

14.2.6 PERFORMANCE

The Roberts technique of removing hidden lines requires large quantities of computation. Roberts gives the figure of one second per object up to thirty objects. Above thirty, as one might expect, the time increases rapidly — 90 seconds for forty objects, 22 minutes for 200. Consequently, despite the elegance of the method, it is somewhat uneconomical as a means of presenting three-dimensional information.

The computation for object-object comparisons grows as the square of the number of objects potentially visible. This behavior is a consequence of comparing each object to the plane faces of all other objects. If there are N objects, the number of such computations is proportional to

$$N(N - 1) = N^2 - N$$

Thus the algorithm is extremely slow for complicated scenes.

14.3 WARNOCK ALGORITHM

John Warnock developed the idea of examining portions of the display screen for visible features rather than examining each feature to see if it is visible. This approach catalyzed further development of hidden-line and hidden-surface algorithms.

The key operation in the Warnock algorithm is determining, for a portion of the display screen, if anything interesting appears there. If nothing appears in this window area, then that portion of the screen need not be considered further: it is blank. If a feature (e.g. line, surface, vertex) appears in the window and is *simple enough* to display directly, the algorithm generates the display. However, the collection of features appearing within the window may be too complicated to analyze and display directly. In this case, the algorithm announces that it has *failed* to process the window. The examination of features in a window, then, has three possible results:

1. No features are visible in the window.
2. A display is generated because the feature or features in the window are classified as simple.
3. The algorithm fails because the features in the window are too complex to analyze.

NORMAN RAUSE

Fig.

Floyd calls such procedures *non-deterministic*, because announce failure if they cannot succeed in performing their [92].

The examination of windows is supervised by a *controller* (1) make sure that all possible windows on the display are examined and (2) cope with the *failure* of the procedure for a window.

If the procedure for examining a window fails, the controller divides the window into several smaller windows, and examines each in turn. This process is applied recursively until the window is smaller than a resolvable spot on the display screen. If the window is still too complicated for the algorithm to display directly, the controller calls for a dot to be displayed at a default position. Because the resolution of the screen is only 1024 by 1024, 10 binary subdivisions of the window size will suffice to reach the finest level of resolution.

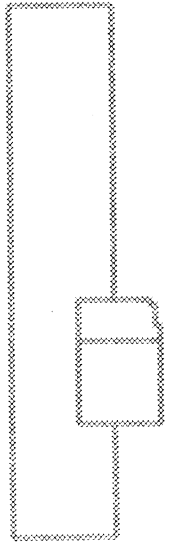
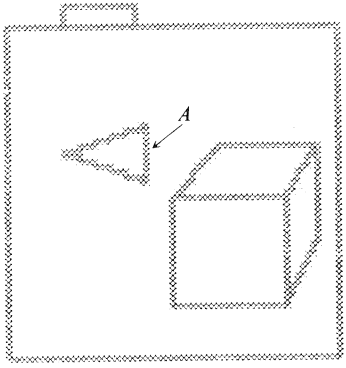
Figure 14-8a shows a scene with hidden-line elimination produced by the Warnock algorithm. Each small 'x' represents a dot displayed on the screen. Figure 14-8b shows the same scene, with no hidden lines removed. Figure 14-8c shows each of the square windows examined by the algorithm in the process of computing which dots to display. These windows are subdivided only near visible features: the window marked *A* actually has an edge of a polygon passing through it (compare Figure 14-8b), but the algorithm has determined that this edge is not visible in the window because a nearer surface completely obscures it.

The choice of criteria used to decide if information in a window is simple enough to display directly affects the number of subdivisions required to produce a display. Figure 14-8c is produced with a procedure which *never* finds information simple enough. The procedure used to produce Figure 14-8d, on the other hand, is designed to detect certain cases in which only one polygon is visible in a window. For example, the line marked *B* was determined to be simple enough to display directly. The same line in Figure 14-8c was not found to be simple enough, the window was subdivided, and dots were eventually displayed when the window size reached the resolution of the screen.

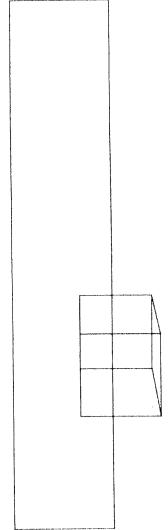
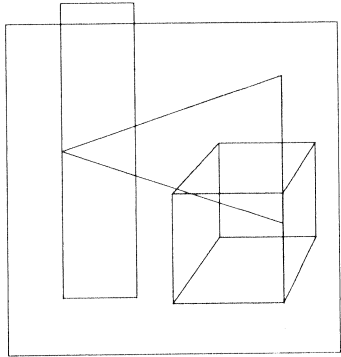
The subdivision process can be viewed as a scheme for resolving the failure to examine large windows. Alternatively, it can be viewed as a process of selectively generating subgoals from the main goal of the hidden-line problem for the scene. If a goal proves too difficult to achieve, it is replaced by a set of subgoals.



Fig.

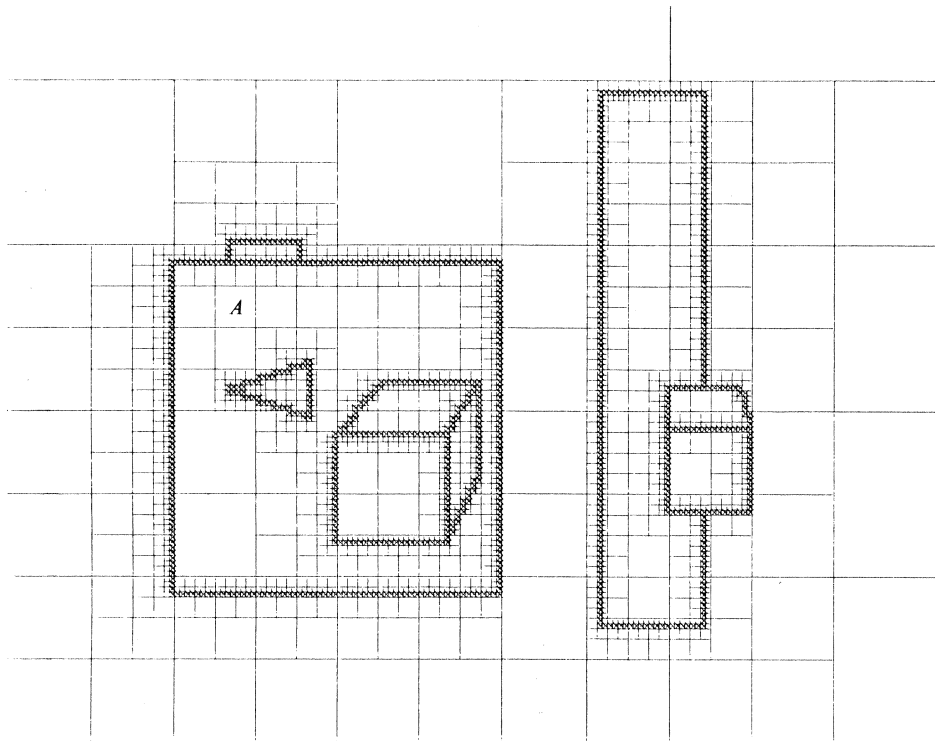


(a)

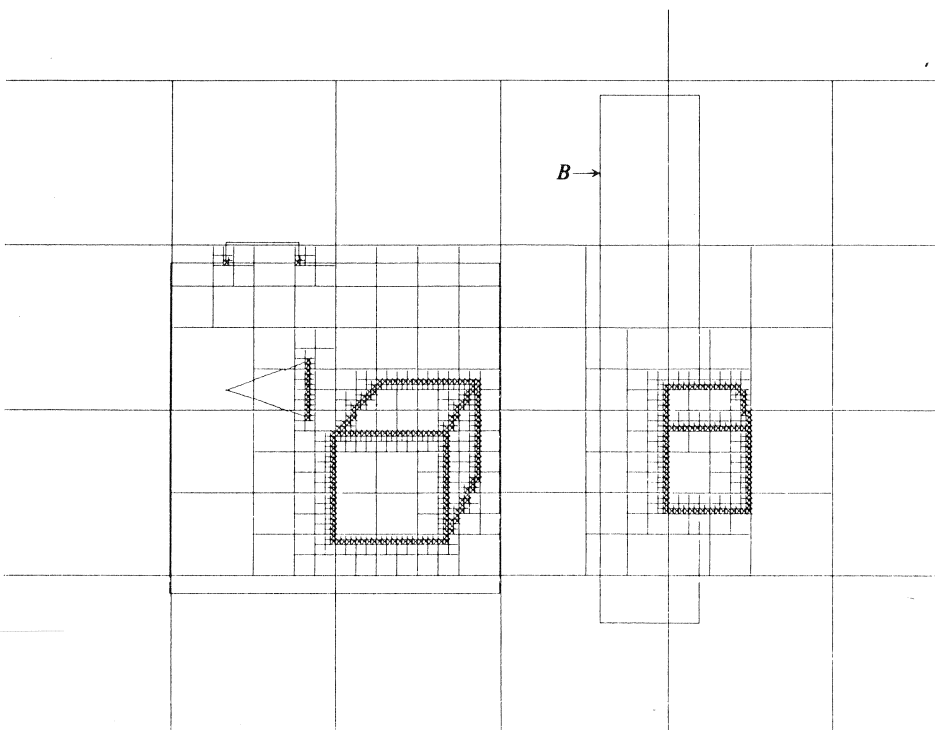


(b)

