon and its display-file entry.

routine. If we give absolute coordinates, the following algorithm may be employed. (See Figure 3-5.)

3.1 Algorithm POLYGON-ABS-2(AX, AY, N) Entry of an absolute polygon into the display file

Arguments	AX, AY arrays containing the vertices of the polygon
	N the number of polygon sides
Global	DF-PEN-X, DF-PEN-Y the current pen position
Local	I for stepping through the polygon sides

```
BEGIN
   IF N < 3 THEN RETURN ERROR 'POLYGON SIZE ERROR';
   enter the polygon instruction
   DF-PEN-X ← AX[N];
   DF-PEN-Y ← AY[N];
   DISPLAY-FILE-ENTER(N);
   enter the instructions for the sides
   FOR I = 1 TO N DO LINE-ABS-2(AX[I], AY[I]);
   RETURN;
END;
```

We might also wish to be able to construct polygons relative to the current pen position. We would then understand the first point specified to be a relative move from

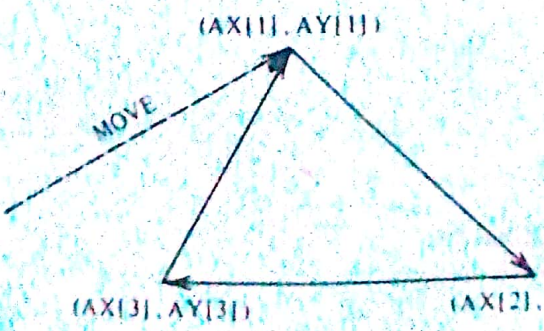


FIGURE 3-5
Absolute polygon.

the current position, and subsequent points to be used in relative line commands for the sides. (See Figure 3-6.)

3.2 Algorithm POLYGON-REL-2(AX, AY, N) Entry of a relative polygon into the display file

```
Arguments   AX, AY arrays containing the relative offsets of the vertices
            N the number of polygon sides
Global      DF-PEN-X, DF-PEN-Y the current pen position
Local       I for stepping through the polygon sides.
TMPX,TMPY Storage of the point at which the polygon is closed.
BEGIN
   IF N < 3 THEN RETURN ERROR 'POLYGON SIZE ERROR';
   DF-PEN-X ← DF-PEN-X + AX[1];
   DF-PEN-Y ← DF-PEN-Y + AY[1];
   save the starting point for closing the polygon
   TMPX ← DF-PEN-X;
   TMPY ← DF-PEN-Y;
   enter the polygon instruction
   DISPLAY-FILE-ENTER(N);
   enter the instructions for the sides
   FOR I = 2 TO N DO LINE-REL-2(AX[I], AY[I]);
   close the polygon
   LINE-ABS-2(TMPX, TMPY);
   RETURN;
END;
```

In the above algorithm, the polygon starting position must be calculated from the specified offset and the current pen position. This location must be saved temporarily so that it may be used in the final instruction, which closes the figure.

AN INSIDE TEST

Having entered the commands in the display file, we can show outlined forms for polygons by simply modifying the interpreter so that it treats command codes 3 and greater as move commands. However, we might also wish to be able to show the polygons as solid objects by setting the pixels inside the polygon as well as those on the boundary. Let us consider how we can determine whether or not a point is inside of a polygon. One method of doing this is to construct a line segment between the point in question

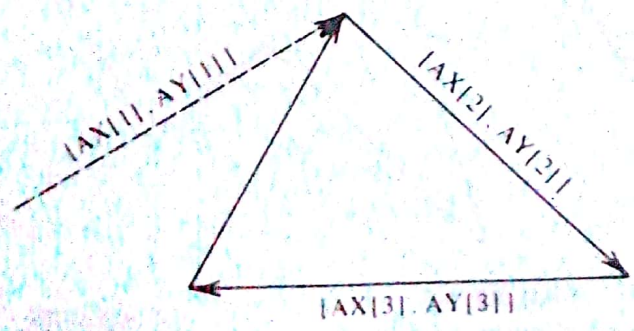
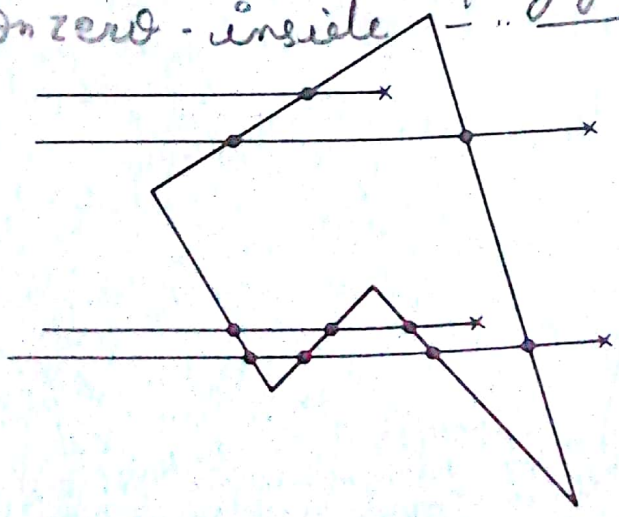


FIGURE 3-6
Relative polygon.

and a point known to be outside the polygon. It is easy to find a point outside the polygon; one could, for example, pick a point with an x coordinate smaller than the smallest x coordinate of the polygon's vertices. One then counts how many intersections of the line segment with the polygon boundary occur. If there are an odd number of intersections, then the point in question is inside; an even number indicates that it is outside. This is called the *even-odd method* of determining polygon interior points. (See Figure 3-7.)

When counting intersection points, one must be cautious when the point of intersection is also the vertex where two sides meet. To handle this case, we must look at the other endpoints of the two segments which meet at this vertex. If these points lie on the same side of the constructed line, then the point in question counts as an even number of intersections. If they lie on opposite sides of the constructed line, then the point is counted as a single intersection. (See Figure 3-8.)

There is an alternative method for defining a polygon's interior points called the *winding-number method*. Conceptually one can stretch a piece of elastic between the point in question and a point on the polygon boundary. The end attached to the polygon is slid along the boundary until it has made one complete circuit. We then examine the point in question to see how many times the elastic has wound around it. If it has wound at least once, then the point is inside. If there is no net winding, then the point is outside. Calculating the winding number for a point is not as difficult as the method just described. We begin, as in the even-odd method, by picturing a line segment running from outside the polygon to the point in question and consider the polygon sides which it crosses. However, in the winding-number method, instead of just counting the intersections, we give each boundary line crossed a *direction number*, and we sum these direction numbers. The direction number indicates the direction the polygon edge was drawn relative to the line segment we constructed for the test. For example, to test a point (x_a, y_a), let us consider a horizontal line segment $y = y_a$ which runs from outside the polygon to (x_a, y_a). We find all of the sides which cross this line segment. Now there are two ways for a side to cross. The side could be drawn starting below the line, cross it, and end above the line (first y value less than the second y value). In this case, we give a direction number of –1 to the side. Or the edge could start above the



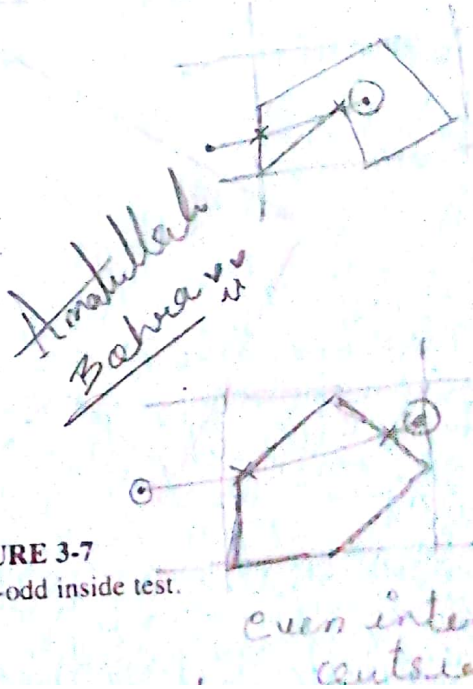


FIGURE 3-7
Even-odd inside test.

