CHAPTER NINE

HIDDEN SURFACES AND LINES

INTRODUCTION

In the last chapter we described how to obtain different views of a scene. We developed perspective projections which made constructed objects look more realistic. But for a realistic scene, we should only draw those lines and polygons which could actually be seen, not those which would be hidden by other objects. What is hidden and what is visible depend upon the point of view. As seen from the front, the front of a building is visible, while the back of the structure is hidden; but as seen from the rear, this situation is reversed. We cannot see the contents of the building from outside because they are hidden by the building's walls, but from a point of view inside the building, some of the contents should be displayed. So far, we have learned how to model and project three-dimensional objects, but all parts of the objects are always displayed. This gives our drawings a transparent quality. Such figures are called *wire-frame drawings* because they look as if they are wire outlines of the supposedly solid objects. Complex objects can easily turn into a confusing clutter of line segments. It may be difficult to judge which lines belong to the front of the object and which to the back. The removal of hidden portions of objects is essential to producing realistic-looking images. (See for example Plates 4 through 15.) In this chapter we consider the problem of removing those lines which would normally be hidden by part of the object. If we can assign this task to the machine, then the user will be free to construct the entire model (front, back, inside, and outside) and still be able to see it as it will actually appear. This problem is not nearly as easy as it might seem at first. What nature does with ease (and a lot of parallel processing) we must do through extensive computation. There exist many solutions to the hidden surface and line problem. Many approaches have been tried; perhaps the best solution has yet to be found. We shall discuss several approaches, but only one solution will be implemented. The first section considers *back-face detection* and removal. This is sufficient for single convex objects. The second section considers the removal of hidden surfaces by means of the *painter's algorithm*.

BACK-FACE REMOVAL Imp.

Hidden-line removal can be a costly process, and so it behooves us to apply easy tests to simplify the problem as much as possible before undertaking a thorough analysis. There is a simple test which we can perform which will eliminate most of the faces which cannot be seen. This test identifies surfaces which face away from the viewer. They are the surfaces which make up the back of the object. They cannot be visible because the bulk of the object is in the way. This does not completely solve the hidden-surface problem because we can still have the front face of an object obscured by a second object or by another part of itself. But the test can remove roughly half of the surfaces from consideration and thus simplify the problem.

We begin our discussion by noting that we shall only consider polygons. Lines cannot obscure anything, and although they might be obscured, they are usually found only as edges of surfaces on an object. Because of this, polygons suffice for most drawings. Now a polygon has two surfaces, a front and a back, just as a piece of paper does. We might picture our polygons with one side painted light and the other painted dark. But given the arrays of vertex coordinates which represent polygons, how can we tell which face is which? Easy! We shall say that when we are looking at the light surface, the polygon will appear to be drawn with counterclockwise pen motions. If we move our point of view to the other side, so that the dark surface may be seen, the pen drawing the polygon will appear to move clockwise. (See Figure 9-1.)

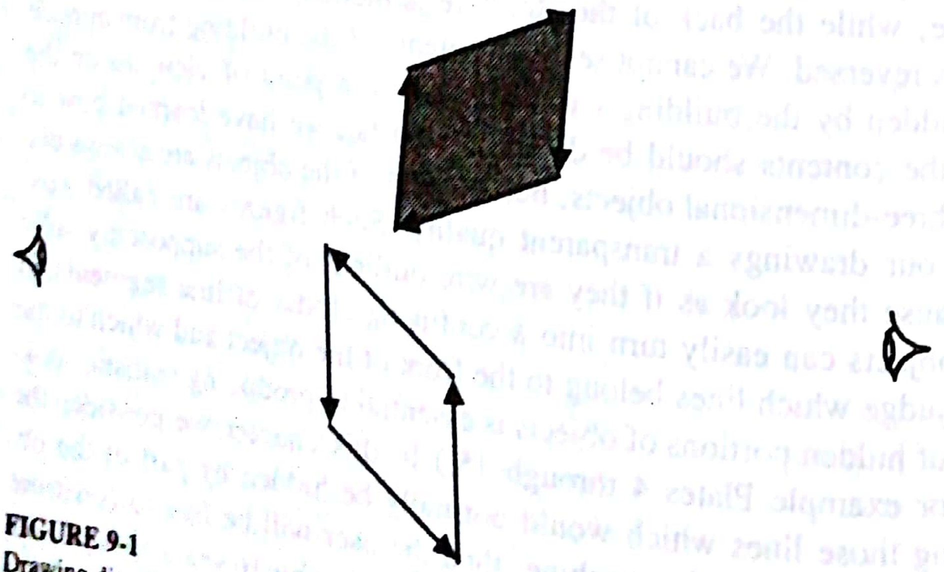

FIGURE 9-1
Drawing direction changes with the side viewed.

Now a little mathematics will give us the direction that the polygon surface is facing (the normal to the plane of the polygon). We shall describe two methods of finding the vector normal to the plane of the polygon. The first method uses an operation that is called the *vector cross product*. The cross product of two vectors is a third vector with the length equal to the product of the lengths of the two vectors times the sine of the angle between them and, most important to us, a direction perpendicular to the plane containing the two vectors. (See Figure 9-2.)

The formulas for a three-dimensional cross product in a right-handed coordinate system are as follows. For

$$[R_x \quad R_y \quad R_z] = [P_x \quad P_y \quad P_z] \times [Q_x \quad Q_y \quad Q_z] \tag{9.1}$$

we have

$$\begin{aligned} R_x &= P_yQ_z - P_zQ_y \\ R_y &= P_zQ_x - P_xQ_z \\ R_z &= P_xQ_y - P_yQ_x \end{aligned} \tag{9.2}$$

Now two sides of a polygon describe two vectors in the plane of the polygon, so the cross product of two polygon sides forms a vector pointing out from the polygon face. Will this vector point out from the dark face or the light face? That depends on whether the two sides form a convex or concave angle.

Let's assume that we are dealing with two adjacent sides which do not lie in the same line and which meet in a convex vertex, that is, a vertex where two sides meet to form a convex angle. The vector cross product will yield a vector which points out of the light face. (See Figure 9-3.)

The problem with this first approach to finding the vector normal to the polygon is that some searching and checking are required to find a vertex at a convex corner of noncollinear sides. The second method (which we shall use) is suggested by Newell and described in [SUT74]. The calculation is as follows: If the n vertices of the polygon are (x_i, y_i, z_i), then form the sums over all vertices

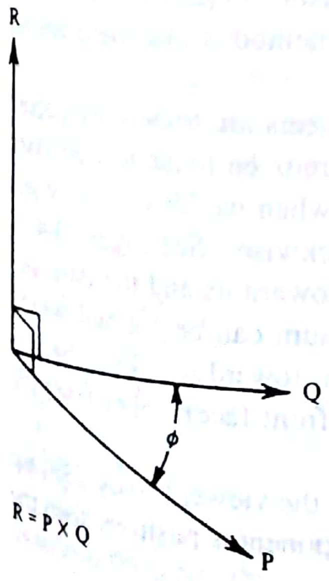

FIGURE 9-2
The vector cross product.

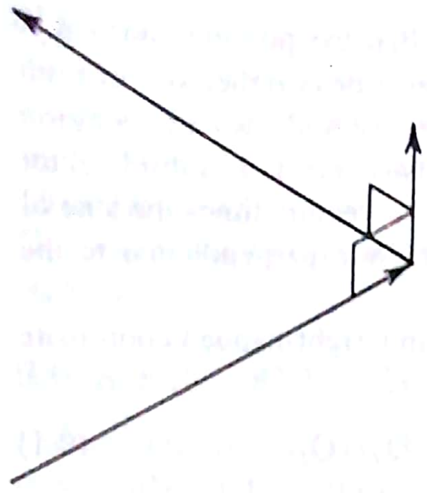

FIGURE 9-3
The cross product point out of the light face at a convex vertex.

$$a = \sum_{i=1}^{n} (y_i - y_j)(z_i + z_j)$$

$$b = \sum_{i=1}^{n} (z_i - z_j)(x_i + x_j) \qquad (9.3)$$

$$c = \sum_{i=1}^{n} (x_i - x_j)(y_i + y_j)$$

where if i = n, then j = 1; otherwise, j = i + 1.

The result [a b c] is a vector normal to the polygon. Each of these sums gives twice the area of the projections of the polygon on a plane. That is, if we project the polygon along the x direction to the yz plane, then the area of that projected polygon is a/2. The values b/2 and c/2 are the areas of the projections on the xz and xy planes, respectively. (The proof of this is left as an exercise.) So a, b, and c describe the projection of the polygon in the x, y, and z directions. But this is directly related to the direction of the plane. The amount of area projected along the z direction, for example, is proportional to the z component of the polygon's normal vector. So [a b c] is a normal vector to the plane of the polygon. The virtue of this method is that there are no special cases to check; we simply compute the sums.

Now, suppose that we make the rule that all solid objects are to be constructed out of polygons in such a way that only the light surfaces are open to the air; the dark faces meet the material inside the object. This means that when we look at an object face from the outside, it will appear to be drawn counterclockwise. (See Figure 9-4.)

If a polygon is visible, the light surface should face toward us and the dark surface should face away from us. Since a cross product or sum can be formed which gives the direction of the light face, this vector should point toward us. So if the normal vector points toward the viewer, the face is visible (a front face), otherwise, the face is hidden (a back face) and should be removed.

How can we tell whether or not a vector points toward the viewer? To do this, we examine the z component of the normal vector. If the z component is positive, then the polygon faces toward the viewer; if negative, it faces away.

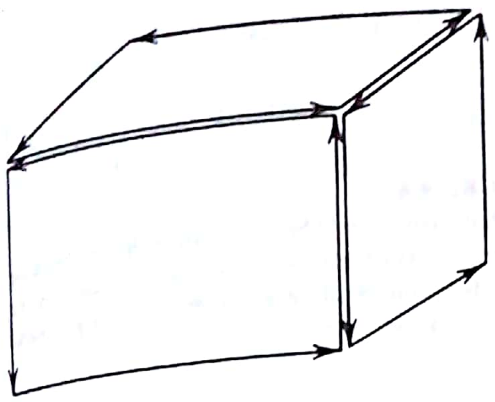

FIGURE 9–4
All exterior faces are colored light (drawn counterclockwise

This is a special case of the general problem of comparing two vectors. If we have two vectors (say R and S) and wish to compare their directions, we use the vector dot product.

$$a = R \cdot S = |R||S| \cos \theta \tag{9.4}$$

As we saw in Chapter 8, the dot product gives the product of the lengths of the two vectors times the cosine of the angle between them. This cosine factor is important to us because if the vectors are in the same direction ($0 \le \theta < \pi/2$), then the cosine is positive and the overall dot product is positive; but if the directions are opposing ($\pi/2 < \theta \le \pi$), then the cosine and the overall dot product are negative. (See Figure 9-5.)