FIGURE 14-8



(c)



B →

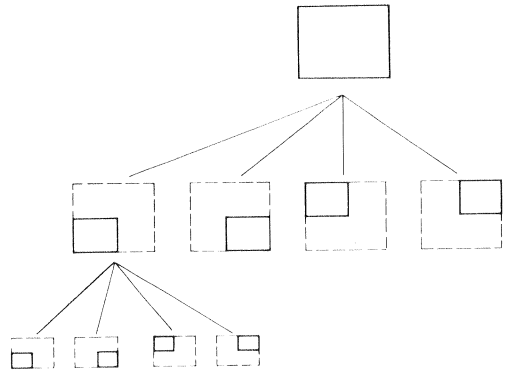


FIGURE 14-9

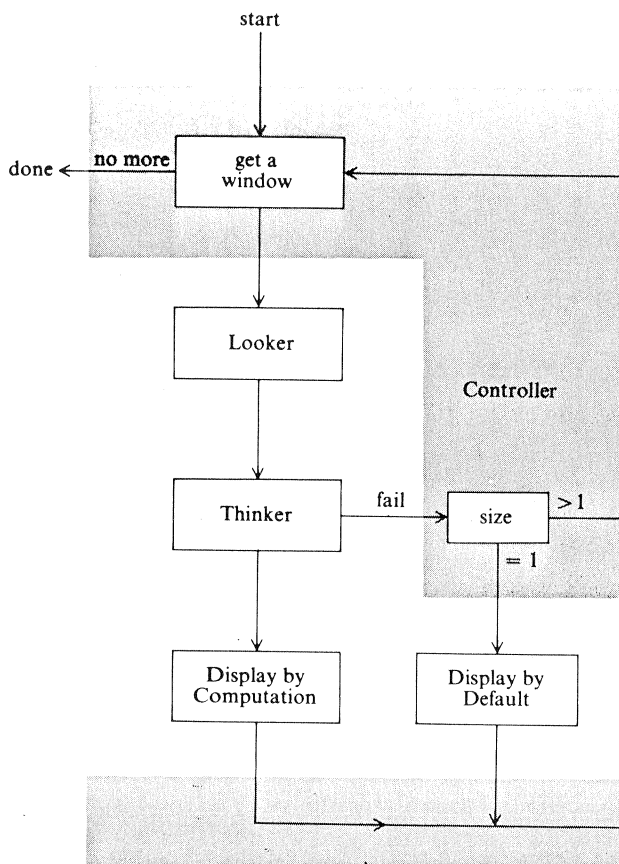
reach directly, the controller generates four subgoals whose solution is equivalent to the first goal (see Figure 14-9).

This view of the algorithm also suggests that whenever the attempt to achieve a goal fails, the algorithm calls itself recursively four times — once for each new subgoal. The tree picture of the subdivision process also suggests a terminology for describing the process. Subdivisions of a window are called *descendants* of the window and the larger window is called their *ancestor*.

We can characterize Warnock's algorithm by its five major components, discussed in subsequent sections: the Looker, the Thinker, Display by Computation, Display by Default, and the Controller. These five parts are shown in Figure 14-10. The Looker examines a particular window and determines what parts, if any, of the objects in the scene are visible in that portion of the screen. The Looker collects data about all potentially visible objects for subsequent use by the Thinker.

The Thinker uses the data collected by the Looker to determine if the features in this window can be displayed directly. If the Thinker is able to display the data presented to it by the Looker, it calls for Display by Computation. If the Thinker finds the situation described by the Looker too complicated, it will announce its failure, whereupon the Controller will subdivide the window or call for Display by Default.

The Controller system handles subdivisions of the windows examined by the Looker and maintains a list of unexamined windows. If the Thinker fails to provide answers and the window is so small that it covers only one resolution unit on the display, then Control will call for Display by Default which will result in a single dot on the screen.



FI

This processing technique is quite general. It could be applied to curved surfaces and to regions of the screen of any desired size. A strategy of subdividing the window to produce simpler surfaces is sufficient to solve the hidden-line problem. The algorithm can be implemented with a trivial Looker and Thinker or with quite complicated versions. Simple versions make it quite easy to program the Warnock algorithm.

14.3.1 SPECIALIZATION OF THE ALGORITHM

We shall describe a particular algorithm within the general framework outlined above. The Controller, shown in Figure 14-11, is designed to process square windows. Subdivision, when it takes place,

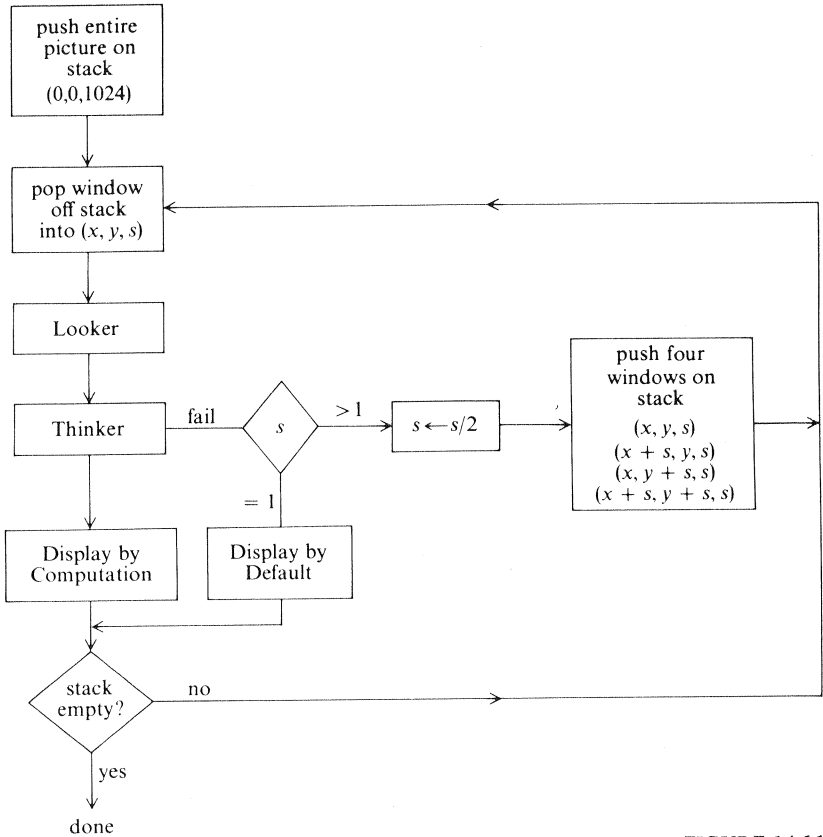


FIGURE 14-11

window into four square windows, each with one-half the length of the side of the original. This Controller does a prefix walk of the tree represented by the subdivided windows. The Controller keeps in a push-down stack the windows that have not yet been examined. If the Thinker fails on a window, the Controller subdivides the window into four and pushes these squares onto the stack. It repeatedly pops windows off the stack and processes them until the stack is empty.

We shall assume that objects are plane-faced polyhedra; each face of an object will be bounded by a polygon. There are many assumptions that we might make about the shape of this polygon. For instance, we could confine our attention to triangles or to polygons with four or fewer sides. We might insist that the polygons be convex, because it is relatively easy to determine if a convex polygon is outside a particular

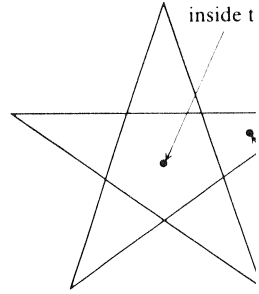


FIGURE 14-12

window. We might allow non-convex polygons, or we might permit polygons to overlap themselves in complicated ways as in Figure 14-12.