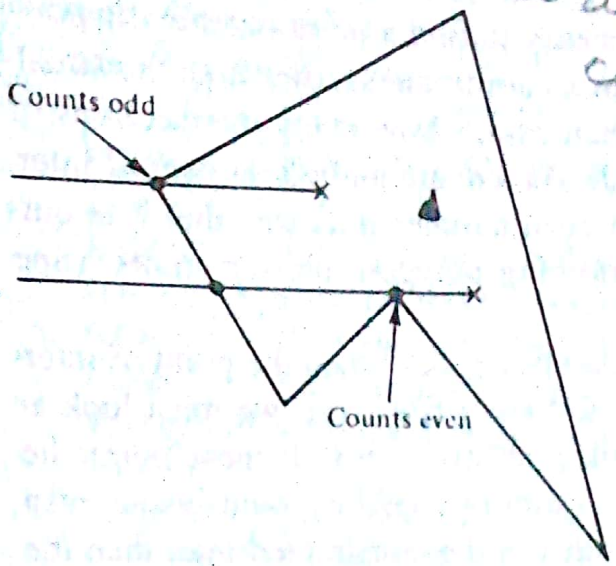


FIGURE 3-8
Count of vertex intersections.

line and finish below it (first y value greater than the second y value). This case is given a direction of 1. The sum of the direction numbers for the sides that cross the constructed horizontal line segment yields the winding number for the point in question. (See Figure 3-9.)

Using the winding number to define the interior points can yield different results from the even-odd method when a polygon is allowed to overlap itself. (See Figure 3-10.) The polygon-filling algorithm presented in this book is based on the even-odd method.

POLYGON INTERFACING ALGORITHMS

Before we become too deeply involved in the details of filling a polygon, let us give the algorithm that is needed to interface polygons with the rest of our graphics system.

We shall be able to show polygons either filled or in outline. We shall therefore provide the user with a method of indicating his preference. This is done by setting a global flag which can be checked at the time when the polygon is actually drawn.

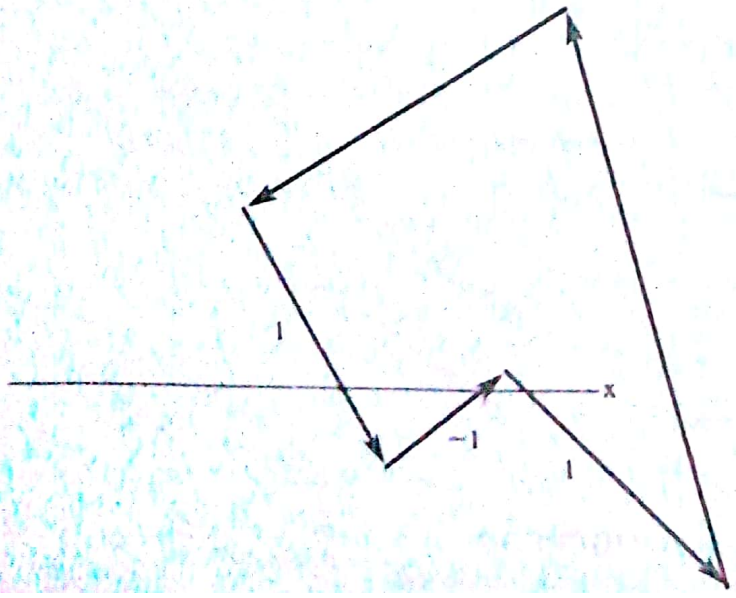
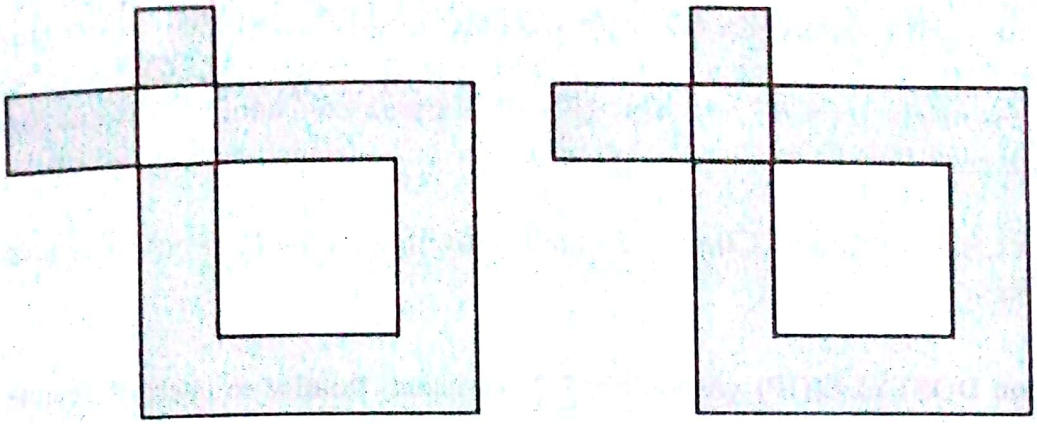


FIGURE 3-9
Calculating winding number as the sum of the direction numbers for the sides crossed



Even-odd method Winding number method

FIGURE 3-10
Interior points for a polygon that overlaps itself.

3.3 Algorithm SET-FILL(ON-OFF) User routine to set a flag indicating that polygons should be filled

```
Argument    ON-OFF the user's fill setting
Global      SOLID a flag which indicates filling of polygons
BEGIN
   SOLID ← ON-OFF;
   RETURN;
END;
```

Just as we have different line styles, we shall provide for different interior styles for polygons. Fill styles may be implemented as different colors, different shades of gray, or different filling patterns. We shall give the user a routine for selecting the desired interior style. Style values between 1 and 16, inclusive, will be acceptable. We map these values to opcodes between −16 and −31.

3.4 Algorithm SET-FILL-STYLE(STYLE) User routine to set the polygon interior style

```
Argument    STYLE the user's style request
Constant    FIRST-FILL-OP first fill op = −16
BEGIN
   DISPLAY-FILE-ENTER( FIRST-FILL-OP − (STYLE − 1));
   RETURN;
END;
```

Fill styles are handled in the same manner as line styles. A code indicating the desired style setting is placed in the display file. We can use negative integers as the operation codes for both line and fill styles. In our system, the codes 0 through −15 will be reserved for line styles and codes −16 through −31 will refer to fill styles. The user again specifies a positive integer for the interior style, which is converted to a negative number between −16 and −31 before it is stored in the display file. What the

system actually does with a polygon fill style depends upon how fill styles are implemented. The system might select a color or an intensity value (FILLCHR) to be used in filling. Or it might set a SCAN-DECREMENT parameter which indicates the filling of every nth scan line. It might also set an index into a table of patterns (FILL-PATTERN).

The DOSTYLE algorithm of Chapter 2 can be modified to include both line and interior style settings.

3.5 Algorithm DOSTYLE(OP) (Algorithm 2.21 revisited) Routine to interpret change of style commands

```
Argument   OP indicates the desired style.
Constant   FIRST-FILL-OP first fill op = -16
BEGIN
  IF OP ≤ FIRST-FILL-OP; THEN decode op and set polygon fill style
  ELSE decode op and set line style;
  RETURN;
END;
```

The INTERPRET algorithm must also be extended to handle polygon commands. When a polygon command is discovered, control is then transferred to the DOPOLYGON command, which processes the polygon.

3.6 Algorithm INTERPRET(START, COUNT) (Algorithm 2.22 revisited) Scan the display file performing the instructions

```
Arguments  START the starting index of the display-file scan
           COUNT the number of instructions to be interpreted
Local      NTH the display-file index
           OP, X, Y the display-file instruction
BEGIN
  a loop to do all desired instructions
  FOR NTH = START TO START + COUNT - 1 DO
    BEGIN
      GET-POINT(NTH, OP, X, Y);
      IF OP < -31 THEN DOCHAR(OP, X, Y)
      ELSE IF OP < 1 THEN DOSTYLE(OP)
           ELSE IF OP = 1 THEN DOMOVE(X, Y)
                ELSE IF OP = 2 THEN DOLINE(X, Y)
    END;                ELSE DOPOLYGON(OP, X, Y, NTH);
  RETURN;
END;
```

The polygon command indicates that there is a polygon and indicates how many sides it has. If the user has requested filled polygons by setting the SOLID flag, then this information is given to the FILL-POLYGON routine which will do the actual filling in. In all other respects, the polygon command is treated as a move.