The formula for computing a dot product is as follows:

$$\begin{aligned} a &= R \cdot S \\ &= [R_x \quad R_y \quad R_z] \cdot [S_x \quad S_y \quad S_z] \\ &= R_xS_x + R_yS_y + R_zS_z \end{aligned} \tag{9.5}$$

For the back-face check, one vector is the normal to the polygon and the other is the depth direction [0 0 1]. So the test for a back face is then a check on the sign of the z component of the normal vector. (See Figure 9-6.)

Where should all of this checking be done? Certainly it should be done after the transformation to view plane coordinates. It should follow projection (the projection can affect whether the polygon image is drawn clockwise or counterclockwise). We should also wait until after clipping has been performed so that we won't bother with polygons outside of the window. There is a point in the CLIP-POLYGON-EDGE

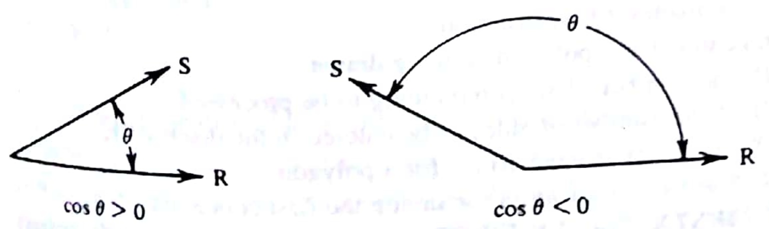

FIGURE 9-5
Cosine of the angle is positive if the vectors are pointing somewhat in the same direction and negative if they point away from each other.

Imp.

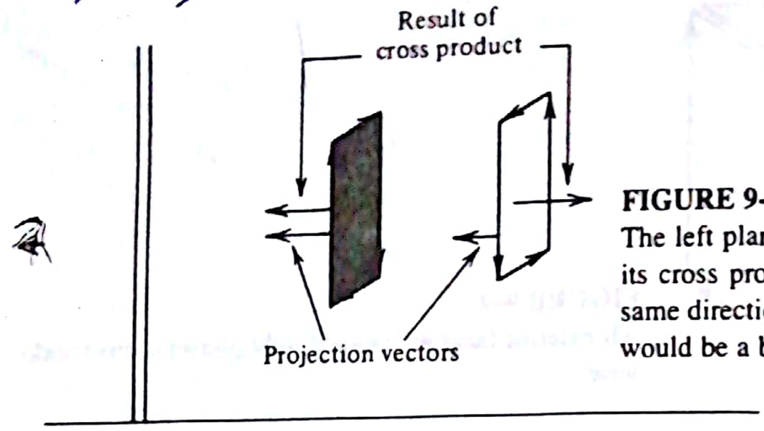

FIGURE 9-6
The left plane would be visible from the left because its cross product and projection vectors point in the same direction (positive dot product). The right plane would be a back face.

routine where the clipped polygon is sitting in a temporary storage buffer waiting to be placed into the display file. We test to be sure that the number of sides is at least 3 before the polygon is saved. At this point we can also perform our back-face check, requiring the polygon to also be a front face before it is saved.

BACK-FACE ALGORITHMS

Now let us detail the algorithms which will remove back faces. To begin with, we may not always want hidden lines removed, so we should provide the user with the means of turning the process on or off.

9.1 Algorithm SET-HIDDEN-LINE-REMOVAL(ON-OFF) User routine to set the hidden-line removal indicator

```
Argument   ON-OFF user specification for removing hidden lines
Global     HIDDEN hidden-line removal flag.
BEGIN
    HIDDEN ← ON-OFF;
    RETURN;
END;
```

The following is the modified CLIP-POLYGON-EDGE routine, which now includes a check for back faces when the HIDDEN flag is true.

9.2 Algorithm CLIP-POLYGON-EDGE(OP, X, Y, Z) (Revision of algorithm 8.35) Close and enter a clipped polygon into the display file

```
Arguments  OP, X, Y, Z a display-file instruction
Global     PFLAG indicates that a polygon is being drawn
           COUNT-IN the number of sides remaining to be processed
           COUNT-OUT the number of sides to be entered in the display file
           IT, XT, YT temporary storage arrays for a polygon
           NEEDFIRST array of indicators for saving the first command
           FIRSTOP, FIRSTX, FIRSTY, FIRSTZ arrays for saving the first command
           CLOSING indicates the stage in polygon
```

```
            FRONT-FLAG, BACK-FLAG indicate whether front and back clipping is
            done
            HIDDEN flag for hidden-line removal
Local       I for stepping through the polygon sides
BEGIN
  COUNT-IN ← COUNT-IN − 1;
  CLIP-BACK(OP, X, Y, Z);
  IF COUNT-IN ≠ 0 THEN RETURN;
  close the clipped polygon
  CLOSING ← 5;
  IF BACK-FLAG AND NOT NEEDFIRST[5] THEN
  CLIP-BACK(FIRSTOP[5], FIRSTX[5], FIRSTY[5], FIRSTZ[5]);
  CLOSING ← 6;
  IF FRONT-FLAG AND NOT NEEDFIRST[6] THEN
  CLIP-FRONT(FIRSTOP[6], FIRSTX[6], FIRSTY[6], FIRSTZ[6];
  CLOSING ← 1;
  IF NOT NEEEDFIRST[1] THEN
  CLIP-LEFT(FIRSTOP[1], FIRSTX[1], FIRSTY[1], FIRSTZ[1]);
  CLOSING ← 2;
  IF NOT NEEDFIRST[2] THEN
  CLIP-RIGHT(FIRSTOP[2], FIRSTX[2], FIRSTY[2], FIRSTZ[2]);
  CLOSING ← 3;
  IF NOT NEEDFIRST[3] THEN
  CLIP-BOTTOM(FIRSTOP[3], FIRSTX[3], FIRSTY[3], FIRSTZ[3]);
  CLOSING ← 4;
  IF NOT NEEDFIRST[4] THEN
  CLIP-TOP(FIRSTOP[4], FIRSTX[4], FIRSTY[4], FIRSTZ[4]);
  CLOSING ← 0;
  PFLAG ← FALSE;
  IF COUNT-OUT < 3 THEN RETURN;
  enter the polygon into the display file
  IF HIDDEN THEN CALL BACK-FACE-CHECK(COUNT-OUT);
  ELSE
     BEGIN
        VIEWING-TRANSFORM(COUNT-OUT, XT[COUNT-OUT],
           YT[COUNT-OUT]);
        FOR I = 1 TO COUNT-OUT DO
           VIEWING-TRANSFORM(IT[I], XT[I], YT[I]);
     END;
   RETURN;
END;
```

The algorithm above uses the BACK-FACE-CHECK function to actually decide whether the polygon is a back face and enters it into the display file. It computes the z component of the vector normal to the polygon according to Equation 9.3. It then checks the sign and enters the polygon only if the z component is positive. Thus, only front faces are entered.

9.3 Algorithm BACK-FACE-CHECK(POLYSIZE) Filters out polygon drawn clockwise

Argument	POLYSIZE the number of sides on the polygon
Global	XT, YT, ZT T-buffer array storage of the vertex points
Local	C z component of a vector for the normal to the plane of the polygon
	I, J for stepping through the vertices

```
BEGIN
    C ← 0;
    FOR I = 1 TO POLYSIZE DO
        BEGIN
            IF I = POLYSIZE THEN J ← 1
            ELSE J ← I + 1;
            C ← C + ( (XT[I] − XT[J]) * (YT[I] + YT[J]));
        END;
    IF C ≤ 0 THEN RETURN;
    VIEWING-TRANSFORM(POLYSIZE, XT[POLYSIZE], YT[POLYSIZE]);
    FOR I = 1 TO POLYSIZE DO VIEWING-TRANSFORM(IT[I], XT[I], YT[I]);
    RETURN;
END;
```

We have added a new global flag, so we should include it in the initializations. We shall make the default hidden-line removal setting FALSE, so that hidden lines will not be removed unless such action is explicitly requested by the user.

9.4 Algorithm INITIALIZE-9A Initialization routine

```
BEGIN
    INITIALIZE-8;
    SET-HIDDEN-LINE-REMOVAL(FALSE);
    RETURN;
END;
```

We have just seen how to remove many of the lines which would be hidden by an object's bulk. The method will suffice for single convex objects, but may be inadequate when several objects or concave surfaces are involved. (See Figure 9-7.) We now survey some of the techniques used to solve the full hidden-surface and hidden-line problems.

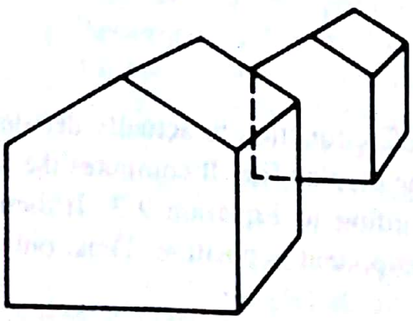

FIGURE 9-7
Hidden lines among front faces.

Z BUFFERS

Consider raster displays backed by a frame buffer. These displays can show surfaces (filled polygons) as well as lines. If we are using such a display, another way to state the hidden-surface problem is that we want to arrange the frame buffer so that the color displayed at any pixel is that of the surface closest to the viewer for that point. To do this, we must somehow compare all of the surfaces which are projected onto the pixel and decide which one can be seen. We must sort the polygons according to their position in space. This notion of *geometrically sorting* the surfaces is central to hidden surface and line removal.