The version of the algorithm described here will assume that polygons are planar, that they have an arbitrary number of sides, and that they may be represented as an ordered list of vertices. Coordinates of vertices are stored in screen coordinates, as in Equation 14-2.

14.3.2 THE LOOKER

The Looker compares a polygon taken from the data representing the scene with a window generated by the Controller. The size and position of the window are specified in screen coordinates. The X , Y and Z coordinates of the vertices of the polygon are spatially related to the window in the X - Y plane in several ways, as shown in Figure 14-13.

For each polygon, the Looker decides which of these cases it falls into. It may suffice to detect three cases: surrounding (a), disjoint (b), and intersector (c,d,e).

The information calculated by the Looker is crucial for the elimination of hidden lines. A surrounding polygon clearly has features farther from the eye than the surrounding surface (see Figure 14-14a).

The Looker considers all polygons of all objects. An essential part of the algorithm is that the list of polygons is sorted in an order such that the polygon whose nearest corner is closest to the eye appears first in the list, and the polygon with the farthest away nearest corner appears last. The value of Z_s at the nearest vertex of the polygon is calculated. Whenever the Looker encounters a surrounding polygon, it reorders the list so that the farthest away point of the polygon in the window as

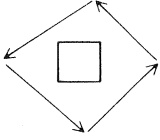
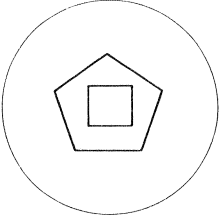
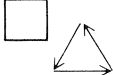
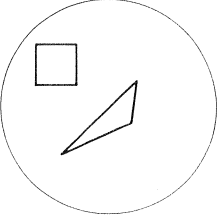
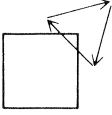
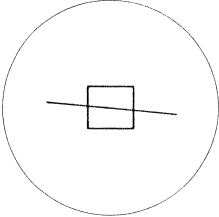
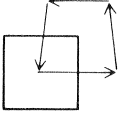
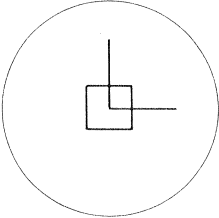
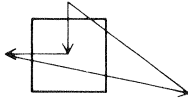
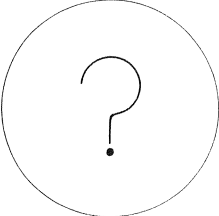
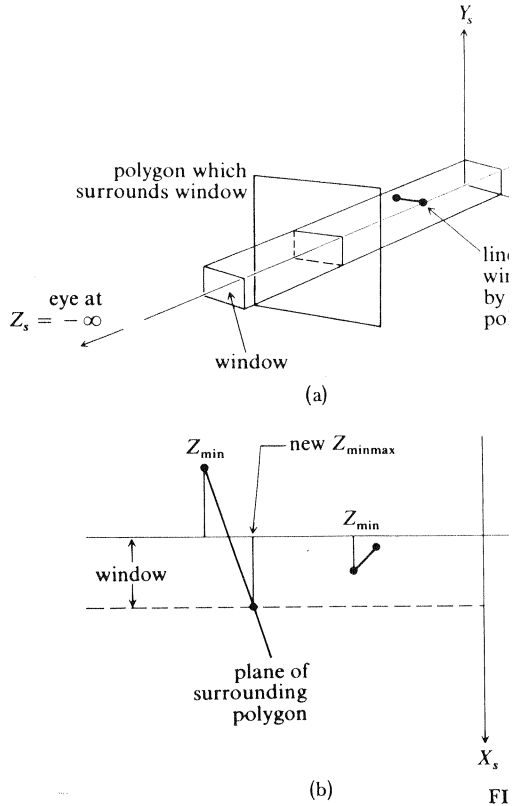
Name	Drawing	Symbol	Fig
(a) Surrounder			L
(b) Clean miss (disjoint)			
(c) Single line intersection			
(d) Single vertex included			
(e) other, more complicated			

FIGURE 14-13



(Figure 14-14b). When considering another polygon in the Z_{min} is greater than Z_{minmax} , it is clearly hidden by the surround. Thus the search through the ordered list of polygons is prematurely terminated by the discovery of a surrounder.

A great deal of computation can be avoided if the Looker uses *ancestral information* (Figure 14-15). For example, if a polygon surrounds a window, it clearly surrounds all subdivisions of that window. There is no point then in examining a polygon to see if it surrounds a window if it was known to surround some ancestor window. When the Looker decides that a polygon surrounds a window, the fact is recorded so that the computation need not be repeated for descendants of that window. It is also important to record that a polygon is disjoint from a window, because it will be disjoint from all descendants of the window. Data is stored in the polygon

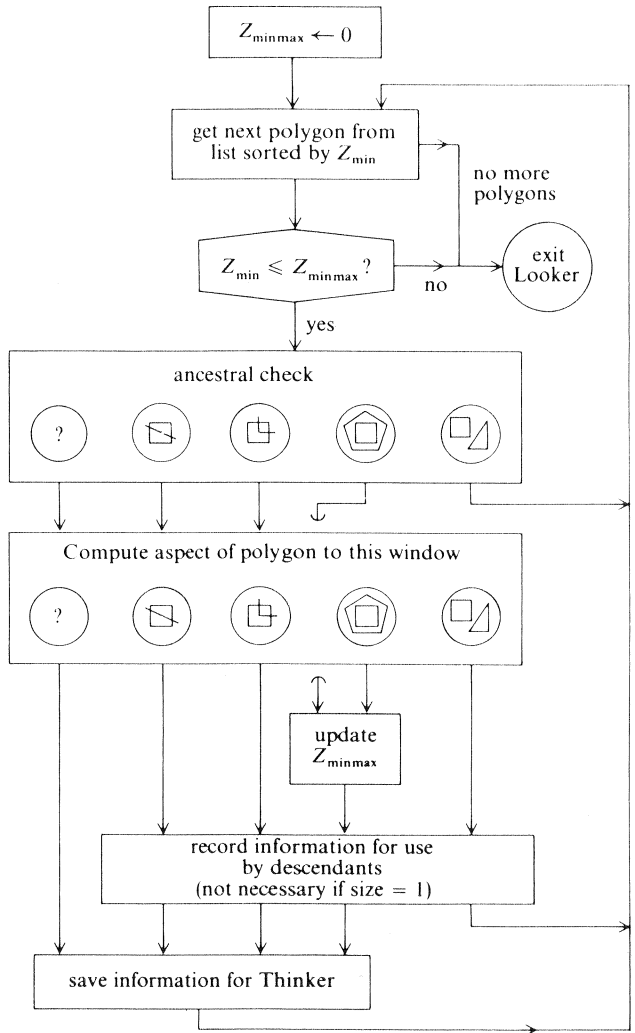


FIGURE 14-15

indicate whether polygons are known to be surrounders or known to be outside certain windows.

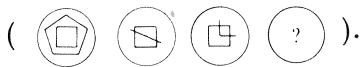
In addition, we might save data indicating that a polygon has only one edge in a window and no vertex, or that it has only one vertex in the window and no complete edges, or other data of a similar sort. Keeping records of such data can reduce the number of edges of the

polygon which need to be considered in determining whether a polygon is of interest in subdivisions of the window.

The first thing the Looker does with a polygon is to do an ancestral check. The Looker retrieves any information known about that polygon, such as that the polygon was entirely outside the ancestor of the present window. In such a case, of course, the polygon need not be considered further. If the polygon surrounds an ancestor of the present window, it obviously surrounds the present window and some computation can be avoided.


If the ancestral check fails to yield any information about a polygon, the Looker must compute the spatial relationship between the polygon and the present window. The Looker needs to know whether the edge of the polygon passes through the window and if not, whether the polygon surrounds the window or is entirely outside the window. The results of these computations are saved for use by the Thinker. If the present window must be subdivided,





The results are also saved for use by the Thinker. In the circular form of Looker and Thinker we give here, the Thinker keeps lists of all surrounding and intersecting polygons (



14.3.3 THE THINKER

The function of the Thinker is to solve the hidden-line problem. The Looker has found no polygons which surround or intersect the window. If the window is clearly the window is blank. Otherwise, the question to be answered

the Thinker is: does there exist a surrounder () which

other surrounders and intersectors (   )

surround the window?

A simple way of answering this question is to compare the corner values of the planes of the polygons at the four corners of the window.

under consideration. If the depth of a surrounder polygon is less than the depths of all other polygons at the corners of the window, then that surrounder indeed hides all other possible features in the window. If we are producing a line-drawing (Figure 14-8a), the window is blank; if we are making a shaded image, we display a shade appropriate to the surrounder polygon surface.