3.7 **Algorithm DOPOLYGON(OP, X, Y, INDEX)** *Routine to process a polygon command*

```
Arguments   OP X Y the display-file instruction
            INDEX the position in the display file of the instruction
Global      SOLID a flag to indicate if the polygon should be filled in
BEGIN
   IF SOLID THEN FILL-POLYGON(INDEX);
   DOMOVE(X, Y);
   RETURN;
END;
```

FILLING POLYGONS

One way of filling polygons is to first draw the edges of the polygon in a blank frame buffer. Then starting with some "seed" point known to be inside the polygon, we set the intensity to the interior style and examine the neighboring pixels. We continue to set the pixel values in an increasing area until we encounter the boundary pixels. This method is called a *flood fill* because color flows from the seed pixel until reaching the polygon boundary, like water flooding the interior of a container. The flood-fill method can be quite useful in some cases. It will work with any closed shape in the frame buffer, no matter how that shape originated. However, it requires a frame buffer free of pixels with the polygon interior style, in order to avoid confusion. It also requires a seed pixel.

We could draw solid polygons by considering every pixel on the screen, applying our inside test, and setting those pixels which satisfied it. This would avoid the need for a seed pixel, but the method would be rather costly. Many pixels can be immediately eliminated by comparing them with the maximum and minimum boundary points. We really need to consider only those pixels which lie within the smallest rectangle which contains the polygon. If we first find the largest and smallest y values of the polygon, we need to consider only points which lie between them. Let us suppose that we start with the largest y value and work our way down, scanning from left to right as we go, in the manner of a raster display. Our constructed test lines will be the horizontal lines at the current y scanning value. Many problems in computer graphics can be approached a scan line at a time. Algorithms which take this approach are called *scan-line algorithms*. Often we can take advantage of what we learned in processing one scan line to make the calculation easier for the next scan line. (See Figure 3-11.)

We could draw the boundary of the polygon in a blank frame buffer and then examine the pixels in the box around the polygon, scan line by scan line. Moving across the scan line, when we encounter a pixel with the intensity of the boundary, we enter the polygon. Subsequent pixels are given the interior intensity until we encounter a second boundary pixel. The problems with this scheme are that we must start with a frame buffer free of pixels with the polygon boundary intensity and we must be careful about cases where two polygon edges meet at a single pixel. We can avoid these problems by determining the polygon boundary values directly from the polygon instruction, instead of from the frame buffer. Using the display-file instructions, we can deter-

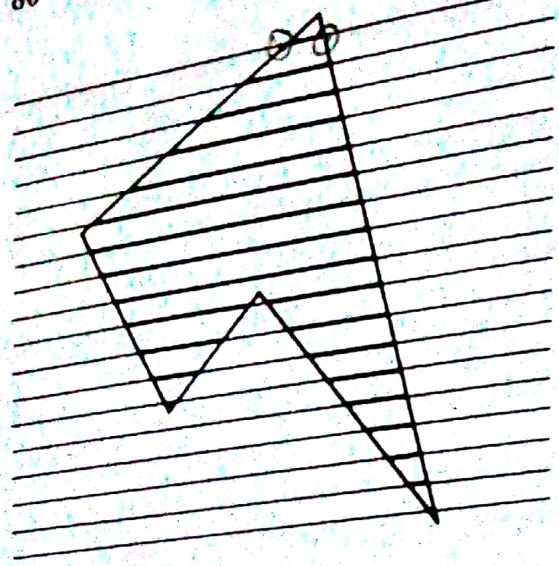


FIGURE 3-11
Filling along scan lines.

mine where the scan line crosses the polygon boundary. To test a point, we do not have to compute the intersection of our scan line with every polygon side. We need to consider only the polygon sides with endpoints straddling the test line. (See Figure 3-12.)

It will be easier to identify which polygon sides should be tested if we first sort the sides in order of their maximum y value. As we scan down the polygon, the order in which the sides must be considered will match the order in which they are stored.

Each time we step down to a lower y value, we can examine the sides being considered, in order to determine whether we have passed their lower endpoints. If we have stepped past the lowest y value of a side, it may be removed from the set of sides being considered. Now for each y value we know exactly which polygon sides can be crossed. (See Figure 3-13.)

We maintain our list of sides so that all the sides which are currently being considered will be grouped together. We shall keep two pointers to mark the boundaries of the group, START-EDGE and END-EDGE. All edges stored with list indices greater or equal to START-EDGE and less than END-EDGE should be considered. An edge in the list before START-EDGE has yet to be encountered. Those which lie behind END-EDGE will have been passed.

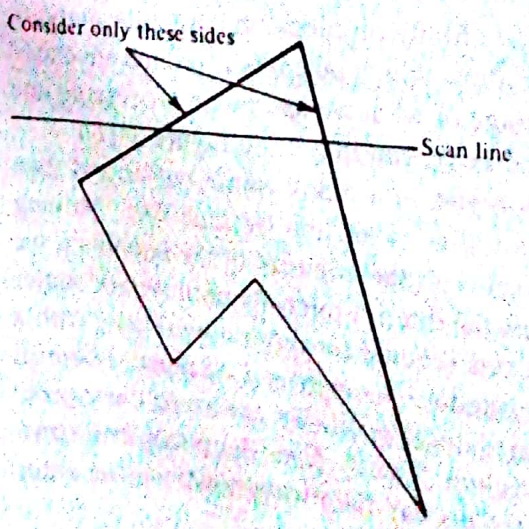


FIGURE 3-12
Consider only the sides which intersect the scan line.

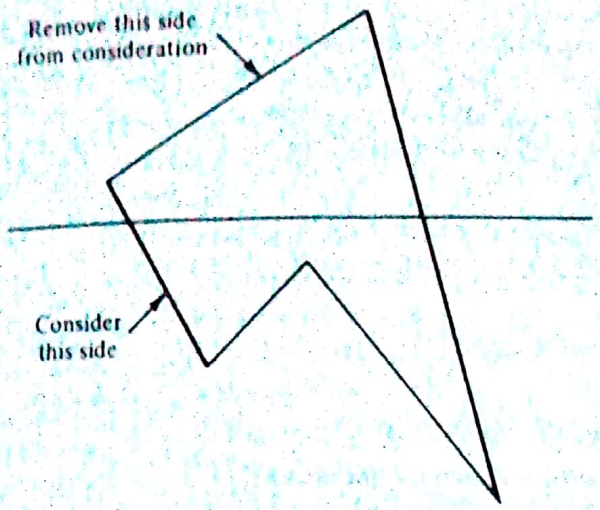


FIGURE 3-13
When the scan line passes the bottom endpoint of a side, remove it from consideration. When the scan line passes the top endpoint of a side, consider the side.

Our task becomes setting those pixels on the horizontal scan line which lie inside the polygon. It is really not necessary to examine every pixel on the horizontal line. We can think of the polygon as breaking up a horizontal scan into pieces. (See Figure 3-14.) According to the even-odd definition of the polygon interior, the pieces alternate light and dark. If we know the endpoints of the pieces—that is, the points where the scan line crosses the polygon's sides—then we can use our vector generator (or equivalent) to fill in the entire piece. We do not have to consider each pixel in the scan-line segment individually. Suppose we compute the x values for all intersections of polygon sides with a given horizontal line, and then sort these x values. The smallest x value will be the left polygon boundary. At this point the polygon begins. The next x value indicates where the polygon ends. Therefore, a line segment drawn between these values will fill in this portion of the polygon. We can pair up the sorted x values in this manner for passage to our line-drawing routine. (See Figure 3-15.)