One simple approach to this problem relies on a device called a *Z buffer*. The Z buffer was described by Catmull [CAT74]. It is a large array with an entry for each pixel on the display (like a frame buffer). The Z buffer is used to save the z coordinate values. It helps us to sort out the polygons by keeping track of the z position of the surfaces being displayed. When the frame buffer is cleared, the Z-buffer elements are all set to a very large negative value (a value which is beyond anything which will be imaged). The initial value may be thought of as the z position of the background. Polygons will be entered one by one into the frame buffer by the display-file interpreter, using scan conversion algorithms such as those discussed in Chapter 3. Suppose that the algorithm which turns on each pixel of the polygon knows the projected z position of the point being displayed. It could then compare the z position of the polygon point with the Z-buffer value and decide if the new surface is in front of or behind the current contents of the frame buffer. If the new surface has a z value greater than the Z-buffer value, then it lies in front; its intensity value is entered into the frame buffer, and its z value is entered into the Z buffer. If the z value of the new surface is less than the value in the Z buffer, then it lies behind some polygon which was previously entered. The new surface will be hidden and should not be imaged. No frame buffer or Z-buffer entries will be made. The comparison is carried out on a pixel-by-pixel basis, and we must be able to find the z position of the projected point for every pixel of the polygon.

If we were to modify our system to use a Z buffer for hidden-surface removal, we might extend our display file to keep the z coordinates of the polygon vertices. From the z coordinates of the vertices, we can find the z coordinate of any interior point. We have seen in Chapter 3 (Algorithm 3.15 UPDATE-X-VALUES) how to incrementally find the x values for the polygon edges as we step through the scan lines. The same technique can be used to find the z values along the edges at each step. All we need is the starting z value and the change in z for each step in y for the polygon edges. So for each scan line, then, we still generate pairs of x coordinates which indicate where to fill; but in addition, we know a z value for each of the fill-span endpoints. By reapplying this linear interpolation technique, this time using the span's change in z for each step in the x direction, the z values at each pixel can be calculated. THE FILLIN algorithm would have to obtain the endpoint z coordinates as arguments, perform the interpolation in x, and compare the resulting z values to the Z-buffer values to decide whether to change the frame buffer intensity.

A Z buffer can be expensive. It requires a lot of memory (one entry for each pixel), and each entry must have a sufficient number of bits to distinguish the possible z values. It can also be time-consuming in that a decision must be made for every pixel instead of for the entire polygon. However, it is a very simple method, simple enough to implement in hardware to overcome the speed problems. And the time required to process a scene is proportional to the number of objects in the scene. (Some other techniques which compare every polygon against every other polygon require time proportional to the square of the number of polygons.) This, together with the continuing drop in the cost of memory, makes this method (or its extensions) an increasingly popular approach to the hidden-surface problem.

SCAN-LINE ALGORITHMS

Actually, if we are willing to collect the polygons and process them together, then a full-screen Z buffer is not needed; a scan-line Z buffer will suffice and requires much less memory. The idea was suggested by Carpenter [CAR76]. The reason a large amount of memory is needed for a Z buffer is that we process each polygon independently. As each polygon is rasterized, we must in effect be able to remember the depth of each of its pixels so that they may be compared against later polygons. A polygon may be as large as the screen, so we need a full-screen Z buffer. But we can reduce the memory requirements by processing all of the polygons together on a scan-line by scan-line basis. This in effect is repeatedly doing a Z-buffer hidden-surface removal for a screen that is only one pixel high (a single scan line). The Z buffer only needs to hold one scan line's worth of depth information. When the scan line is done, we save the result, reinitialize the Z buffer, move to the next scan line, and do the next scan-line Z-buffer sort. The key here is that all polygons are processed together. Instead of considering all scan lines for a polygon before moving on to the next polygon, we consider all polygons for a scan line before moving on to the next scan line. This can be done using algorithms much like those of Chapter 3, only instead of entering the edges of a single polygon into the edge list, we enter the sides of all the polygons. We must also be careful when pairing edges for filling that we pair edges belonging to the same polygon.

A variation of the scan-line approach does not require a Z buffer at all. Where a scan line cuts a polygon, a line segment or span is described. It is these spans which are being sorted for the scan line's hidden-surface removal. To order them, we need only determine the depth of the spans at a few points. The interesting points are the span endpoints (and intersection points if polygon faces can penetrate one another). At the x position where a span begins, we could compare its depth with that of the other spans active at that point to decide which is closest to the viewer. Likewise, when a span ends, we could determine which of the remaining spans active at that point should be shown. But often we can get by with even less work. Usually each scan line looks much like its neighbors and we can use this to simplify the calculation. If faces do not interpenetrate, and the order in x of the span ends does not change from one scan line to the next, then the depth ordering is the same for the two scan lines and no new sorting in z is needed. Depth reordering is needed only when the sweep through the scan lines encounters a new polygon, passes a polygon, or finds a change in the order of span endpoints.

THE PAINTER'S ALGORITHM

There is a property of frame buffers used by Newell [NEW72], which has become known as the *painter's algorithm*. The algorithm gets its name from the manner in which an oil painting is created. The artist begins with the background. He can, if he wishes, fill the entire canvas with the background scene. The artist then paints the foreground objects. There is no need to erase portions of the background; the artist simply paints on top of them. The new paint covers the old so that only the newest layer of paint is visible. A frame buffer has this same property. We enter a filled polygon into the frame buffer by changing the proper pixels to values corresponding to the polygon's interior style. If we then enter a second polygon "on top of" the first, some of those same pixels will be changed to correspond to the second polygon's interior style. Wherever the second polygon lies, the first polygon's pixel settings have been forgotten. The second polygon has "covered up" the first. The painter's algorithm tells us to enter first those polygons which are farthest from the viewer (the background) and enter last the objects closest to the viewer (the foreground). Hidden surfaces can be covered up by choosing the correct order to draw them and taking advantage of the properties of frame buffers. (See Figure 9-8.)

The painter's algorithm is a simple idea, but let's look at what is involved in its implementation. First of all, we cannot process each polygon independently, as we did

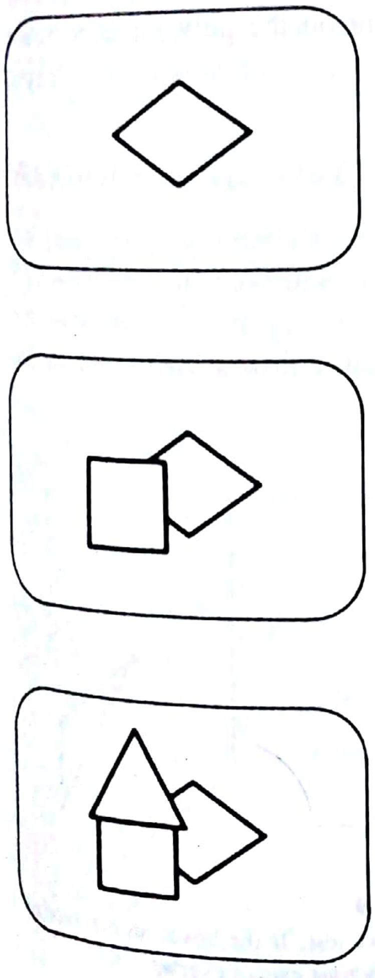

FIGURE 9-8
The most recent filled polygon overwrites previous pixel values.

in the back-face check or the full-screen Z buffer. We must compare each polygon with all of the rest to see which is in front of which. We must, in effect, sort the polygons to determine a priority for their display. The sorting will determine the order in which they will be entered into the display file (the order in which they will be drawn). For efficiency, we try to limit the number of comparisons and we try to make the comparison process fast.

COMPARISON TECHNIQUES

There are several techniques for determining the relevancy and relative position of two polygons. Not all tests may be used with all hidden-surface algorithms, and some of the tests are not always conclusive. Sometimes we can use simple tests for many of the cases and resort to more costly tests only when all else fails.

One technique that is often useful is called the *minimax test* or *boxing test*. We may not need to know the order of every polygon relative to every other polygon; it may suffice to know just the relative orderings of those polygons that overlap. So a test that can quickly tell us if two polygons do not overlap is useful. The minimax test will do just that. This test says that if we place boxes around two polygons and if the two boxes don't overlap, then the polygons within them cannot overlap. (See Figure 9-9.)

We want the boxes to be as small as possible and still contain the polygons. That means we want the top of the box to be as low as possible. The lowest we can make the top of the box is where it just touches the highest point on the polygon, that is, the

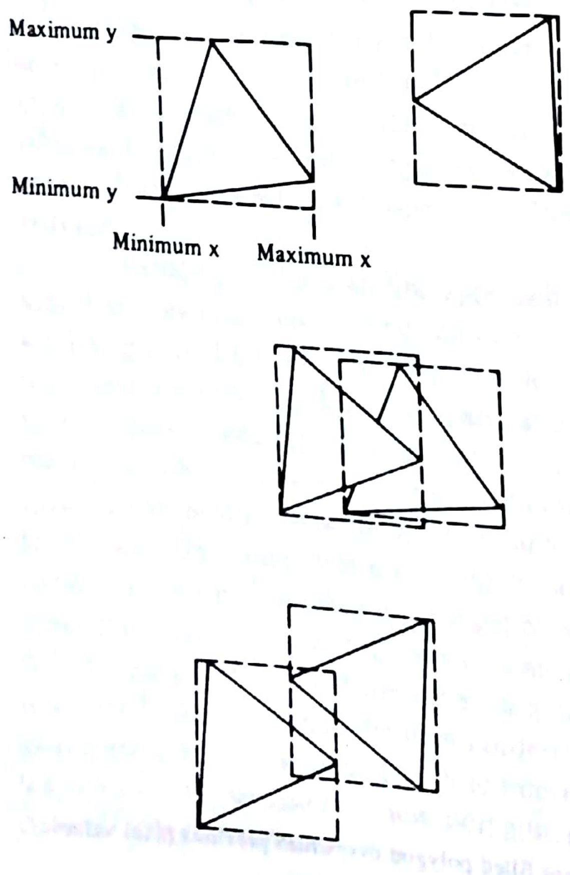

FIGURE 9-9
The minimax test. If the boxes do not overlap, then the polygons cannot overlap.

maximum of the y coordinates of the polygon's vertices. Similarly, the highest position of the bottom of the box is the minimum of the y coordinates of the vertices. Two boxes will not overlap if one is above the other, that is, if the bottom of one box is higher than the top of the other box. This means that the minimum of the y coordinates of one polygon is greater than the maximum of the y coordinates of the other polygon. We can, of course, switch the roles of the two polygons. We can also compare left and right sides by a similar test using the x coordinates.

If the minimax test is true, then the polygons do not overlap; but if the boxes do overlap, we still are not certain whether the polygons within them overlap. We will have to perform further tests.