The condition that the surrounder-polygon depth be less than the depths of other polygons at the corners of the window in order to hide the other polygons is sufficient but not necessary. The reason is that, for the purpose of the depth comparison, we are extending the planes of intersector polygons to cover the entire window. If the extended polygon is hidden, so will be the actual polygon. However, a surrounder might hide an intersector but not its extension.

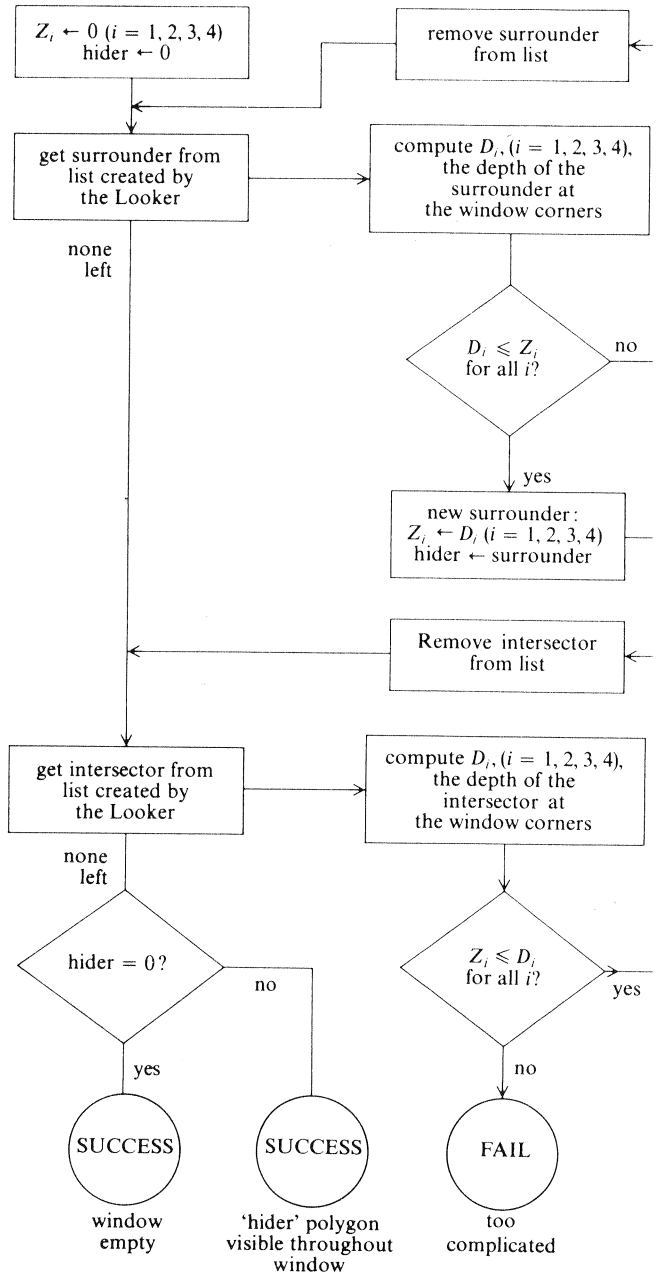
If the depth tests fail to yield a surrounder nearer the eye than all other polygons, the Thinker announces that the situation is too complex to analyze, and the control will suitably subdivide the window.

This simple operation of the Thinker is adequate to solve the hidden-line problem (Figure 14-16). However, many subdivisions can be avoided if we design a slightly more complicated Thinker (compare Figure 14-8c and 14-8d). The more complex the Thinker, the fewer the subdivisions required. However, a complex Thinker might slow the algorithm more than might a few more subdivisions.

The first useful extension to the Thinker is one that enables it to detect the case of 0 surrounders of a window and exactly 1 intersector polygon. If we are generating an outline drawing, clearly every edge or portion of an edge of the intersector which falls within the window should be Displayed by Computation. In this case Display by Computation clips the edges of the intersector polygon against the window and displays any visible lines or portions of lines.

Another extension is detecting the case of a bonafide surrounder (the case where $hider = 0$ in Figure 14-16) and only one intersector which is not hidden and which lies entirely in front of the surrounder, as shown in Figure 14-17. The solid dots represent depth computations, used to establish that the plane of the intersector polygon lies closer to the eye than the surrounder polygon.

Another extension to the Thinker will process intersecting surfaces correctly. If two polygons intersect, we may desire to show the *implied edge* which appears at the intersection. The line labeled *A* in Figure 14-8a is such an edge; the tip of the triangle penetrates the square. The



FIG

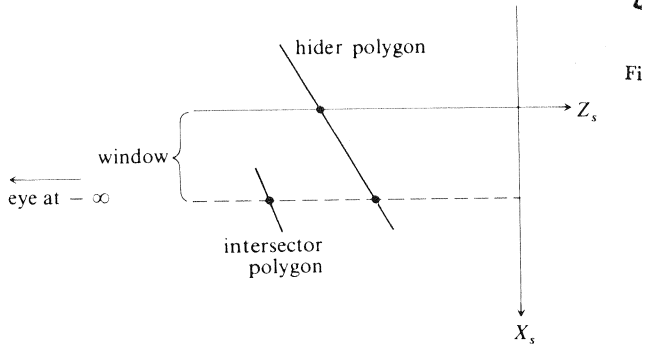


FIGURE 14-17

Thinker of Figure 14-16 must be augmented to check for two surrunders which may *penetrate* each other within the window, as shown in Figure 14-18. The actual display of the implied edge results because windows with penetrating surrunders cause failure of the Thinker and hence cause subdivision. Eventually, the window size reaches 1, and a dot is Displayed by Default.

14.3.4 PERFORMANCE OF THE ALGORITHM

The computation time consumed by the Warnock algorithm is roughly proportional to the complexity of the final display and not proportional to the complexity of the scene. The amount of computation can be gauged by the number of subdivisions required. Subdivisions always result in a displayable feature somewhere within the window being subdivided; therefore computation time is proportional to *visible* complexity. The decision procedure used in the Looker and Thinker can speed processing of various classes of images: we have already shown that the Looker required to process penetrating polygons is more complex than the simple Looker. An evaluation of the performance of several decision procedures is given in [177].

If a shaded display is required, small modifications to the algorithm are necessary. When the Thinker finds a surrunder which hides all other features in the window, only one surface is visible throughout the window, namely the surface of the surrunder. This is enough information to determine the shading intensity for the entire window. Additional logic is needed to compute appropriate intensities for the dots generated when the window size is reduced to one resolution unit.

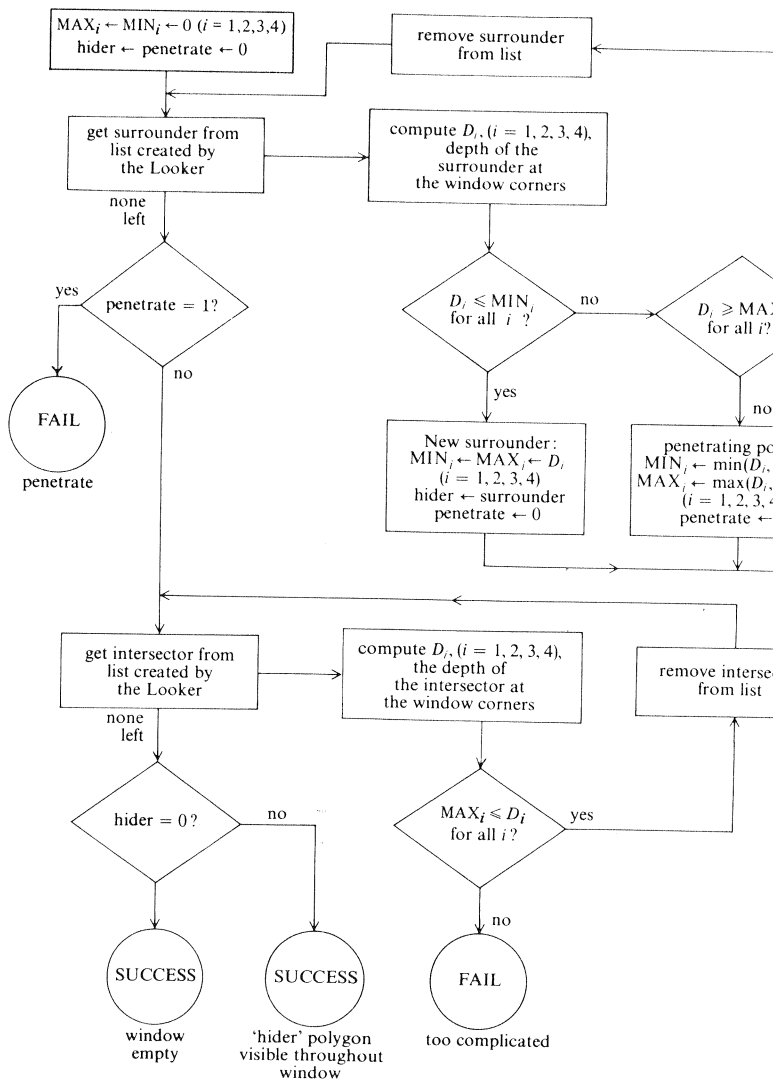


FIG1

The output of the algorithm is not convenient for raster displays, because windows are examined according to the goal-sense nature of the algorithm, and not according to ascending or descending Y_s coordinate. An interface between the Warnock algorithm and raster display hardware has been designed; it demonstrates ingenious hardware to drive video displays [47].