In summary, an algorithm for filling a solid polygon should begin by ordering the polygon sides on the largest y value. It should begin with the largest y value and scan down the polygon. For each y, it should determine which sides can be intersected and find the x values of these intersection points. The x values are sorted, paired, and passed to a line-drawing routine.

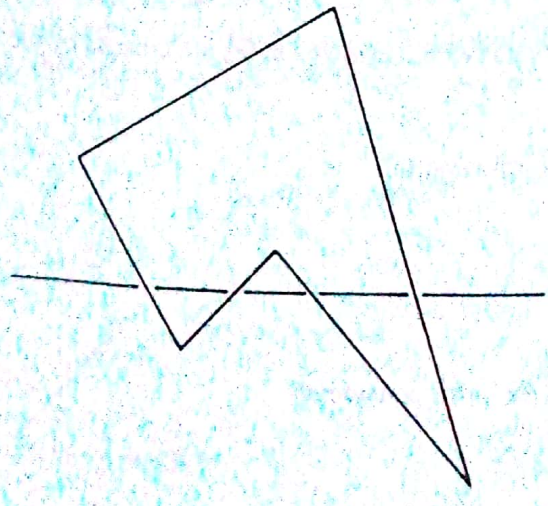


FIGURE 3-14
The polygon breaks the scan line into pieces.

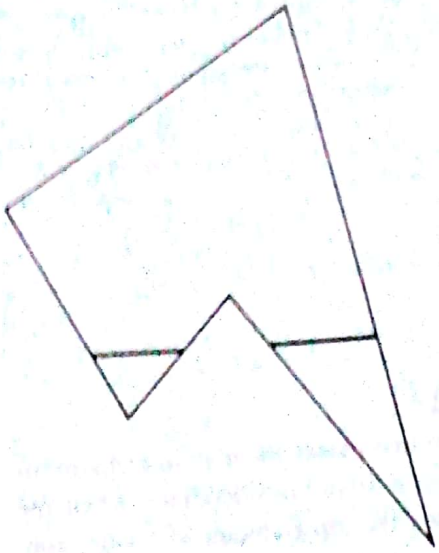


FIGURE 3-15
x values are paired and used for line drawing.

The algorithm which performs the yx scan and fills in the polygon is called FILL-POLYGON. It begins by retrieving the polygon shape information from the display file and sorting it by largest y value. This is accomplished by means of the LOAD-POLYGON algorithm. The filling in of the polygon is done by repeating five steps. The first step is to determine if any additional polygon sides should be considered for this scan line. The INCLUDE routine makes this determination. The second step is to sort the x coordinates of the points where the polygon sides cross the scan line so that they may be easily paired. This is done by the XSORT routine. The third step is to actually turn on the pixels between the polygon edges, which is done by FILL-SCAN. Next, the current scan line is decremented; and finally, the UPDATE-X-VALUES routine determines the points of intersection of the polygon with this new scan line, and removes from consideration any edges which have been passed. These steps are repeated until all polygon edges have been passed by the scanning process.

3.8 Algorithm FILL-POLYGON(INDEX) Fills in a polygon

```
Arguments   INDEX the display-file index of the instruction
Global      YMAX an array of upper y coordinates for polygon sides
            SCAN-DECREMENT the size of a scan-line decrement
Local       EDGES the number of polygon sides considered
            SCAN the y value of the scan line
            START-EDGE, END-EDGE indicate which polygon sides are crossed by the
            scan line
BEGIN
    load global arrays with the polygon vertex information
    LOAD-POLYGON(INDEX, EDGES);
    are there enough sides to consider
    IF EDGES < 2 THEN RETURN;
    set scan line
    SCAN ← YMAX[1];
    initialize starting and ending index values for sides considered
    START-EDGE ← 1;
    END-EDGE ← 1;
```

```
    fill in polygon
    pick up any new sides to be included in this scan
    INCLUDE(END-EDGE, EDGES, SCAN);
    repeat the filling until all sides have been passed
    WHILE END-EDGE ≠ START-EDGE; DO
      BEGIN
        make sure the x values are in order
        XSORT(START-EDGE, END-EDGE − 1);
        fill in the scan line
        FILL-SCAN(END-EDGE, START-EDGE, SCAN);
        next scan line
        SCAN ← SCAN − SCAN-DECREMENT;
        revise x values
        UPDATE-X-VALUES(END-EDGE − 1, START-EDGE, SCAN);
        and see if any new edges should be considered
        INCLUDE(END-EDGE, EDGES, SCAN);
      END;
    RETURN;
END;
```

Now let's consider in more detail what we would like to know about each polygon edge. We would like to know the largest and smallest y-coordinate values. The largest y value indicates at which point in the scanning process to include this edge. The smallest y value will determine when the line has been passed and need no longer be considered. We shall also need to store x-coordinate information so that we can determine where the edge will intersect the scan line. For this purpose, the x value of the endpoint with largest y value should be saved. Thus we save both x and y coordinates of the endpoint which will first be encountered in the scanning process. Now as we step down through successive scan lines, the point of intersection will shift in x. The amount by which it shifts can be determined if we know the change in x for a change in y along the edge, that is, if we know the reciprocal of the slope of the edge. These four items (X-TOP, Y-TOP, Y-BOTTOM, INVERSE-SLOPE) are all that we need to know about each side. (See Figure 3-16.)

We shall want to store this information for all nonhorizontal lines. Horizontal lines can be ignored because they are parallel to the scan lines and cannot intersect them. We shall want to store this information ordered according to Y-TOP, because this is the order in which it will be accessed. The retrieval and storing of the edge informa-

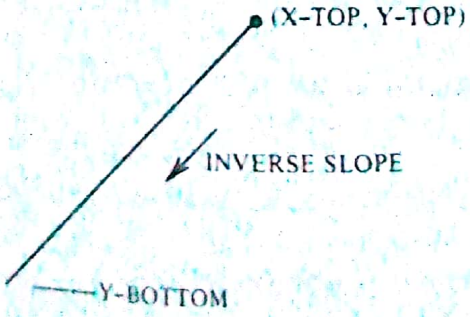


FIGURE 3-16
Parameters stored for each side of the polygon.

tion are performed by a routine which we have called LOAD-POLYGON. The algorithm begins by setting one endpoint of the edge equal to the vertex in the original polygon command. The point is converted from normalized coordinates to actual device coordinates. We add 0.5 to get the effect of rounding the point to a pixel number. The y value is rounded right away by the INT function, which is the same as FLOOR for positive arguments. The x value will be rounded at every scan line.

The LOAD-POLYGON algorithm next steps through the display file. It retrieves a new vertex by means of the GET-POINT routine. This vertex becomes the second endpoint of a polygon edge (the first endpoint came from the previous step). Horizontal edges are ignored, but information for nonhorizontal lines is saved in order by the POLY-INSERT routine. When all edges have been considered, the algorithm returns the number of edges actually saved.