A minimax test applied to the z coordinates can often tell the relative ordering of two polygons which do overlap in x and y. If the smallest z value for one polygon is larger than the largest z value for the other polygon, then the first polygon lies in front.

Another test which is useful is to see if all the vertices of one polygon lie on the same side of the plane containing the other polygon. If polygon P has all its vertices on the same side of the plane of polygon Q as the viewer, then P is in front of Q. If all the vertices lie in the other half-space, then P is behind Q. (See Figure 9-10.)

This test may also be inconclusive because the plane of polygon Q may intersect polygon P. If this should happen, we can try comparing the vertices of Q against the plane of P. Again, this may or may not yield results. (See Figure 9-11.)

If the above tests fail, we might break up the polygons into pieces such that the tests succeed, or we might try to find an xy point common to both polygons and compare the corresponding z-coordinate values.

WARNOCK'S ALGORITHM

An interesting approach to the hidden-surface problem was presented by Warnock [WAR69]. His method does not try to decide exactly what is happening in the scene but rather just tries to get the display right. As the resolution of the display increases, the amount of work which the algorithm must do to get the scene right also increases.

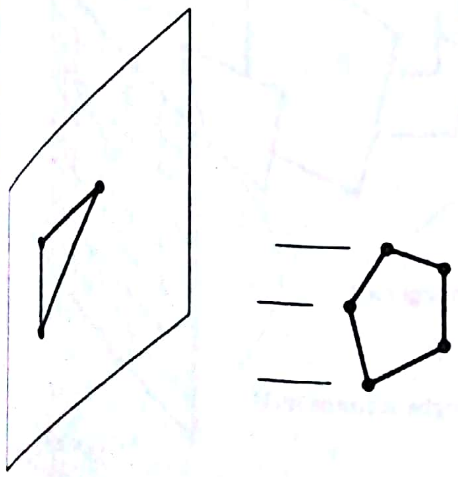

FIGURE 9-10
Vertices of one polygon on one side of the plane of another polygon.

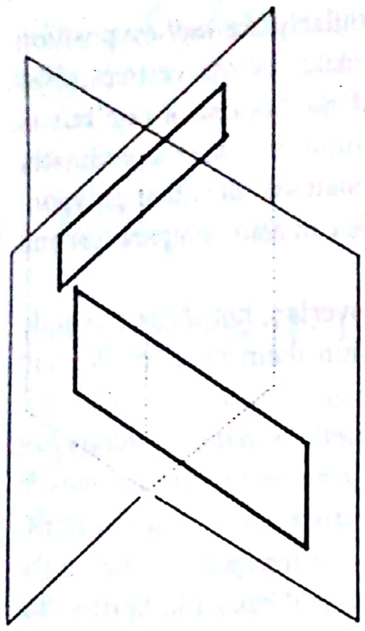

FIGURE 9-11
Vertices of each polygon straddle the plane of the other.

(This is also true for scan-line algorithms.) The algorithm divides the screen up into sample areas. In some sample areas it will be easy to decide what to do. If there are no faces within the area, then it is left blank. If the nearest polygon completely covers it, then it can be filled in with the color of that polygon. If neither of these conditions holds, then the algorithm subdivides the sample area into smaller sample areas and considers each of them in turn. This process is repeated as needed. (See Figure 9-12.) It stops when the sample area satisfies one of the two simple cases or when the sample area is only a single pixel (which can be given the color of the foremost polygon). The process can also be allowed to continue to half or quarter pixel-sized sample areas, whose color may be averaged over a pixel to provide antialiasing.

The test for whether a polygon surrounds or is disjoint from the sample area is much like a clipping test to see if the polygon sides cross the sample-area boundaries.

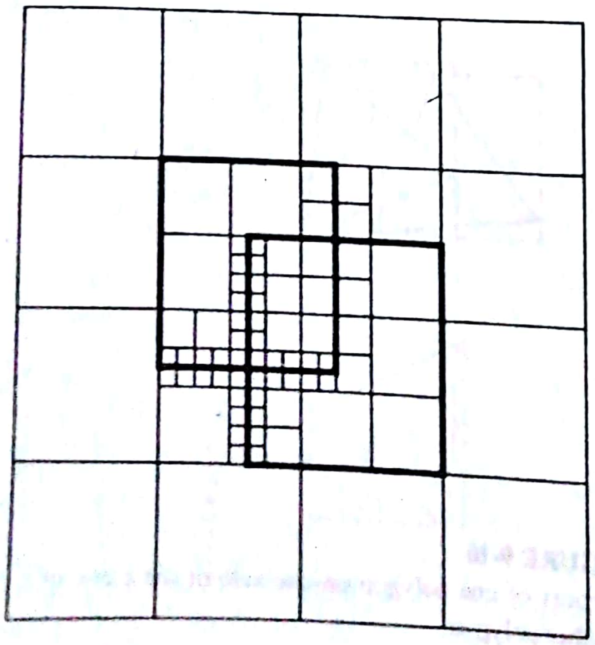

FIGURE 9-12
Subdivision of a scene.

Actually, the minimax test can be employed to identify many of the disjoint polygons. A simple test for whether a polygon is in front of another is a comparison of the z coordinates of the polygon planes at the corners of the sample area.

At each subdivision, information learned in the previous test can be used to simplify the problem. Polygons which are disjoint from the tested sample area will also be disjoint from all of the subareas and do not need further testing. Likewise, a polygon which surrounds the sample area will also surround the subareas.

FRANKLIN ALGORITHM

We mentioned how the number of possible comparisons of polygons grows as the square of the number of polygons in the scene. Many of the hidden-surface algorithms exhibit this behavior and have serious performance problems on complex scenes. Franklin [FRA80] developed an approach which gives linear time behavior for most scenes. This is done by overlaying a grid of cells on the scene (similar to Warnocks approach, only these cells are not subdivided). The size of the cells is on the order of the size of an edge in the scene. At each cell the algorithm looks for a covering face and determines which edges are in front of this face. It then computes the intersections of these edges and determines their visibility. (See Figure 9-13.) The idea is that as objects are added to the scene and the number of polygons increases, the new objects will either be hidden by objects already in the scene or will hide other objects in the scene. While the number of objects increases, the complexity of the final scene (after hidden portions are removed) does not increase. By considering only the edges in front of the covering face for a cell, the algorithm considers only the edges likely to be in the final image. Although the total number of edges may increase, this increase occurs, for the most part, behind the covering faces, and the number of edges in front will remain small.

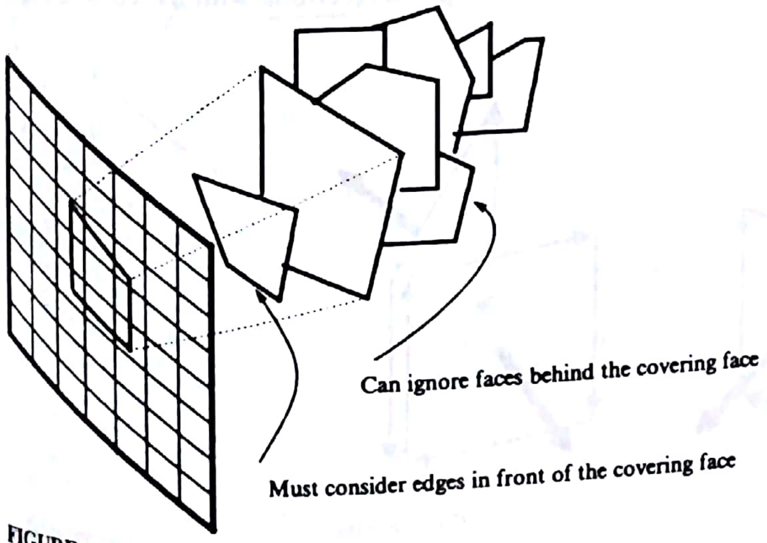
Can ignore faces behind the covering face
Must consider edges in front of the covering face

FIGURE 9-13
Faces behind the first face to cover a cell can be ignored for that cell.

HIDDEN-LINE METHODS

The hidden-surface techniques such as the painter's algorithm, which rely on the properties of the frame buffer, are not sufficient for calligraphic displays. On such displays we must not draw the hidden portions of lines. This means that for each line, we must decide not only what objects lie in front of it but also just how those objects hide it. Calligraphic displays were available long before raster displays became economical, and the hidden-line problem was attacked before the hidden-surface problem.

The first solution [ROB63] compared lines to objects. For each object, it considered relevant edges to see if the object hid them. The object might not hide an edge at all or might hide it entirely. It might hide an end, making the visible portion of the edge smaller, or hide the middle, making two smaller visible line segments. After comparison of the line and object, the resulting visible line segments were compared in turn to the remaining objects. A segment which survives comparison to all objects is drawn.

We do not have to compare the line against all of the polygon edges in an object in order to tell if the object hides the line. The only edges which can change whether the line is visible or not are those on the boundary where a front face meets a back face. These are called *contour edges*. We can find the contour edges by examining the object. For a solid object, each edge has two polygons adjacent to it. If the polygons which meet at the edge are both front faces or both back faces, then we have an *interior edge*; but if one is a front face and the other a back face, then it is a contour edge. (See Figure 9-14.)

Instead of comparing all lines to each object, we can compare the contour edges of all objects to each line. For each intersection of a line with a contour edge, the line either passes behind an object or emerges from it. So with each such intersection, the number of faces hiding the line either increases or decreases by 1. Appel [APP67] has termed the number of faces hiding a line its *quantitative invisibility*. His method for hidden-line removal is to find the quantitative invisibility for an initial point on a line, and then to follow along the connected lines, finding intersections with the contour edges

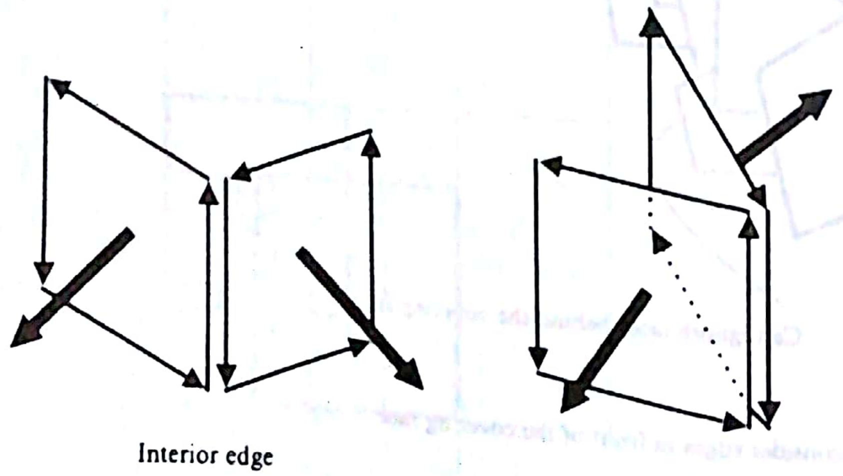

FIGURE 9-14
Interior and contour edges.

and maintaining the quantitative invisibility. The portions of the lines with quantitative invisibility 0 are drawn.