14.4 SCAN-LINE ALGORITHMS

Exceptionally realistic pictures of solid objects can be generated by using a raster-scan display such as a television monitor. Generating these pictures requires techniques for removing hidden surfaces and for shading visible surfaces. The principal technique amongst these is the *scan-line algorithm* for hidden-surface elimination: an algorithm that generates a shaded picture on a line-by-line basis, ready for display on a television monitor. Several such algorithms have been developed, making use of some of the techniques used in the earlier hidden-line algorithms. In particular, they use the concept of generating a picture by treating each region of the screen in turn rather than each element of the object; and they use non-deterministic methods in a controlled fashion to resolve complex situations. In addition, two properties of raster-scan images are exploited to increase the efficiency of scan-line algorithms: *scan-line coherence* and geometrical simplification of the three-dimensional space into a two-dimensional space for making decisions about hidden surfaces.

Scan-line coherence is a property of scan-line displays of most scenes: that is, adjacent scan-lines appear very similar. The algorithm takes advantage of the similarities to reduce the computation required for each scan-line to an *incremental* calculation: information saved when processing one scan line is used to speed processing of the next one. The efficiency achieved by scan-line coherence is somewhat analogous to the ancestral checks of the Warnock algorithm and to the advantage the Warnock algorithm achieves from processing blank, uninteresting areas of the screen very rapidly, and concentrating only on those portions where detail is visible.

The geometrical simplification which aids the scan-line algorithms results from the particular choice of *windows* examined by the algorithm: the windows are one scan-line high and span the width of the screen (see Figure 14-19). As in the Warnock algorithm, the windows are positioned in the screen coordinate space. The windows are processed consecutively by ascending or descending Y_s coordinate, just as a raster-scan display might show the scan-lines represented by the windows. Furthermore, within each window the algorithms proceed in a strictly left-to-right manner. The use of a top-to-bottom, left-to-right window-processing strategy insures that display data are generated in the same order as required by the raster scanning hardware.



Fig.

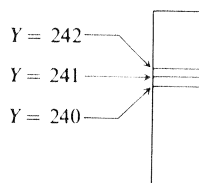
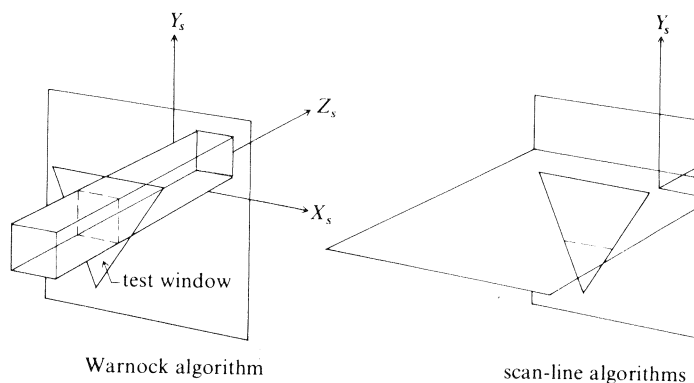


FIGURE 14-19



FI

The geometrical simplification occurs when a planar polygon screen coordinate system is intersected with a scan-line window shown in Figure 14-20. The intersection is a *line* in the Y_s plane corresponding intersection with a window of the Warnock algorithm a *polygon* in three-dimensional screen coordinate space.

The scan-line algorithm must decide what polygons are visible in the scan-line window, and these decisions are all made by comparing *segments* in the X_s - Z_s plane. The decisions are substantially simpler than those of the Warnock algorithm, which requires comparing entire polygons.

The intersection of the scan-line window and a planar polygon is a collection of line segments. Figure 14-21 shows what segments look like in the X_s - Y_s plane. On a given scan line, a polygon is visible in terms of its segments. The polygon intersects the scan-line window at $Y_s = \alpha$ with one segment. The segment is described by the X_s coordinates of the edges of the polygon which bound the segment. For example, at $Y_s = \alpha$, the segment is bounded by the edges AD and BC . The X_s coordinates for the left and right edges of the segment are simple linear functions of Y_s , i.e. the edge equation is $X_s = a + bY_s$.

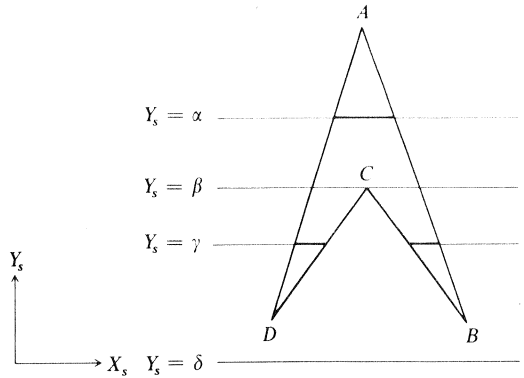


FIGURE 14-21

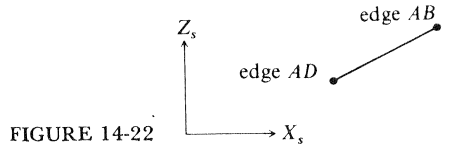


FIGURE 14-22

The polygon of Figure 14-21 viewed from above when $Y_s = \alpha$ is shown in Figure 14-22. The Z_s coordinates of the left and right ends of the segment are just the Z_s coordinates of the edges AD and AB at $Y_s = \alpha$. These coordinates are also simple linear functions of Y_s , i.e. $Z_s = c Y_s + d$.

At $Y_s = \beta$, the single segment becomes two segments because two new edges, CD and CB enter in the window at $Y_s = \beta$. Finally, at $Y_s = \delta$, no segments of this polygon remain.

14.4.1 PROCESSING A SCAN-LINE

The hidden-line problem is reduced to deciding, for each scan line, which segments or portions of segments should be displayed. In Figure 14-23, on scan line $Y_s = k$, there are two segments, as shown by the arrows.

This same scan line is drawn differently in Figure 14-24, using the plane of the paper to represent the $Y_s = k$ plane. We can see the depth relationships of the two segments, and also see which parts are hidden by other parts. But the situation could become quite complicated, as shown in Figure 14-25. In this case, the non-deterministic procedure

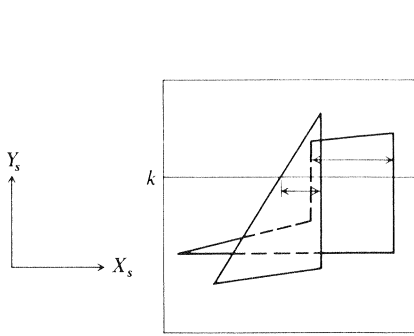


FIGURE 14-23

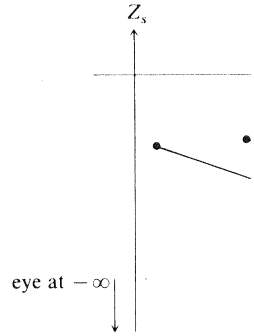


FIGURE 14-24

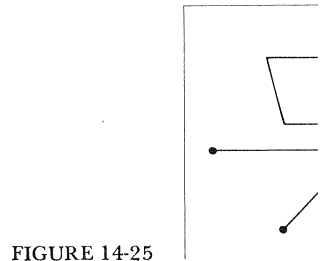


FIGURE 14-25

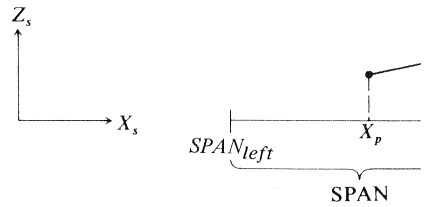


FIGURE 14-26

used to process scan lines announces *failure*. The width of the span is divided into smaller sections, or *sample spans*. Each span is defined by its left and right ends, in screen coordinates: $SPAN_{left}$ and $SPAN_{right}$. The same procedure is then applied to these spans.