3.9 Algorithm LOAD-POLYGON(I, EDGES) A routine to retrieve polygon side information from the display file. Positions are converted to actual screen coordinates

```
Arguments  I the display-file index of the instruction
           EDGES for return of the number of sides stored
Global     WIDTH-START, HEIGHT-START starting index of the screen
           WIDTH width in pixels of the screen
           HEIGHT height in pixels of the screen
Local      X1, Y1, X2, Y2 edge endpoints in actual device coordinates
           I1 for stepping through the display file
           K for stepping through the polygon sides
           DUMMY for a dummy argument
           SIDES the number of sides on the polygon
BEGIN
  set starting point for a side
  GET-POINT(I, SIDES, X1, Y1);
  X1 ← X1 * WIDTH + WIDTH-START + 0.5;
  adjust y coordinate to nearest scan line
  Y1 ← INT(Y1 * HEIGHT + HEIGHT-START + 0.5);
  get index of first side command
  I1 ← I + 1;
  initialize an index for storing side data
  EDGES ← 1;
  a loop to get information about each side
  FOR K = 1 TO SIDES DO
    BEGIN
      get next vertex
      GET-POINT(I1, DUMMY, X2, Y2);
      X2 ← X2 * WIDTH + WIDTH-START + 0.5;
      Y2 ← INT( Y2 * HEIGHT + HEIGHT-START + 0.5);
      see if horizontal line
      IF Y1 = Y2 THEN X1 ← X2
      ELSE
        BEGIN
          save data about side in order of largest y
          POLY-INSERT(EDGES, X1, Y1, X2, Y2);
```

```
            increment index for side data storage
            EDGES ← EDGES + 1;
            old point is reset
            Y1 ← Y2;
            X1 ← X2;
          END;
        I1 ← I1 + 1;
      END;
    set EDGES to be a count of the edges stored.
    EDGES ← EDGES - 1;
    RETURN;
END;
```

The POLY-INSERT algorithm is basically an insertion sort. It determines the maximum y value for the two endpoints and compares it with previously entered edges to determine where in the sequence the new edge belongs. It begins with the last element, to see if it belongs at this end. If the new edge's maximum y value is smaller than that of all previous edges, it is entered at this end. If not, the last edge is moved down one position, opening a possible storage location in the next-to-the-last place. This comparison with, and shifting of, the edges is continued until the appropriate position for the new edge is found, at which point information for the new edge is inserted. The data is stored in four separate arrays (one for each type of information). These arrays should be dimensioned to match the maximum number of sides a polygon can have in the system (one entry for each side). We have named the arrays YMAX to save Y-TOP, YMIN to save Y-BOTTOM, XA to save X-TOP, and DX to hold the change in x for each scan decrement. For a constant scan decrement, there will be a fixed change in the x-intersection value for each scan. We can find the amount that the intersection point will shift by multiplying the rate of change in x for a change in y by the scan decrement. The scan decrement is the distance between the scan lines actually being filled. Usually this will be 1, so that every scan line is filled, but by allowing other values, we can achieve different fill styles. For example, a scan decrement of 2 would fill every other scan line. (See Figure 3-17.) We use the INT function to round the x values to integer pixel positions in determining XA and DX. This is so the sides of the polygon will match the boundary drawn with the Bresenham algorithm.

3.10 Algorithm POLY-INSERT (J, X1, Y1, X2, Y2) The ordered insertion of polygon edge information

```
Arguments   J insertion index
            X1, Y1, X2, Y2 endpoints of the polygon side (y values rounded)
Global      YMAX, YMIN, XA, DX arrays for storage of polygon edge information
            SCAN-DECREMENT step between filled scan lines
Local       J1 for stepping through the stored edges
            YM the maximum y value of the new edge
BEGIN
  insertion sort into global arrays on maximum y
  J1 ← J;
  find the largest y
```

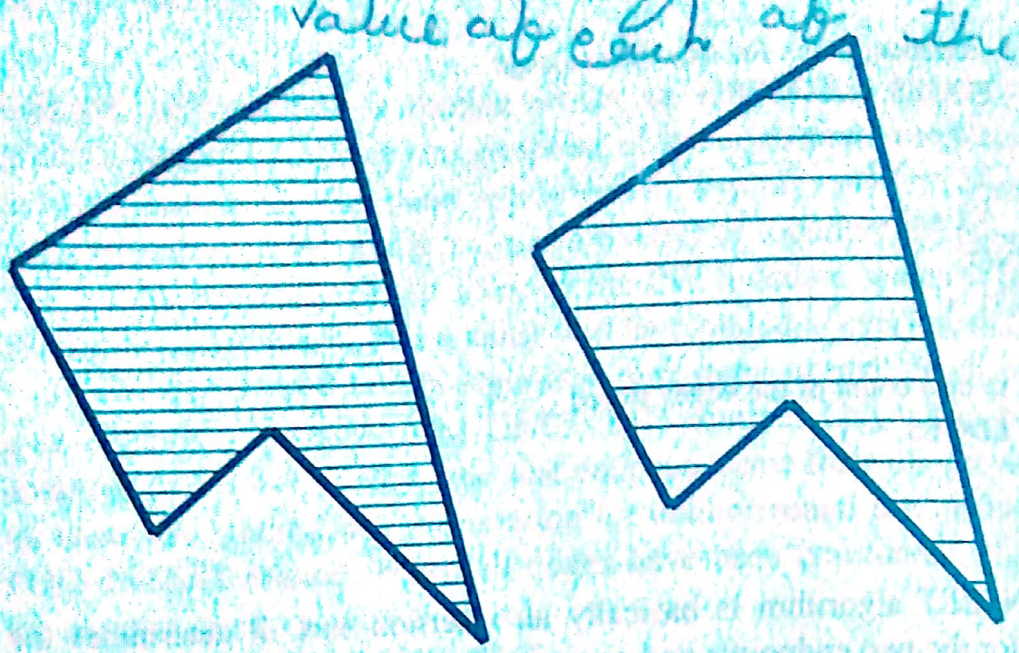


FIGURE 3-17
Using scan decrement for fill styles.

```
YM ← MAX(Y1, Y2);
find correct insertion point, moving items out of the way as we go
WHILE J1 ≠ 1 AND YMAX(J1 − 1) < YM DO
    BEGIN
        move up the insertion slot
        YMAX[J1] ← YMAX[J1 − 1];
        YMIN[J1] ← YMIN[J1 − 1];
        XA[J1] ← XA[J1 − 1];
        DX[J1] ← DX[J1 − 1];
        J1 ← J1 − 1;
    END;
insert information about side
YMAX[J1] ← YM;
DX[J1] ← ((INT(X2) − INT(X1)) / (Y2 − Y1)) * ( − SCAN-DECREMENT);
see which end is on top
IF Y1 > Y2 THEN
    BEGIN
        YMIN[J1] ← Y2;
        XA[J1] ← INT(X1);
    END
ELSE
    BEGIN
        YMIN[J1] ← Y1;
        XA[J1] ← INT(X2);
    END;
RETURN;
END;
```

The edge information for the polygon has been stored for our use, but we need not consider every edge with every scan line. We need to maintain the START-EDGE and END-EDGE pointers to delimit the edges of interest. (See Figure 3-18.)

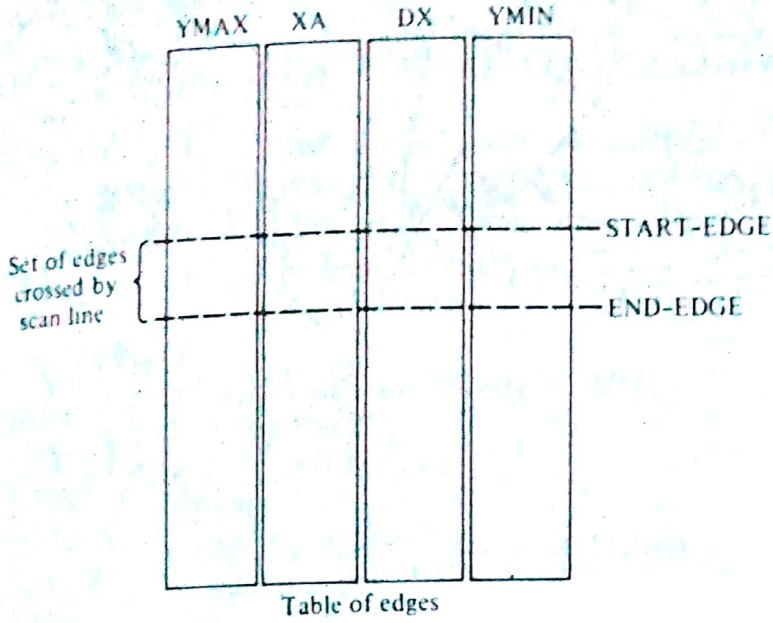


FIGURE 3-18
Table entries for edges which cross the current scan line.