BINARY SPACE PARTITION

Now let's implement a hidden-surface algorithm. The method we shall use is based on the painter's algorithm and is credited to Fuchs [FUC80]. It is called binary space partition (BSP). The idea is to sort the polygons for display in back-to-front order. To order the polygons we use the test which compares all the vertices of one polygon against the plane of another polygon. We extend this test so that if the plane intersects the polygon, we divide the polygon along the plane. Using this test we can pick one polygon and compare all the other polygons to it, splitting them into two groups, those in front and those behind. For each of these two subgroups we can again select a polygon and use it to separate the subgroup. This process is repeated until all polygons have been sorted. The sort works in the same fashion as Hoare's Quicksort algorithm, repeatedly separating the polygons into smaller and smaller groups. The result may be pictured as a binary tree. At each node of the tree is a polygon. In one subtree, or branch, are all the polygons in front of the plane of this polygon, and in the other branch are those polygons which lie behind it. (See Figure 9-15.) Once we have created this tree, we can perform an in-order traversal to obtain the polygons in back-to-front order.

Before discussing the details of the algorithm, we note that it requires the collection and sorting of all the polygons before any of them are drawn. We shall need some storage area to collect the polygon instructions until we are ready to sort them. We shall need to know when to start collecting polygons and also when all polygons have been collected so that sorting can proceed. We shall also need to save more information about a polygon than just its vertices. We shall need to save the edge and fill styles for

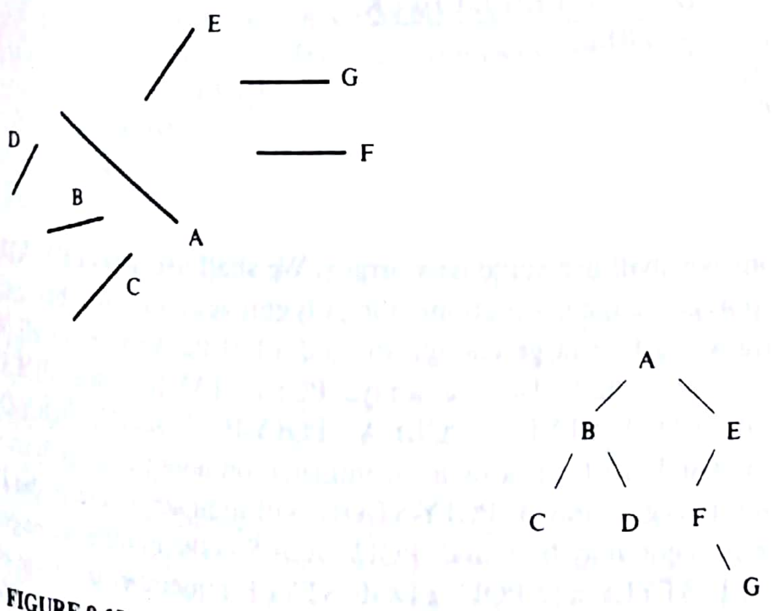

FIGURE 9-15
An ordered tree of polygons.

the polygon. The change-of-style commands will have to be inserted to match the order in which the polygons are actually drawn (not the order that they are entered by the user). It will also be useful to save the equation for the plane of the polygon. This can be used in the polygon comparison.

Let's first consider when to save and when to sort the polygons. In our system we shall limit hidden-line processing to single display-file segments. We shall sort out all the objects within a segment, but we will not check to see if an object in one segment hides an object in a different segment. With this organization, a reasonable point at which to sort and save the polygons is just before the display-file segment is closed. When the segment is closed, we know that all relevant polygons have been entered and no other display-file segment's polygons will be around to confuse us. We shall therefore modify the CLOSE-SEGMENT algorithm to include a call to our hidden-surface removal routine. We shall use the HIDDEN flag to tell us to save polygons, so polygons will be saved any time this flag is TRUE; but with each CLOSE-SEGMENT, the collected polygons will be sorted and moved to the display file. This groups them according to segment.

The modified CLOSE-SEGMENT routine calls the HIDDEN-SURFACE-CHECK routine to actually do the sorting and saving. It looks as follows:

```
9.5 Algorithm CLOSE-SEGMENT (Modification of algorithm 5.2)
Global     NOW-OPEN the name of the currently open segment
           FREE the index of the next free display-file cell
           HIDDEN the hidden-surface removal flag
           SEGMENT-START, SEGMENT-SIZE arrays for start and size of the seg-
           ments
BEGIN
    IF NOW-OPEN = 0 THEN RETURN ERROR 'NO SEGMENT OPEN';
    DELETE-SEGMENT(0);
    IF HIDDEN THEN HIDDEN-SURFACE-CHECK;
    SEGMENT-START[0] ← FREE;
    SEGMENT-SIZE[0] ← 0;
    NOW-OPEN ← 0;
    RETURN;
END;
```

To save the polygons we shall use some new arrays. We shall use arrays ID, XD, YD, and ZD to hold the polygon edge instructions (the polygon vertices). We shall call this the D buffer. The arrays must be large enough to hold all of the vertices of all of the polygons being sorted. We shall also use arrays POLY-START, POLY-SIDES, POLY-FILL-STYLE, POLY-EDGE-STYLE, POLY-A, POLY-B, POLY-C, POLY-D, POLY-FRONT, and POLY-BACK to form a table of information about each polygon. There is one entry for each polygon drawn. POLY-START will indicate where in the D buffer the vertices for the polygon may be found. POLY-SIDES is the number of edges on the polygon. POLY-FILL-STYLE and POLY-EDGE-STYLE remember the style of the polygon. The arrays POLY-A, POLY-B, POLY-C, and POLY-D contain the coefficients for the equation of the plane of the polygon. POLY-FRONT and POLY-BACK will hold the indices of other polygon entries. They are the branches that will form our sorted tree of polygons.

We shall modify our BACK-FACE-CHECK algorithm to save the polygons. The modifications will be to calculate the entire normal vector to the polygon plane (not just its z component) and to check the solid flag, calling the SAVE-POLY-FOR-HSC routine if the polygon is filled.

9.6 Algorithm BACK-FACE-CHECK(POLYSIZE) (Revision of algorithm 9.3) Filters out polygon drawn clockwise

Argument	POLYSIZE the number of sides on the polygon
Global	XT, YT, ZT T-buffer array storage of the vertex points
	SOLID a flag which indicates polygon filling
	A, B, C a vector for the normal to the plane of the polygon
Local	I, J for stepping through the vertices

```
BEGIN
   A ← 0;
   B ← 0;
   C ← 0;
   FOR I = 1 TO POLYSIZE DO
      BEGIN
         IF I = POLYSIZE THEN J ← 1
         ELSE J ← I + 1;
         A ← A + ((YT[I] − YT[J]) * (ZT[I] + ZT[J]));
         B ← B + ((ZT[I] − ZT[J]) * (XT[I] + XT[J]));
         C ← C + ((XT[I] − XT[J]) * (YT[I] + YT[J]));
      END;
   IF C ≤ 0 THEN RETURN;
   IF SOLID THEN SAVE-POLY-FOR-HSC(POLYSIZE, A, B, C)
   ELSE
      BEGIN
         VIEWING-TRANSFORM(POLYSIZE, XT[POLYSIZE], YT[POLYSIZE]);
         FOR I = 1 TO POLYSIZE DO VIEWING-TRANSFORM(IT[I], XT[I],
            YT[I]);
      END;
   RETURN;
END;
```

The SAVE-POLY-FOR-HSC algorithm enters the polygon information into the polygon table at POLY and into the D buffer starting at DFREE. Most of the entries are straightforward. As we saw in Chapter 8, the A, B, and C coefficients in the equation of the plane (Equation 8.4) can be the elements of the normal vector which is determined in BACK-FACE-CHECK. The fourth coefficient D may be found by substituting for x, y, and z in the equation the coordinates of a point known to be on the plane. This is what is done for the POLY-D array.

By setting POLY-FRONT to POLY − 1, we form a linked list of all the polygons (each linked to its predecessor). We shall use the linked list to access the polygons when sorting.

9.7 Algorithm SAVE-POLY-FOR-HSC(POLYSIZE, A, B, C) Save a polygon in the polygon table for the hidden-surface check

Arguments	POLYSIZE the number of sides on the polygon
	A, B, C a vector normal to the plane of the polygon
Global	POLY index of the next free polygon-table cell
	CURRENT-FILL-STYLE the interior style of the polygon
	CURRENT-LINE-STYLE the edge style of the polygon
	IT, XT, YT, ZT arrays containing the vertices of one polygon
	ID, XD, YD, ZD D-buffer arrays containing the vertices of all polygons involved in the hidden-surface check
	DFREE the next free D-buffer cell
	POLY-START, POLY-SIDES, POLY-FILL-STYLE, POLY-EDGE-STYLE, POLY-A, POLY-B, POLY-C, POLY-D, POLY-FRONT, POLY-BACK the polygon table
Local	I for stepping through the polygon vertices
Constant	POLYGON-TABLE-SIZE the size of the polygon-table arrays
	D-BUFFER-SIZE the size of the D buffer

```
BEGIN
   IF POLY = POLYGON-TABLE-SIZE THEN
      RETURN ERROR 'POLYGON TABLE OVERFLOW';
   POLY-START[POLY] ← DFREE;
   POLY-SIDES[POLY] ← POLYSIZE;
   POLY-FILL-STYLE[POLY] ← CURRENT-FILL-STYLE;
   POLY-EDGE-STYLE[POLY] ← CURRENT-LINE-STYLE;
   POLY-A[POLY] ← A;
   POLY-B[POLY] ← B;
   POLY-C[POLY] ← C;
   POLY-D[POLY] ← - (A * XT[1] + B * YT[1] + C * ZT[1]);
   POLY-FRONT ← POLY - 1;
   POLY-BACK ← 0;
   POLY ← POLY + 1;
   FOR I = 1 TO POLYSIZE DO
      BEGIN
         IF DFREE > D-BUFFER-SIZE THEN
            RETURN ERROR 'D BUFFER OVERFLOW';
         ID[DFREE] := IT[I];
         XD[DFREE] := XT[I];
         YD[DFREE] := YT[I];
         ZD[DFREE] := ZT[I];
         DFREE := DFREE + 1;
      END;
   RETURN;
END;
```

Now let's consider what happens when a display-file segment is closed and the HIDDEN-SURFACE-CHECK algorithm is called. This procedure first determines if there is anything to sort and save. If so, it picks one of the polygons to start the sorting process. This polygon will be the root of the sorted tree of polygons. The algorithm sorts the polygons, saves them in the display file in back-to-front order, and then resets the polygon table and D buffer so that future polygons will be sorted as a new set.