We can detect several simple cases:

1. Only one segment is in the span (see Figure 14-26). This segment is clearly visible in the region $X_p \leq X_s \leq SPAN_{right}$. There are actually four similar cases, which are shown in Figure 14-27. Their handling is obvious.

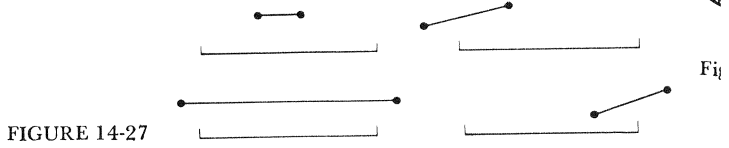


FIGURE 14-27

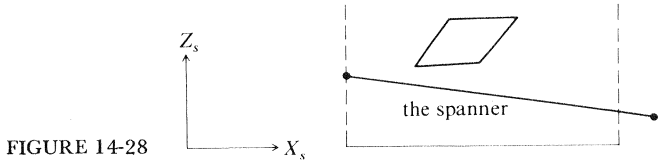


FIGURE 14-28

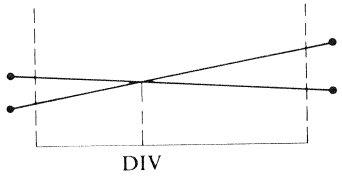


FIGURE 14-29

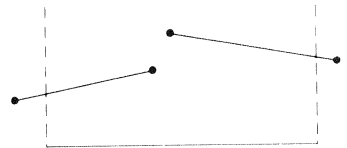
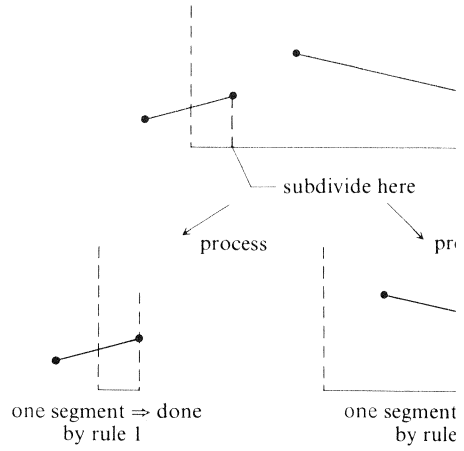


FIGURE 14-30

2. A *spanner* which hides all other segments. A spanner is defined as a segment which extends to or beyond the edges of the sample span (see Figure 14-28). The spanner segment is everywhere nearer the eye than any other segment. Hence it hides the other segments, and is visible in the region $SPAN_{left} \leq X_s \leq SPAN_{right}$.
3. Simple intersection. If only two segments fall inside the span, and they are both spanners, we may have the kind of intersection shown in Figure 14-29. In this case, we can compute the X_s coordinate of the point of intersection. One segment is visible for $SPAN_{left} \leq X_s \leq DIV$, the other for $DIV \leq X_s \leq SPAN_{right}$.
4. Complicated cases. The remainder of cases are considered complicated. Figure 14-30 shows an example. We must subdivide the sample span at some point and try the test procedures again. However, we do not recursively subdivide but instead ask: 'What is the left-most segment endpoint in the span?' and subdivide at that point. If there is no segment endpoint in the span, we divide the span at its midpoint by default.

The reason for dividing at the left-most endpoint is that this hastens our ability to resolve the complicated case. A simple



1

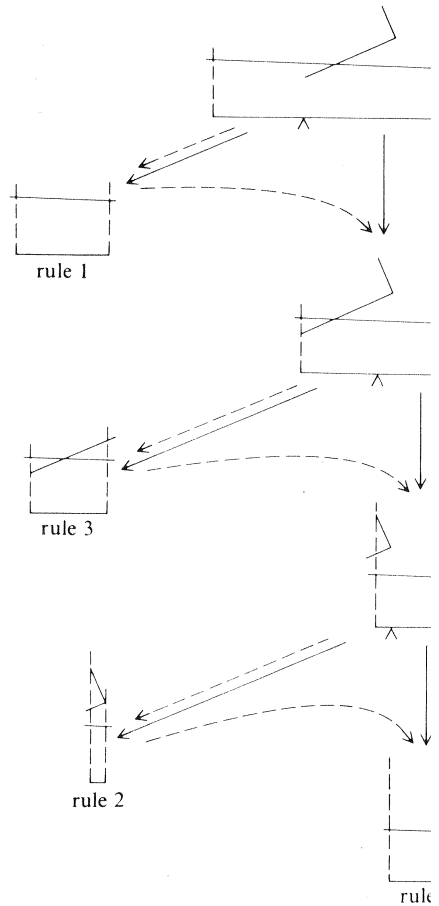
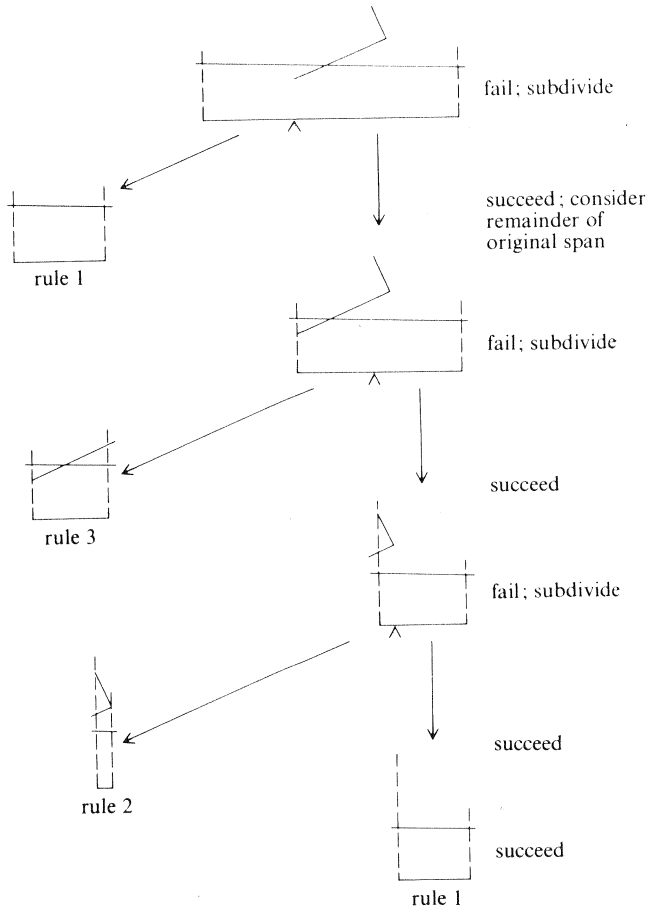


FIGURE 14-32



Fig

FIGURE 14-33

subdivision is shown in Figure 14-31. The example of Figure 14-32 is more complicated. This subdivision process gives a tree of divisions needed to decide on the shading for the original span.

Another subdivision scheme might find the division point, process the left sample-span and then try to process the entire remainder of the original span. The processing procedure is then non-recursive, as shown in Figure 14-33.

14.4.2 SCAN-LINE COHERENCE

The process we have detailed will generate the display for one scan-line from a description of segments on that scan-line. The process can be repeated for successive scan-lines. However, adjacent scan-lines often

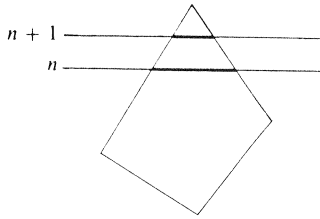
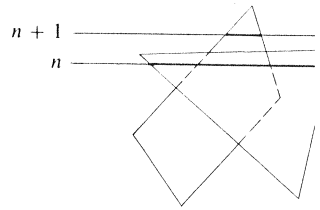


FIGURE 14-34



FI

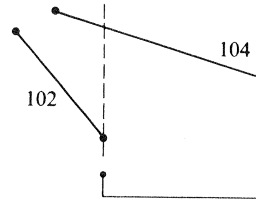


FIGURE 14-36

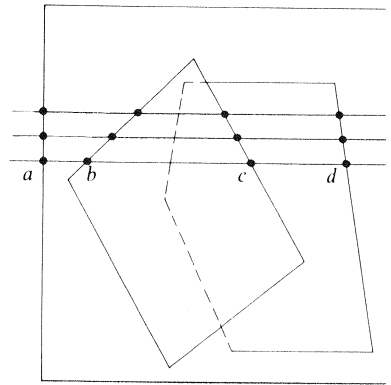


FIGURE 14-37

have very similar displays, as shown in Figure 14-34. Of course, not *always* the case, as demonstrated in Figure 14-35.