The algorithm we have called INCLUDE adds new edges to the group being considered. Because of the order in which we have stored the edges, the next edge to be included will be the next edge in the array. To include the new edge, we only have to increment the END-EDGE boundary. For each new scan line, the INCLUDE algorithm checks the largest y value for the next edge; if the scan has gone below this value, then END-EDGE is incremented to include the edge.

3.11 Algorithm INCLUDE(END-EDGE, FINAL-EDGE, SCAN) Include any edges newly intersected by the scan line

```
Arguments   END-EDGE index of the side being considered for inclusion
            FINAL-EDGE index of last side
            SCAN position of current scan line
Global      YMAX, XA, DX arrays of edge information
            SCAN-DECREMENT the size of a scan-line decrement
BEGIN
  WHILE END-EDGE ≤ FINAL-EDGE AND YMAX[END-EDGE] ≥ SCAN DO
    include a new edge
    END-EDGE ← EDGE-END + 1;
  RETURN;
END;
```

The edge information between START-EDGE and END-EDGE is kept ordered on the x-intersection value. (See Figure 3-19.)

The task of maintaining this ordering belongs to the algorithm called XSORT. The XSORT routine steps through the currently active edges. If the position is correct, then nothing happens. If, however, the element is out of place, it is "bubbled up" to its correct position by a series of exchanges with its neighbors. When a new edge is entered, it may have to be shuffled down the array to place it in order. In subsequent checks, however, the edges will almost always be in their correct positions. The exception is when polygon sides are allowed to cross.

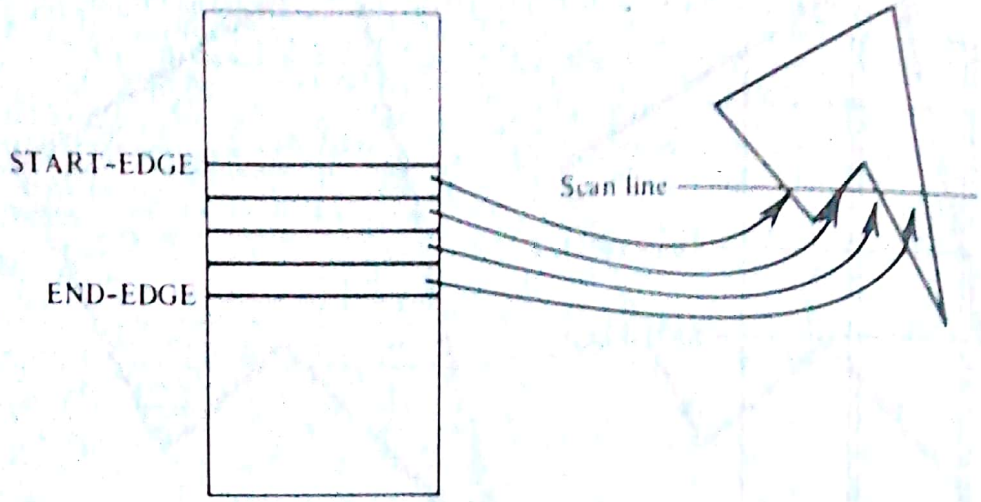


FIGURE 3-19
Table entries sorted on x.

3.12 Algorithm XSORT(START-EDGE, LAST-EDGE) Checking the order of the intersection

```
Arguments   START-EDGE index of the first of the edges considered
            LAST-EDGE index of the last edge whose order is to be checked
Global      YMIN, XA, DX arrays of edge information
Local       K, L for stepping through the edges
            T temporary storage for the exchange
BEGIN
  FOR K = START-EDGE TO LAST-EDGE DO
    BEGIN
      L ← K;
      WHILE L > START-EDGE AND XA[L] < XA[L − 1] DO
        BEGIN
          T ← YMIN[L];
          YMIN[L] ← YMIN[L − 1];
          YMIN[L − 1] ← T;
          T ← XA[L];
          XA[L] ← XA[L − 1];
          XA[L − 1] ← T;
          T ← DX[L];
          DX[L] ← DX[L − 1];
          DX[L − 1] ← T;
          L ← L − 1;
        END;
    END;
  RETURN;
END:
```

The next algorithm, FILL-SCAN, actually fills in a scan line. It contains a loop which steps through all current intersection points, connecting pairs with line segments. The actual line-segment drawing is done by FILLIN.

3.13 Algorithm FILL-SCAN(END-EDGE, START-EDGE, SCAN) Fill in the scan line

Argument START-EDGE, END-EDGE indicates which edges are crossed by the scan line
SCAN the position of the scan line
Global XA an array of edge intersection positions
Local NX the number of line segments to be drawn
J for stepping through the edges
K for stepping through line segments

```
BEGIN
  NX ← (END-EDGE − START-EDGE) / 2;
  J ← START-EDGE;
  FOR K = 1 TO NX DO
    BEGIN
      FILLIN(XA[J], XA[J + 1], SCAN);
      J ← J + 2;
    END;
  RETURN;
END;
```

The FILLIN routine may depend upon the type of display system being used. If FILLIN uses some vector-generating routine provided by the display, FILLIN may have to store the current line style, set the line style to the polygon interior style, move to one of the endpoints, draw a line to the other endpoint, and reset the line style and pen position to their original values. Alternatively, we can write a simple FILLIN algorithm based on an assumed frame buffer (or some other method of setting individual pixels), as were the vector generation algorithms of Chapter 1.

3.14 Algorithm FILLIN(X1, X2, Y) Fills in scan line Y from X1 to X2

Arguments X1, X2 end positions of the scan line to be filled
Y the scan line to be filled
Global FILLCHR intensity value to be used for the polygon
FRAME the two-dimensional frame buffer array
Local X for stepping across the scan line

```
BEGIN
  IF X1 = X2 THEN RETURN;
  FOR X = X1 TO X2 DO FRAME[X, Y] ← FILLCHR;
  RETURN;
END;
```

For each new scan line, we must examine the currently active edges to see if any have been passed. If the edge should still be considered, the intersection value for the new scan line should be calculated; if it has been passed, the edge should be removed from consideration. This is the job of the algorithm called UPDATE-X-VALUES. To determine if an edge still crosses the scan line, the lowest y value is examined. To up-

date the intersection point, the constant step size (determined in INCLUDE) is added. To remove an edge from consideration, the information for previous edges is moved up one position in the arrays. This overwrites the deleted edge and allows incrementing of the START-EDGE boundary. (See Figure 3-20.)

3.15 Algorithm UPDATE-X-VALUES(LAST-EDGE, START-EDGE, SCAN) Update points of intersection between edges and the scan line

```
Arguments   START-EDGE and LAST-EDGE limits of current edge list
            SCAN the current scan line
Global      XA, DX, YMIN arrays of edge information
Local       K1 index of edge being considered for update
            K2 index of where to store the updated edge

BEGIN
  K2 ← LAST-EDGE;
  FOR K1 = LAST-EDGE TO START-EDGE DO
    BEGIN
      check each edge
      IF YMIN[K1] < SCAN THEN
        BEGIN
          the edge is still active so update its x values
          XA[K2] ← XA[K1] + DX[K1];
          IF K1 ≠ K2 THEN
            BEGIN
              YMIN[K2] ← YMIN[K1];
              DX[K2] ← DX[K1];
            END;
          decrement K2 so the edge won't get overwritten
          K2 ← K2 − 1;
        END;
    END;
  START-EDGE ← K2 + 1;
  RETURN;
END;
```

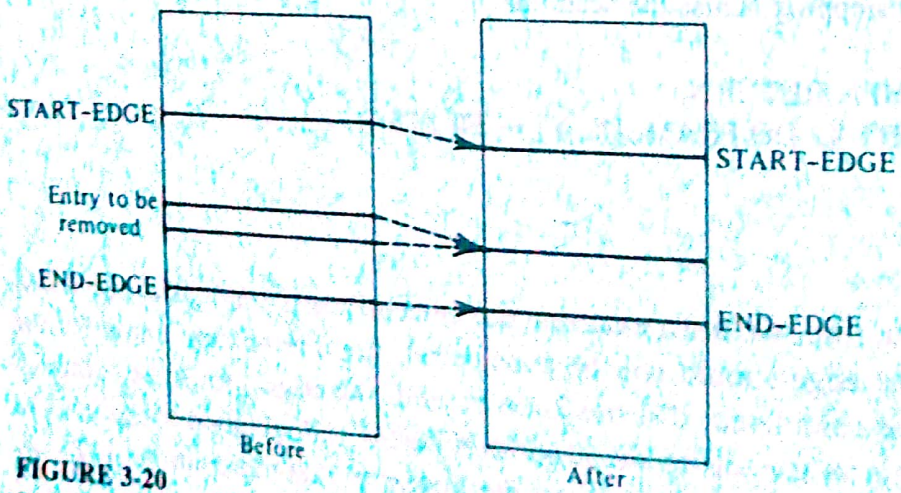


FIGURE 3-20
Removal of a table entry.