9.8 Algorithm HIDDEN-SURFACE-CHECK A routine to remove hidden surfaces

```
Global    POLY index of next free cell in the polygon table
          SOLID the polygon-filling flag
          DFREE the next free D-buffer cell
Local     ROOTPOLY the root of the space partition tree
BEGIN
    IF POLY = 1 THEN RETURN;
    ROOTPOLY ← POLY - 1;
    SORT-POLYGONS(ROOTPOLY);
    SAVE-POLYGONS-IN-ORDER(ROOTPOLY);
    POLY ← 1;
    DFREE ← 1;
    RETURN;
END;
```

The sorting routine is given in recursive form (it calls itself). Nonrecursive versions are possible, but they are more complex. The algorithm checks to see if there is something to sort; if not, it returns. It uses a loop to step through the polygons in a linked list. The index of the next polygon in the list is found in the POLY-FRONT array for the current polygon. It compares the first polygon in the list (ROOT-NODE) against all of the other polygons in the list (TEST-POLY). The COMPARE-POLYS procedure not only decides on which side of ROOT-NODE each TEST-POLY lies but also links the TEST-POLY onto a sublist accordingly. When the loop has finished, the list will have been divided into two sublists. There will be a list of the polygons in front of the ROOT-NODE, which will be attached to the ROOT-NODE's POLY-FRONT entry, and there will be a list of those behind, attached to POLY-BACK for the ROOT-NODE. The algorithm, then, recursively sorts each of these two sublists.

9.9 Algorithm SORT-POLYGONS(ROOT-NODE) A routine to build a sorted binary tree of polygons

```
Argument  ROOT-NODE the polygon at the root of the tree
Global    POLY-FRONT, POLY-BACK links for the tree branches
Local     TEST-POLY, NEXT-POLY polygons to be compared to the root
BEGIN
    IF ROOT-NODE = 0 THEN RETURN;
    TEST-POLY ← POLY-FRONT[ROOT-NODE];
    POLY-FRONT[ROOT-NODE] ← 0;
    WHILE TEST-POLY ≠ 0 DO
        BEGIN
            NEXT-POLY ← POLY-FRONT[TEST-POLY];
            COMPARE-POLYS(ROOT-NODE, TEST-POLY);
            TEST-POLY ← NEXT-POLY;
        END;
        SORT-POLYGONS(POLY-FRONT[ROOT-NODE]);
        SORT-POLYGONS(POLY-BACK[ROOT-NODE]);
        RETURN;
    END;
```

There is actually a lot of work that goes on in the COMPARE-POLYS algorithm. It compares each vertex of the TEST polygon against the plane of the ROOT polygon and decides which side it is on. But it will also split the TEST polygon into two separate polygons if it is intersected by the plane. The technique is very similar to what we used for clipping. When clipping, we compare points against a clipping plane and only keep those points which lie on one side. In COMPARE-POLYS we keep the points on both sides but in two separate groups, so we end up with two polygons. As with clipping, when an edge crosses the plane, we calculate the intersection point, using it to build an ''invisible'' edge along the plane boundary. (See Figure 9-16.)

The algorithm determines on which side of the plane a point lies by checking the sign of the result of substituting the coordinates of the point into the expression for the plane (see Equations 8.69 and 8.70). It finds the first vertex point which is not on the plane and then compares following vertices, checking for a change in sign which indicates a crossing of the plane. If a side ever crosses the plane, then we will have to split the polygon. To do this, we form a new polygon starting at DFREE (the next free cell in the D buffer). The new polygon begins with the first intersection point. We must also save the points for the other side of the split. This side will also include the intersection point. Since the split polygon can have more vertices that the original, we cannot store them in place, so we use arrays IE, XE, YE, and ZE (the E buffer) to save the other half of the split polygon. The E buffer must be large enough to hold one polygon. After the first intersection is discovered, vertices are copied to either the D-buffer piece or the E-buffer piece. Intersection points go to both.

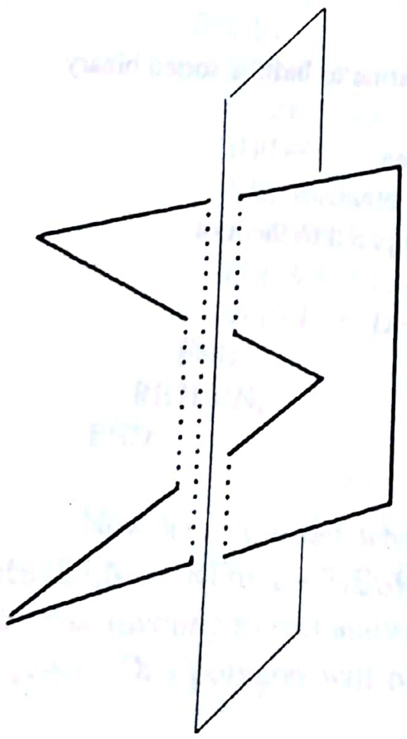

FIGURE 9-16
Dividing the polygon along a plane.

Although we are finding the sign of each vertex, we are actually looking for changes in sign between two vertices; that is, we are considering the edge between the vertex pair. After stepping through all vertices, we must consider the first once again in order to characterize the final edge, that is, to close the polygon.

9.10 Algorithm COMPARE-POLYS(ROOT, TEST) Finds on which side of ROOT the TEST polygon lies and links it to that branch. If the plane of ROOT intersects the TEST polygon, then it is split

Arguments	ROOT the polygon providing the plane used in the comparison
	TEST the polygon being compared and linked (and possibly split)
Global	POLY-START, POLY-SIDES, POLY-A, POLY-B, POLY-C, POLY-D, POLY-FRONT the polygon table
	ID, XD, YD, ZD the D-buffer arrays containing polygon vertices
	DFREE the next free D-buffer cell
	IE, XE, YE, ZE the E-buffer arrays for vertices of split-off polygon
	EFREE the next free E-buffer cell
Local	FIRST-V index of the first vertex of the polygon
	LAST-V index of the last vertex of the polygon
	CROSS-V index of the vertex just before the polygon crosses the test plane
	SPLIT-V index of the first vertex of the split polygon
	J for stepping through the polygon vertices
	STEST the first nonzero sign parameter
	SLAST the sign parameter of the previous vertex
	SNEW the sign parameter for the current vertex
	U0, U, V test point parameters
	W parameter for the intersection point
	XP, YP, ZP the intersection point
	CHANGED-BEFORE a flag to indicate the first time the polygon crosses the plane
	CROSSED a flag to indicate if the polygon edge crossed the plane

```
BEGIN
  FIRST-V ← POLY-START[TEST];
  LAST-V ← FIRST-V + POLY-SIDES[TEST] − 1;
  CROSS-V ← FIRST-V;
  U0 ← POLY-A[ROOT] * XD[FIRST-V] + POLY-B[ROOT] * YD[FIRST-V]
  + POLY-C[ROOT] * ZD[FIRST-V] + POLY-D[ROOT];
  STEST ← SIGNOF[U0];
  find the first vertex not on the plane
  U ← U0;
  WHILE STEST = 0 AND CROSS-V < LAST-V DO
     BEGIN
        CROSS-V ← CROSS-V + 1;
        U ← POLY-A[ROOT] * XD[CROSS-V] + POLY-B[ROOT] * YD[CROSS-V]
           + POLY-C[ROOT] * ZD[CROSS-V] + POLY-D[ROOT];
          STEST ← SIGNOF(U);
     END;
  IF STEST = 0 THEN
     BEGIN
        test polygon was contained in the plane so treat as in front
```

```
            POLY-FRONT[TEST] ← POLY-FRONT[ROOT];
            POLY-FRONT[ROOT] ← TEST;
            RETURN;
        END;
    SLAST ← STEST;
    EFREE ← 1;
    SPLIT-V ← DFREE;
    CHANGED-BEFORE ← FALSE;
    FOR J = CROSS-V + 1 TO LAST-V DO
        BEGIN
            step through all vertices
            V ← POLY-A[ROOT] * XD[J] + POLY-B[ROOT] * YD[J]
                + POLY-C[ROOT] * ZD[J] + POLY-D[ROOT];
            SNEW ← SIGNOF(V);
            IF SNEW = - SLAST THEN
                BEGIN
                    point has crossed the plane, find intersection
                    CROSSED ← TRUE;
                    W ← U / (U - V);
                    XP ← (XD[J] - XD[J - 1]) * W + XD[J - 1];
                    YP ← (YD[J] - YD[J - 1]) * W + YD[J - 1];
                    ZP ← (ZD[J] - ZD[J - 1]) * W + ZD[J - 1];
                    CALL PUT-IN-D(ID[J], XP, YP, ZP);
                    CALL PUT-IN-E(ID[J], XP, YP, ZP);
                    SLAST ← SNEW;
                END
            ELSE CROSSED ← FALSE;
            IF SNEW = STEST OR (SNEW = 0 AND SLAST = STEST) THEN
                BEGIN
                    point belongs on the first side of the plane
                    IF CROSSED THEN IE[EFREE - 1] ← 1;
                    IF CHANGED-BEFORE THEN PUT-IN-E(ID[J], XD[J], YD[J], ZD[J])
                    ELSE CROSS-V ← CROSS-V + 1;
                END;
            ELSE
                BEGIN
                    point belongs on the second side of the plane
                    CHANGED-BEFORE ← TRUE;
                    IF CROSSED THEN ID[DFREE - 1] ← 1;
                    CALL PUT-IN-D(ID[J], XD[J], YD[J], ZD[J]);
                END;
            U ← V;
        END;
    now we must close the polygon
    IF SLAST ≠ STEST THEN
        BEGIN
            W ← V / (V - U0);
            XP ← (XD[FIRST-V] - XD[J]) * W + XD[J];
            YP ← (YD[FIRST-V] - YD[J]) * W + YD[J];
            ZP ← (ZD[FIRST-V] - ZD[J]) * W + ZD[J];
            CALL PUT-IN-D(ID[FIRST-V], XP, YP, ZP);
            CALL PUT-IN-E(1, XP, YP, ZP);
        END;
    finally, hang the polygon(s) on the ROOT node
    IF SPLIT-V ≠ DFREE THEN SET-UP-SPLIT-POLY(ROOT, TEST, SPLIT-V, STEST);
    ADJUST-ORIGINAL-POLY(ROOT, TEST, FIRST-V, CROSS-V, STEST);
    RETURN;
END;
```

Once the polygon has been tested (and split if necessary), two routines are used to clean things up. If the polygon is split, then the vertices for one half are copied in the D buffer. But we do not have a corresponding polygon-table entry. The routine SET-UP-SPLIT-POLY forms this table entry. It also links the polygon onto either the POLY-FRONT or POLY-BACK branch of the ROOT polygon, depending on whether it lies in front or in back of the ROOT polygon plane.