Considerable savings in computation time can be made if the algorithm takes advantage of scan-line similarities. A reasonable assumption is that if a certain span from one segment endpoint to another, as shown in Figure 14-36, is a simple case on scan line k , then it will be a simple case on scan line $k+1$. The span will be a *predicted sample span* on scan-line $k+1$. Predicted sample spans are determined by positions of segment edges, and not by particular X_s values. As we move fi

line k to $k+1$, the X_s position of the edge will change, and our predicted sample span must change accordingly. Thus it is convenient to describe the predicted sample spans as 'from the right edge of segment 102 to the right edge of segment 104.' If, at any time, one of these edges exits or becomes hidden, we cease prediction of that span.

Stated differently, we use the record of subdivisions on scan line k to predict fruitful subdivisions for scan line $k+1$.

As an example, the heavy dots in Figure 14-37 divide each scan line into predicted sample spans. The dots are called *predicted sample points*. Thus the four sample spans ab , bc , cd , de will subdivide the entire scan line such that the decisions for each of the four spans are simple cases, i.e. they do not require further subdivision. With excellent scan-line to scan-line coherence, we should rarely need to subdivide a span.

14.4.3 IMPLEMENTATION AND PERFORMANCE

A variety of scan-line algorithms has been created; the discussion above is taken from the algorithm of Watkins [301], which was designed to be implemented in hardware and utilizes the non-determinism, windowing and screen-coordinate concepts of the Warnock algorithm. This algorithm was derived from earlier work by Wylie, Romney, *et al* [318, 238]. Another algorithm, designed by Bouknight [28, 29] uses explicit computation to avoid the non-deterministic behavior of the Warnock algorithm, but does not employ scan-line coherence speedups.

The scan-line algorithm described above is quite fast, although its dependence on complexity of the scene is difficult to analyze. Watkins tabulated the performance of the algorithm for a variety of scenes and discovered that the computation grows roughly as the *visible* complexity increases.

The algorithm can be implemented in software (Appendix VII) or hardware.* The hardware implementation is inexpensive compared to previous hardware techniques [246] and can generate images of quite complicated scenes in real time. By real time we mean that the calculations required to generate display information take no longer than the raster-scan of the frame.

* At the University of Utah, Watkins has built prototype equipment that implements his algorithm.

APPENDIX VI

THE WARNOCK HIDDEN-LINE ALGORITHM

This appendix describes an implementation of the Warnock algorithm which is described briefly in Chapter 14. Specific Looker and Thinker computations are described, followed by a SAIL program implementation.

AVI.1 THE LOOKER

The task of the Looker is to classify a polygon as either (1) a surrounder of the window, (2) disjoint from the window, or (3) an intersector of the current window. In addition, it may compute some details of any intersection (e.g. 1 vertex, 1 free edge, etc).

The computations all closely resemble *clipping*: a set of lines representing the edges of a polygon is clipped against the window (Notice that this is two-dimensional clipping). If any edge of a polygon intersects the window, then the polygon is an intersector. Further information is available from the clipping operation, such as the screen

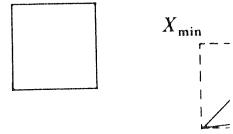


FIGURE AVI-1

coordinates of the clipped edge. Another way to decide whether an edge passes through the window is to substitute the coordinates of the four corners of the window into the line equation of the edge. If the four resulting numbers are the same, then all four corners of the window lie on one side of the line, and the line does not pass through the window.

If no edges of a polygon intersect the window, then the polygon is either disjoint from the window or it surrounds the window. The method for distinguishing these two cases requires a fair amount of computation, and it is often convenient to be able to detect disjoint polygons very rapidly. If any of the following conditions are met, the window and the polygon are disjoint:

1. All vertices of the polygon lie to the right of the window.
2. All vertices of the polygon lie to the left of the window.
3. All vertices of the polygon lie below the window.
4. All vertices of the polygon lie above the window.

This check can be performed quickly if, for each polygon, minimum X , minimum Y , and maximum Y values of the coordinates of all its vertices are stored. These minimum and maximum values can be viewed as defining a rectangle which surrounds the polygon. Comparison of that rectangle and the window quickly tells whether further computation is required (*cf.* boxing), as can be seen in Figure AVI-1. If this simple check fails, it is still possible that the window and polygon are disjoint, but more computation is required to make the decision.

If the line equations for the edges of the polygon determine that all four corners of the window lie on the interior side of all the edges, then the polygon surrounds the square (This requires that the line equations be formulated appropriately). This condition is sufficient but not necessary for concave polygons (see Figure AVI-2).

The method for determining surroundedness which Warnock uses is to draw a line from part of the window to a point known to be outside the polygon.

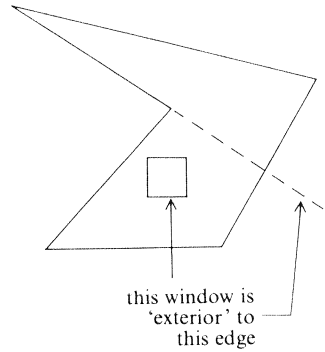
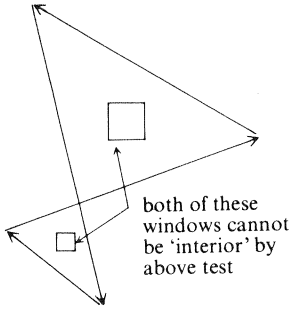


FIGURE AVI-2

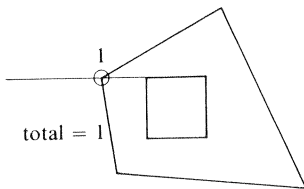
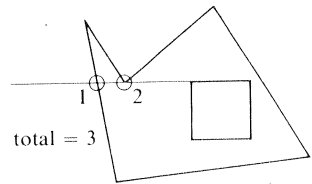
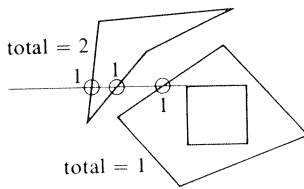


FIGURE AVI-3

outside the polygon. We then count the number of edges of the polygon which intersect the line. If the number is odd, then the polygon surrounds the window; if even, the polygon does not surround the square (disjoint). The only difficulty with this computation occurs when a vertex of the polygon lies directly on the line. In this case, the vertices adjacent to the vertex on the line are considered. If these two vertices are on different sides of the line emanating from the window, then one intersection is recorded, otherwise two (see Figure AVI-3).

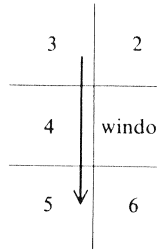
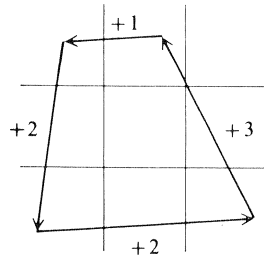
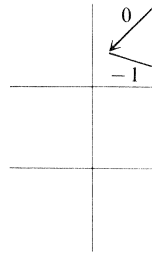


FIGURE AVI-4



total = 8
surrounder



total =
disjoin

FIG 1

Another way to determine whether a polygon surrounds the window is to compute the angle about the window through which each edge passes, and to keep a running total of these incremental angles. As each edge of the polygon is processed in order. If the resulting total angle is zero, then the polygon is outside the window (disjoint). If the total angle is ± 360 degrees, the polygon surrounds the window. If the total angle is ± 720 degrees, the polygon surrounds the window twice and must overlap itself. If the resulting angle is not a multiple of 360 degrees, the angle computation has gone astray. The technique is simplified to consider only 8 regions around the window, as shown in Figure AVI-4. The incremental angle of the directed line segment in region 4 of Figure AVI-4 is +2. The total cumulative angle determines whether the polygon surrounds the window or is disjoint (see Figure AVI-5).