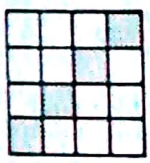
The above algorithms allow us to fill in the interiors of polygons efficiently. We should note, however, that if we are not careful, the method used for determining the polygon boundary may be different from that used by the line generation algorithm to outline the polygon. The interior may turn out to be a pixel wider than the edge, which can be apparent at low resolution.

FILLING WITH A PATTERN

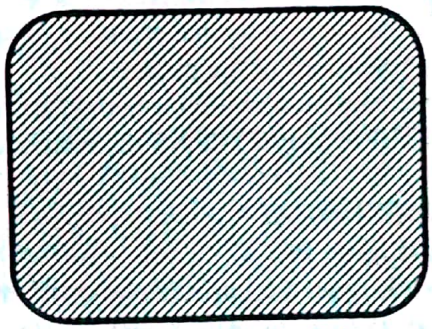
We mentioned that polygon fill styles might be patterns. Patterns are most easily implemented on raster display devices. A pattern is a grid of pixel values which is replicated like tiles to cover the polygon area. A pattern is often fixed, or *registered*, to the imaging surface so that if two polygons are filled with the pattern and placed side by side, the pattern will match at the boundary. We can imagine taking the pattern and replicating it to cover the entire imaging surface, and then erasing it anywhere outside of the polygons which use it. (See Figure 3-21.)

Assuming a frame buffer (or some other means of setting individual points), we can show how patterns might be added to our graphics system. We can set up a table of patterns and use the fill style to select one. We shall provide a routine for placing patterns into the table. We shall also give a version of FILLIN which uses the table. Some examples of 4 × 4 patterns are shown in Figure 3-22.

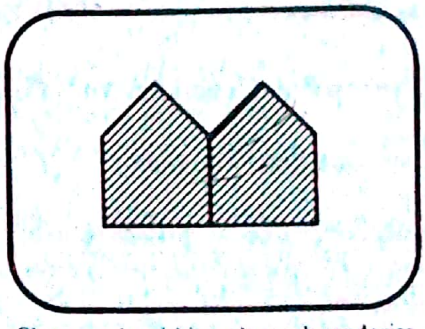
The pattern table can be composed of three arrays—PATTERN-X, PATTERN-Y, and PATTERNS—where PATTERN-X and PATTERN-Y are arrays of numbers which



A pattern



Replicated across the display



Shown only within polygon boundaries

FIGURE 3-21
Replication of a pattern.

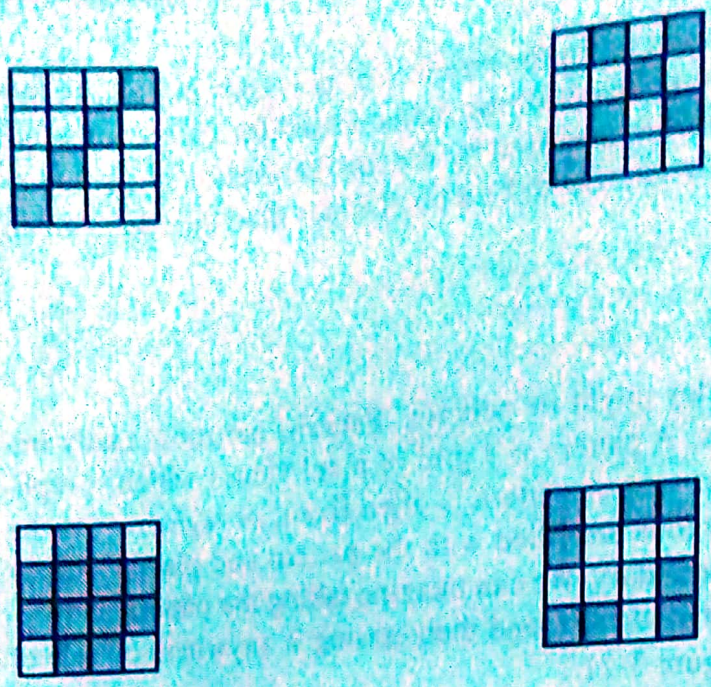


FIGURE 3-22
Some 4 × 4 patterns.

specify the size of each pattern, and PATTERNS is an array of two-dimensional arrays which are the actual patterns. (One way to implement PATTERNS is to make it a three-dimensional array, but this may not be as flexible as alternatives offered by some languages.) We place an individual pattern into the pattern table with the following algorithm.

3.16 Algorithm SET-PATTERN-REPRESENTATION(PATTERN-INDEX, PAT-X, PAT-Y, NEW-PATTERN) Enters a pattern into the pattern table

Arguments PATTERN-INDEX the place in the pattern table to save the pattern
PAT-X, PAT-Y the dimensions of the pattern
NEW-PATTERN the pattern, a two-dimensional array of intensity values

Global PATTERN-X, PATTERN-Y, PATTERNS the pattern table

```
BEGIN
    PATTERN-X[PATTERN-INDEX] ← PAT-X;
    PATTERN-Y[PATTERN-INDEX] ← PAT-Y;
    the following statement depends upon the implementation of PATTERNS, and may
    entail the copying of all the individual intensity values
    PATTERNS[PATTERN-INDEX] ← NEW-PATTERN;
    RETURN;
END;
```

The FILLIN algorithm is revised as follows:

3.17 Algorithm FILLIN(X1, X2, Y) (Revision of algorithm 3.14) Fills in scan line Y from X1 to X2

Arguments X1, X2 end positions of the scan line to be filled
Y the scan line to be filled

Global FILL-PATTERN pattern table index of the pattern to use
PATTERN-X, PATTERN-Y, PATTERNS the pattern table
FRAME the two-dimensional frame buffer array

```
Local       X for stepping across the scan line
            PX, PY for accessing the pattern
            PATTERN-TO-USE the pattern to be used in filling
            PAT-X, PAT-Y the x and y dimensions of the pattern
BEGIN
   IF X1 = X2 THEN RETURN;
   PAT-X ← PATTERN-X[FILL-PATTERN];
   PAT-Y ← PATTERN-Y[FILL-PATTERN];
   if a local array is used as shown in the following statement,
   then the implementation should avoid copying of individual elements
   PATTERN-TO-USE ← PATTERNS[FILL-PATTERN];
   PX ← MOD(X1, PAT-X) + 1;
   PY ← MOD(Y, PAT-Y) + 1;
   FOR X = X1 TO X2 DO
      BEGIN
         FRAME[X, Y] ← PATTERN-TO-USE[PX, PY];
         IF PX = PAT-X THEN PX ← 1
         ELSE PX ← PX + 1;
      END;
   RETURN;
END;
```

The MOD functions give the remainder after division of the first element by the second. This is what causes the replication of the pattern. If we have a value X1 or Y which is larger than the pattern size, the MOD function wraps PX and PY back into values within the pattern, thereby repeating the pattern. The 1s are added into this calculation because we have assumed our arrays are dimensioned with starting index 1. We set the local variables PAT-X, PAT-Y, and PATTERN-TO-USE in order to remove the address computation from the loop. Actually, it would be even better to make these variables global and to set them in DOSTYLE (the only reason we did not do so is that DOSTYLE is too system-dependent for a clean example).