9.11 Algorithm SET-UP-SPLIT-POLY(ROOT, TEST, SPLIT-V, STEST) Completes a polygon-table entry for the split-off polygon and attaches it to the root polygon

Arguments	ROOT the root polygon
	TEST the original test polygon
	SPLIT-V index of the first vertex of the split-off polygon
	STEST indicates on which side of the root the split polygon lies
Global	POLY-START, POLY-SIDES, POLY-FILL-STYLE, POLY-EDGE-STYLE, POLY-A, POLY-B, POLY-C, POLY-D, POLY-FRONT, POLY-BACK the polygon table
	POLY index of the next free polygon-table cell

```
BEGIN
    POLY-START[POLY] ← SPLIT-V;
    POLY-SIDES[POLY] ← DFREE − SPLIT-V;
    POLY-FILL-STYLE[POLY] ← POLY-FILL-STYLE[TEST];
    POLY-EDGE-STYLE[POLY] ← POLY-EDGE-STYLE[TEST];
    POLY-A[POLY] ← POLY-A[TEST];
    POLY-B[POLY] ← POLY-B[TEST];
    POLY-C[POLY] ← POLY-C[TEST];
    POLY-D[POLY] ← POLY-D[TEST];
    POLY-BACK[POLY] ← 0;
    IF STEST < 0 THEN
        BEGIN
            POLY-FRONT[POLY] ← POLY-FRONT[ROOT];
            POLY-FRONT[ROOT] ← POLY;
        END
    ELSE
        BEGIN
            POLY-FRONT[POLY] ← POLY-BACK[ROOT];
            POLY-BACK[ROOT] ← POLY;
        END;
    POLY ← POLY + 1;
    RETURN;
END;
```

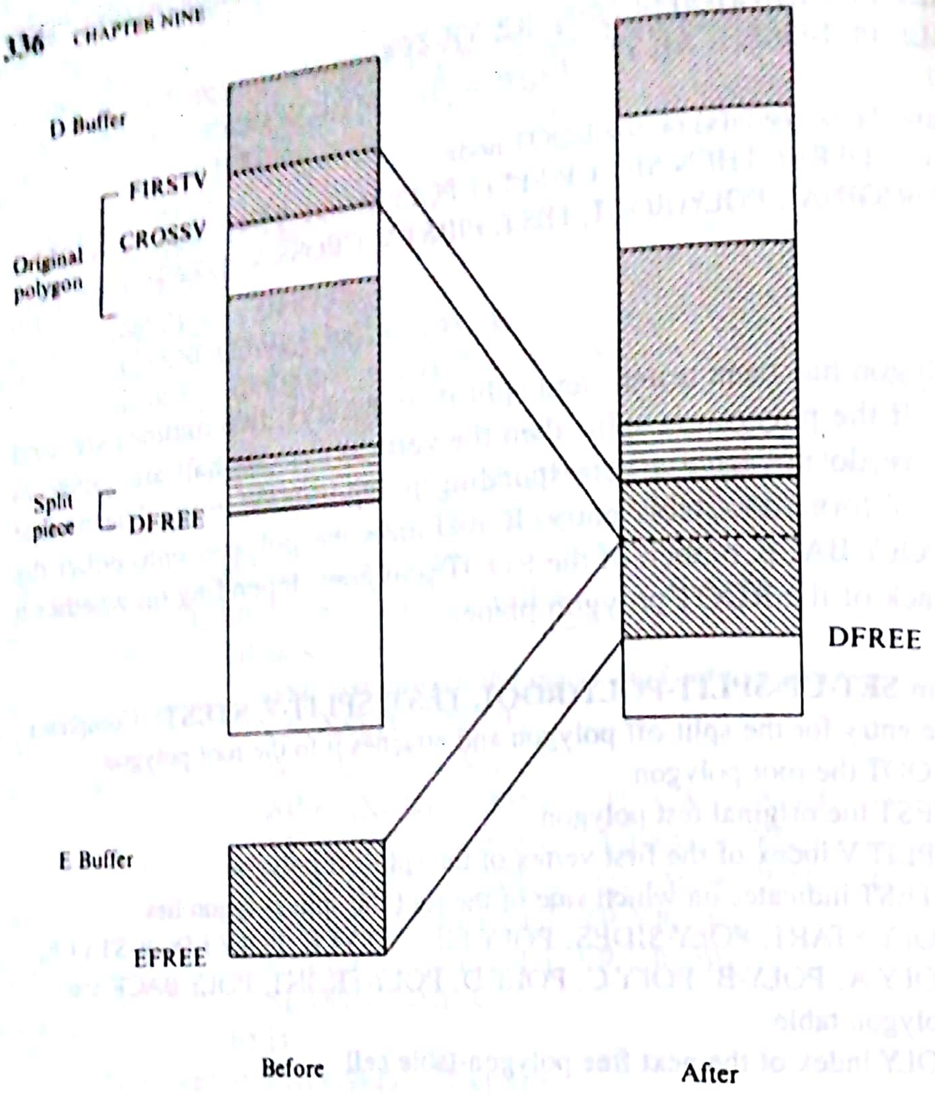

FIGURE 9-17
Case where the number of sides increases with the split.

The second cleanup routine is called ADJUST-ORIGINAL-POLY. If the polygon was split, then the points for one of the subpolygons may be found partly in the D buffer and partly in the E buffer. The D buffer holds the vertices up to the first intersection with the ROOT polygon plane. The remainder is in the E buffer. These two parts must be recombined. If this subpolygon now has fewer vertices than the original, then the vertices in the E buffer can be copied back into the original D-buffer locations. If, however, the size of the polygon has grown, then both the D-buffer piece and the E-buffer piece are copied to a new D-buffer location. The result is linked onto the appropriate branch of the ROOT polygon. (See Figures 9-17 and 9-18.)

9.12 Algorithm ADJUST-ORIGINAL-POLY(ROOT, TEST, FIRST-V, CROSS-V, STEST) Combines all vertices of the polygon in the D buffer and attaches it to the root polygon

Arguments	ROOT the root polygon
	TEST the original test polygon
	FIRST-V index of the first vertex of the polygon
	CROSS-V the vertex just before the polygon crosses the test plane
	STEST indicates on which side of the root the split polygon lies