On many devices there are only two pixel states, on or off. For these displays, the frame buffers and pattern tables can be compactly implemented by using individual bits to describe the pixel states. Pattern dimensions can be chosen to lie on word boundaries, and the FILLIN algorithm can be made more efficient by dealing with entire words of pixel values.

INITIALIZATION

To finish this chapter's algorithms, an initialization routine is needed to set default values for filling and fill-style parameters. If filling with patterns is possible, then the pattern table should be initialized to a default set of patterns.

3.18 Algorithm INITIALIZE-3 Routine to initialize the system

```
Local       P for stepping through the pattern table
Constants   MINIMUM-FILL-OP opcode for first fill style = -16
            NUMBER-OF-PATTERNS the size of the pattern table
```

```
                DEFAULT-PAT-X, DEFAULT-PAT-Y size of the default pattern
                DEFAULT-PATTERN a default pattern array

BEGIN
  INITIALIZE-2;
  DOSTYLE(MINIMUM-FILL-OP);
  SET-FILL(FALSE);
  the following loop is included only if polygons may be filled with patterns
  FOR P = 1 TO NUMBER-OF-PATTERNS DO
    SET-PATTERN-REPRESENTATION(P, DEFAULT-PAT-X, DEFAULT-PAT-Y,
    DEFAULT-PATTERN);
  RETURN;
END;
```

ANTIALIASING

In Chapter 1 we mentioned a technique for smoothing the jagged steps in a line which is introduced by the quantization to finite-sized pixels. Aliasing is a problem for the edges of polygons just as it is for lines, and several antialiasing techniques which use the shading of gray-level displays to reduce the effects have been developed. One technique is to calculate what fraction of a pixel is actually covered by the polygon and how much is background. The pixel intensity displayed would then be the average of the background and polygon intensities weighted by their relative areas.

Another approach is to generate the scene at a higher resolution than that which will actually be used, and then to average the intensity values of neighboring pixels to determine the intensity to be displayed. Increasing the resolution between four and eight times gives good results. Note that this need not be as much work as it seems. The antialiasing can be incorporated as part of the polygon-filling algorithm. Antialiasing need only be applied to the points on the edge (interior points will have the full polygon intensity). It can be carried out scan line by scan line, so we don't need excessive amounts of memory.

AN APPLICATION

Before leaving this chapter, let's discuss how we might use some of the routines which we have developed. Let's consider how we might write a package to draw a bar graph of some data. We have already seen in Chapter 2 how the axes might be drawn and labeled, so we shall concentrate on how to generate the bars. We can make a bar out of a four-sided polygon. (See Figure 3-23.)

Suppose that we construct such a polygon with lower-left corner at the current pen position by using the POLYGON-REL-2(AX, AY, 4) command. Then the necessary AX and AY array values would be

$$AX = 0, DX, 0, -DX$$
$$AY = 0, 0, DY, 0$$

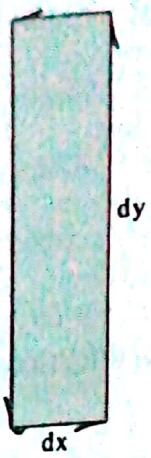


FIGURE 3-23
Constructing a solid bar from a polygon.

Here, DX will give the width of the polygon and DY will give its height. Now we can fix DX at the width which we want our bar graph's bars to have. All bars should have the same width, but the height of each bar depends upon the input data. Each bar might have a different height. We can see, however, that it is easy to change the bar height; we need to change only the value of the DY parameter in the AY array. Thus by poking the appropriate value into the AY array and calling the POLYGON-REL-2 routine, we can generate a bar with any height we choose. Now what we want to do is to draw a series of bars, each at a different horizontal position and each with a height representing the data value for that position. This is easy because we have used a relative polygon command. To position the bar, we need only move the pen to the correct starting position before drawing the polygon. (See Figure 3-24.)

```
MOVE-ABS-2(X, Y);
POLYGON-REL-2(AX, AY, 4);
```

Now we can place these instructions within a loop. Each time through the loop, we get one of the data values to be plotted. We increment X in order to position the pen at the starting point for the next bar. We calculate the height of the bar from the data value and put it into the AY array. And we call the MOVE and POLYGON algorithms to actually draw the bar. Each time through the loop, another bar would be drawn until the graph is complete. (See Figure 3-25.)

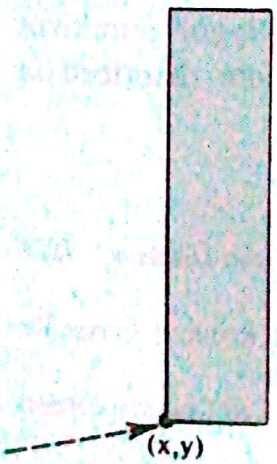


FIGURE 3-24
A MOVE command will position a relative polygon.

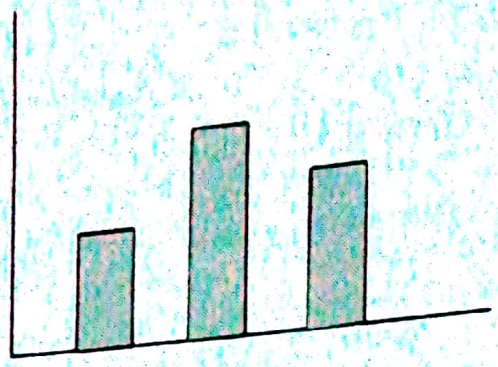


FIGURE 3-25
A bar graph constructed by repeated use of the POLYGON command.

We might note that if the bars are equally spaced and drawn sequentially, the explicit MOVE-ABS-2 command can be replaced by the implicit relative move done by the POLYGON-REL-2 algorithm.

FURTHER READING

A discussion of winding numbers and inside tests may be found in [NEW80]. We represented polygons as a list of vertex coordinates; alternative representations are presented in [BUR77] and [FRA83]. To fill a polygon, we broke it down into individual scan lines. This technique is called scan conversion and is discussed in [BAR73] and [BAR74]. An alternative approach to filling regions such as polygons is to draw their border in the frame buffer and then, starting with a seed pixel within the polygon, progressively examine each pixel, changing its color to the interior style until the boundary is encountered. This technique is called flood fill or seed fill and is discussed in [LIE78]. Instead of filling from a single seed, we can scan the area to be filled, building regions. When two regions merge, we note their equivalence; then on a second pass we shade all regions associated with the interior [DIS82]. The decomposition of polygons into trapezoids or triangles is described in [FOU84], [GAR78], [JAC80], and [LIT79]. Polygons can also be described as the differences of convex polygons [TOR84]. An attempt to save the scan-converted polygons in the display file rather than in a frame buffer is described in [SPR75]. Filling polygons on calligraphic devices with patterns of lines is discussed in [BRA79]. Polygon filling is also discussed in [AGK81], [DUN83], [PAV78], [PAV81], [POL86], and [SHA80]. A discussion of antialiasing of polygons is given in [BAR79] and [CRO81]. Our polygon primitives are based on proposed extensions to the CORE system, which are described in [GSPC79].

[AGK81] Agkland, B. D., "The Edge Flag Algorithm—A Fill Method for Raster Scan Displays," *IEEE Transactions on Computers*, vol. C-30, no. 1, pp. 41–47 (1981).

[BAR73] Barrett, R. C., and Jordan, B. W., Jr., "A Scan Conversion Algorithm with Reduced Storage Requirements," *Communications of the ACM*, vol 16, no. 11, pp. 676–682 (1973).

[BAR74] Barrett, R. C., and Jordan, B. W., Jr., "Scan-Conversion Algorithms for a Cell Organized Raster Display," *Communications of the ACM*, vol. 17, no. 3, pp. 157–163 (1974).

[BAR79] Barros, J., Fuchs, H., "Generating Smooth 2-D Monocolor Line Drawings on Video Displays," *Computer Graphics*, vol. 13, no. 2, pp. 260–269 (1979).