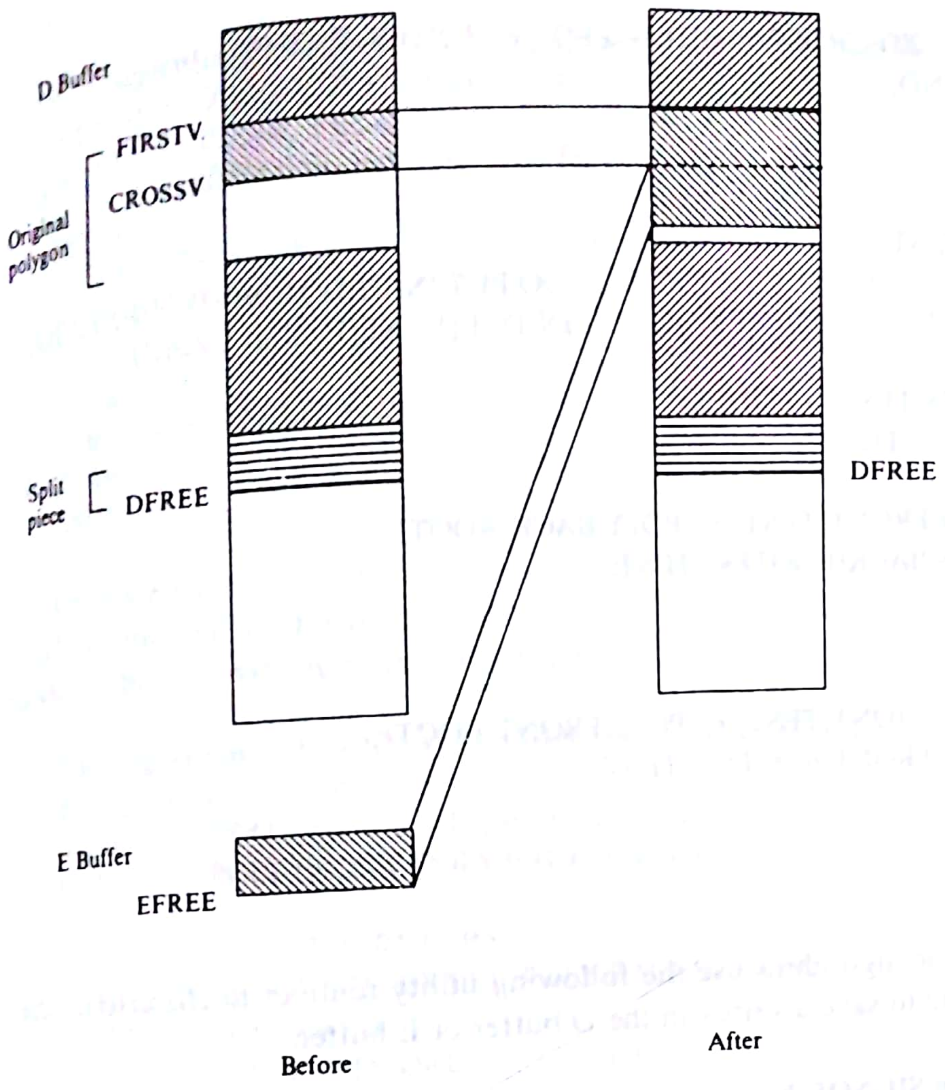

FIGURE 9-18
Case where the number of sides does not increase with the split.

```
Global    POLY-START, POLY-SIDES, POLY-FRONT, POLY-BACK from the polygon
          table
          ID, XD, YD, ZD the D-buffer arrays
          DFREE the next free D-buffer cell
          IE, XE, YE, ZE the E-buffer arrays
          EFREE the next free E-buffer cell
Local     J an index for stepping through the vertices
          SIDES the number of sides on the polygon
BEGIN
   SIDES ← EFREE + CROSS-V − FIRST-V;
   IF SIDES ≤ POLY-SIDES[TEST] THEN
      BEGIN
         the adjusted polygon will fit in the original D-buffer space
         FOR J = 1 TO EFREE − 1 DO
            BEGIN
               ID[CROSS-V + J] ← IE[J];
               XD[CROSS-V + J] ← XE[J];
               YD[CROSS-V + J] ← YE[J];
```

```
                    ZD[CROSS-V + J] ← ZE[J];
                END;
            END
        ELSE
            BEGIN
                POLY-START[TEST] ← DFREE;
                FOR J = FIRST-V TO CROSS-V DO PUT-IN-D(ID[J], XD[J], YD[J], ZD[J]);
                FOR J = 1 TO EFREE DO PUT-IN-D(IE[J], XE[J], YE[J], ZE[J]);
            END;
        POLY-SIDES[TEST] ← SIDES;
        IF STEST < 0 THEN
            BEGIN
                POLY-FRONT[TEST] ← POLY-BACK[ROOT];
                POLY-BACK[ROOT] ← TEST;
            END
        ELSE
            BEGIN
                POLY-FRONT[TEST] ← POLY-FRONT[ROOT];
                POLY-FRONT[ROOT] ← TEST;
            END;
        RETURN;
    END;
```

The comparison algorithms use the following utility routines to characterize the sign of a number and to save a vertex in the D buffer or E buffer.

9.13 Algorithm SIGNOF(X) Returns −1, 0, or 1 to indicate the sign of X

Argument	X a value from which the sign is to be extracted
Constant	ROUNDOFF some small number greater than any round-off error

```
BEGIN
    SIGNOF ← 0;
    IF X < -ROUNDOFF THEN SIGNOF ← -1;
    IF X > ROUNDOFF THEN SIGNOF ← 1;
    RETURN;
END;
```

9.14 Algorithm PUT-IN-D(OP, X, Y, Z) Saves an instruction in the D buffer

Arguments	OP, X, Y, Z a display-file instruction
Global	ID, XD, YD, ZD the D-buffer arrays
	DFREE the next free D-buffer cell

```
BEGIN
    ID[DFREE] ← OP;
    XD[DFREE] ← X;
    YD[DFREE] ← Y;
    ZD[DFREE] ← Z;
    DFREE ← DFREE + 1;
    RETURN;
END;
```

9.15 Algorithm PUT-IN-E(OP, X, Y, Z) Saves an instruction in the E buffer

Arguments	OP, X, Y, Z a display-file instruction
Global	IE, XE, YE, ZE the E-buffer arrays
	EFREE the next free E-buffer cell

```
BEGIN
    IE[EFREE] ← OP;
    XE[EFREE] ← X;
    YE[EFREE] ← Y;
    ZE[EFREE] ← Z;
    EFREE ← EFREE + 1;
    RETURN;
END;
```

Once we have the sorted tree of polygons, we can traverse it to enumerate them in back-to-front order. This is an in-order traversal of the tree which is most simply expressed as the following recursive algorithm.

9.16 Algorithm SAVE-POLYGONS-IN-ORDER(ROOT) Enter polygons into the display file in back-to-front order

Argument	ROOT the root polygon of the spatially sorted tree
Global	POLY-FRONT, POLY-BACK arrays of the polygon table for the branch links

```
BEGIN
    IF ROOT = 0 THEN RETURN;
    SAVE-POLYGONS-IN-ORDER(POLY-BACK[ROOT]);
    SEND-TO-DF(ROOT);
    SAVE-POLYGONS-IN-ORDER(POLY-FRONT[ROOT]);
    RETURN;
END;
```

The algorithm first checks that there is something in the tree. If so, it recursively enters into the display file (in back-to-front order) all of the polygons behind the ROOT polygon. It next enters the ROOT polygon into the display file. Finally, it enters all of the polygons in front of the ROOT polygon by means of another recursive call.

In SEND-TO-DF we first make sure that the proper edge and fill styles are in effect. Then we enter the polygon-drawing command. Finally, we enter the polygon sides.

9.17 Algorithm SEND-TO-DF(POLYGON) Enters a polygon into the display file

Argument	POLYGON the polygon to be entered
Global	POLY-START, POLY-SIDES arrays that are part of the polygon table
	ID, XD, YD arrays that make up the D buffer
Local	I for stepping through the polygon vertices
	F the final vertex of the polygon

```
BEGIN
    CHECK-STYLE(POLYGON);
    F ← POLY-START[POLYGON] + POLY-SIDES[POLYGON] − 1;
    enter the polygon instruction
    VIEWING-TRANSFORM(POLY-SIDES[POLYGON], XD[F], YD[F]);
    enter the polygon sides
    FOR I = POLY-START[POLYGON] TO F DO
        VIEWING-TRANSFORM(ID[I], XD[I], YD[I]);
    RETURN;
END;
```

Checking for the proper edge and fill style is done through the CHECK-STYLE utility.

9.18 Algorithm CHECK-STYLE(POLYGON) Makes sure that the polygon is drawn in the correct style

Argument	POLYGON the polygon to be drawn
Global	CURRENT-FILL-STYLE the interior style of the polygon
	CURRENT-LINE-STYLE the edge style of the polygon
	POLY-FILL-STYLE, POLY-EDGE-STYLE the style entries in the polygon table

```
BEGIN
    IF CURRENT-LINE-STYLE ≠ POLY-EDGE-STYLE[POLYGON] THEN
        BEGIN
            DISPLAY-FILE-ENTER(POLY-EDGE-STYLE[POLYGON]);
            CURRENT-LINE-STYLE ← POLY-EDGE-STYLE[POLYGON];
        END;
    IF CURRENT-FILL-STYLE ≠ POLY-FILL-STYLE[POLYGON] THEN
        BEGIN
            DISPLAY-FILE-ENTER(POLY-FILL-STYLE[POLYGON]);
            CURRENT-FILL-STYLE ← POLY-FILL-STYLE[POLYGON];
        END;
    RETURN;
END;
```

All that remains for our hidden-surface check is an initialization procedure. We use it to set the default edge and fill styles, and we empty the D buffer and polygon table.

9.19 Algorithm INITIALIZE-9 Initialization for hidden-surface routines

Global	CURRENT-FILL-STYLE the interior style of the polygon
	CURRENT-LINE-STYLE the edge style of the polygon
	DFREE the next free D-buffer cell
	POLY index of the next free polygon-table cell

```
BEGIN
    INITIALIZE-9A;
    CURRENT-LINE-STYLE ← 0;
    CURRENT-FILL-STYLE ← -16;
    POLY ← 1;
    DFREE ← 1;
    RETURN;
END;
```

This completes the algorithms needed for hidden-surface removal. Our graphics system does not take full advantage of the power of the binary space partition. With each change of viewing parameters we require the applications program to redraw the scene, constructing fresh display-file segments and causing us to resort the polygons. But note that if only the view changes, and not the objects in the scene, then the sort is essentially the same each time. Different views may change whether a sublist of polygons is in front or in back of the root polygon, but it will not change the way that the polygons were divided into sublists. For some applications which show different views of a fixed world, we can take advantage of this, doing the sort only once, building a single tree. This tree is then traversed in the appropriate order for each scene. To do this, the sorted tree and hidden-surface removal must be incorporated into the applications program.

AN APPLICATION

A major application of computer graphics is in computer-aided design (CAD). The computer can aid in the design process in many areas from architecture to machine parts to electronic circuits. The machine can maintain a data base describing the object being designed. Design components and properties of mating parts may also be available. A model for the part can be constructed and its behavior studied via computer simulation. Of course, it is often quite helpful to be able to draw and to view the part, and this is where computer graphics comes in.

Consider a computer-aided design system for designing machine parts. Such a system might provide a set of primitive shapes and operations for combining them to produce the desired object. But a complex object, if shown in wire frame, may be too confusing. To make a realistic-looking object we should remove hidden lines and surfaces. We should also be able to view the object from different directions so as to inspect all sides of it. We might even employ clipping planes to inspect the interior. To do this on our system we would first set the viewing and clipping parameter for the desired view. We would next turn on the HIDDEN flag and open a display-file segment. We would then draw the part using POLYGON-ABS-3 and POLYGON-REL-3 calls. Finally, we would close and display the segment. The hidden-surface software will produce a realistic image.

FURTHER READING

The first solution to the hidden-line problem was described in [ROB63]. Other hidden-line solutions are described in [APP67], [GAL69], and [LOU70]. Back-face removal is discussed in [LOU70]. The Z buffer is described in [CAT74]. A scan-line Z buffer is suggested in [CAR76]. Other scan-line algorithms are described in [BOU70], [GEA77], and [WYL67]. Some of the hidden-surface algorithms first decompose the polygons into triangles or trapezoids. This ensures that the objects are convex and provides a uniformity which makes them easier to manipulate. Methods for this decomposition are described in [JAC80] and [LIT79]. Sorting into priority order and the painter's algorithm are described in [NEW72]. Warnock's algorithm may be found in [WAR69]. Binary space partition is presented in [FUC80]. Its use for fast generation of different views of a scene is described in [FUC83]. An excellent description and comparison of hidden-surface techniques are found in [SUT74], with a shorter version