The incremental angle ($\Delta\alpha$) is calculated as follows:

$$\Delta\alpha = (\text{region number of second endpoint}) - (\text{region number of first endpoint})$$

$$\text{if } \Delta\alpha > 4 \text{ then } \Delta\alpha \leftarrow \Delta\alpha - 8;$$

$$\text{if } \Delta\alpha < -4 \text{ then } \Delta\alpha \leftarrow \Delta\alpha + 8;$$

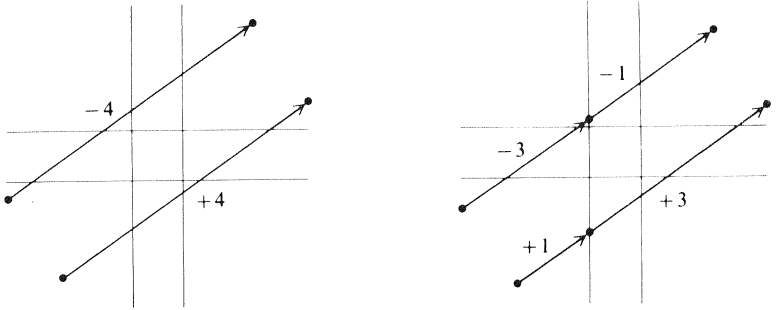


FIGURE AVI-6

One tricky case arises when the incremental angle is ± 4 , as shown in Figure AVI-6. Although these two lines have endpoints in the same regions, they have different incremental angles. The computation of the correct angle can be made by intersecting the line with one of the window boundaries, and then calculating the incremental angle of each part of the two line fragments formed.

This simplified angle measurement can be performed as a side effect of applying a clipping procedure to the edges of the polygon. The clipping operation will yield, for each edge (1) whether it intersects the window at all, (2) if not, the value of the incremental angle traced by the edge. If this information is computed for all edges, the polygon can easily be determined to be (1) a surrounder, (2) a disjoint polygon or (3) an intersector of the window.

AVI.1.1 DEPTH CALCULATIONS

The Looker and Thinker occasionally need to compute the depth (Z_s value) of a polygon at given screen coordinates X_s , Y_s . This computation is performed in the screen coordinate system, and is equivalent to intersecting a ray from the eye through the screen at point X_s , Y_s with the plane of the polygon.

The computation is aided by storing, for each polygon, the equation of the plane of the polygon in screen coordinates:

$$a X_s + b Y_s + c Z_s + d = 0$$

The value of Z_s at any screen position (X_s, Y_s) can then be quickly computed.

AVI.2 THE THINKER

The range of Thinkers suitable for the Warnock algorithm is boundless. Two interesting Thinkers which handle penetration are given below. They use the following symbolic notation: P is the number of penetrators of the window; I is the number of interior polygons of a window, less those whose *planes* are behind the planar surround.

The Thinkers are:

1. Never Display by Computation. The Thinker must distinguish two cases (just as in Figure 14-18):

1. $I \neq 0$ or $P \neq 0 \Rightarrow$ Fail
2. $P = 0$ and $I = 0 \Rightarrow$ Success; window empty

2. Display by Computation if only one interior polygon lies on a (possibly non-existent) surround. The Thinker distinguishes three cases:

1. $I > 1$ or $P \neq 0 \Rightarrow$ Fail
2. $P = 0$ and $I = 0 \Rightarrow$ Success; window empty
3. $I = 1 \Rightarrow$ Success; clip each edge of the interior polygon to the window and display.

Case 3 yields success only if the interior polygon penetrates the plane of a critical surround (hider); otherwise the decision must be to fail.

EXERCISES

1. Suggest various mechanisms for retaining the ancestral information. Examine the methods in the light of the uses of the algorithm: storing when a polygon is examined, retrieving when a window is processed, and saving/restoring information as the controller processes different windows. How does the efficient ancestral data structure interfere with the ordering of the polygons by Z_{min} ?

2. If we restrict the class of surfaces so that no polygons can penetrate each other, what simplifications can be made in the algorithm, excluding those mentioned in Chapter 14?
3. One version of the Warnock algorithm restricts the class of polygons to triangles. If polygons of more than 3 sides are required, they are created from collections of triangles. The motivation for this procedure is that the algorithm will be much more efficient. Give an algorithm for decomposing a collection of polyhedra into a representation consisting only of triangles. Can the Warnock algorithm be made to operate more efficiently on triangles than on arbitrary polygonal faces? What are the drawbacks of this technique? How, if at all, does the subdivision of polygons into triangular regions affect the final line-drawing? What if we are generating a shaded picture?
4. Where does the Warnock algorithm spend its time?


```

*****
BEGIN "WARNOCK ALGORITHM"
COMMENT
*****
                                     D A T A   S T R U C T U R E
*****
THERE ARE THREE KINDS OF OBJECTS IN THE SYSTEM: POINTS, EDGES, AND POLYGONS. DATA ABOUT THESE
OBJECTS ARE RETAINED IN ARRAYS. FOR EXAMPLE, A 'POINT' HAS THREE COORDINATES, HENCE THREE ARRAYS
ARE DECLARED, INDEXED BY THE 'POINT NUMBER', ONE ARRAY FOR EACH OF THE THREE COORDINATES.

POINTS: EACH POINT IS REPRESENTED BY ITS X,Y AND Z COORDINATE IN THE SCREEN COORDINATE SYSTEM.
THESE VALUES ARE STORED IN THE ARRAYS XS, YS AND ZS.

EDGES: EACH EDGE IS REPRESENTED BY THE 'POINT NUMBERS' OF THE TWO ENDPONTS OF THE EDGE. IN
ADDITION A POINTER IS KEPT TO POINT TO THE 'NEXT' EDGE ON THE POLYGON. THE THREE ARRAYS ARE
THUS: EDI1, EDI2, AND EDLINK.

POLYGONS: EACH POLYGON IS REPRESENTED BY A LIST OF EDGES. THIS LIST IS THREADED THROUGH THE EDLINK
CELL OF EACH EDGE, AND A POINTER TO THIS LIST IS RETAINED FOR EACH POLYGON (POLYEDGE).
OTHER DATA ALSO PERTAIN TO POLYGONS: THE FOUR COEFFICIENTS OF THE PLANE EQUATION (POLY1A,
POLY1B, POLY1C, POLY1D), THE ZS VALUE OF THE VERTEX WHICH LIES CLOSEST TO THE EYE
(POLY1ZMIN), AND TWO LISTS: POLYLINK WHICH LINKS ALL POLYGONS ON A LINKED LIST, AND
POLYLIST, WHICH IS USED BY THE LOOKER AND THINKER TO REMEMBER STATUS OF POLYGONS EXAMINED.

THE ARRAY 'HISTORY' IS USED TO KEEP TRACK OF ANCESTRAL RELATIONS. THE ARRAY IS INDEXED BY LEVEL
NUMBER (DEPTH OF RECURSION OF THE MAIN SUBROUTINE) AND BY THE POLYGON NUMBER IN QUESTION. THE ENTRY
IN THE ARRAY IS EITHER:

      -1 ... NO ANCESTRAL INFORMATION. MUST TEST THE POLYGON
       8 ... THE POLYGON IS A SURROUNDER AT THIS LEVEL
       0 ... THE POLYGON IS DISJOINT AT THIS LEVEL

EACH LINE OF THE PROGRAM CONCERNED WITH ANCESTRY IS PRECEDED WITH THE CHARACTERS ##. IF THESE LINES
ARE REMOVED, THE PROGRAM WILL STILL GENERATE THE SAME FINAL PICTURE, BUT THE COMPUTATION TIME
REQUIRED WILL BE GREATER. THIS PARTICULAR IMPLEMENTATION OF THE ANCESTRAL INFORMATION IS VERY
INEFFICIENT, AND IS CHOSEN FOR MAXIMUM CLARITY. A BETTER IMPLEMENTATION IS GIVEN IN WARNOCK'S
PAPER.

DEFINE POLYMAX = "16", EDGEMAX = "63", POINTMAX = "31", ## = "", SAFER = "SAFE";

```

```

REAL WLX,WRX,WBY,WTY,ZMIN,ZMINMAX,Z1,Z2,Z3,Z4,NX,NY,NZ,PX,PY,PZ
      XX1,XX2,YY1,YY2,ZMIN1,ZMIN2,ZMIN3,ZMIN4,ZMAX1,ZMAX2,ZMAX3,ZMAX4;
INTEGER P,SURROUNDERS,INTERSECTORS,POLYPTR,HIDER,PENETRATE,
      NEXTEGE,DELTA,THETA,HOLD,OLDP,J;

```

```

COMMENT

```

```

*****
***** M I S C E L L A N E O U S   P R O C E D U R E S
*****

```

```

INTEGER PROCEDURE GETNEXTEDGE;
BEGIN INTEGER I;

```

```

      IF NEXTEGE = 0 THEN RETURN (0);

```

```

      I ← ED!1[NEXTEGE];

```

```

      PX ← XS[I]; PY ← YS[I]; PZ ← ZS[I];

```

```

      I ← ED!2[NEXTEGE];

```

```

      NX ← XS[I]; NY ← YS[I]; NZ ← ZS[I];

```

```

      NEXTEGE ← ED!LINK [NEXTEGE];

```

```

      RETURN (-1)

```

```

END;

```

```

REAL PROCEDURE GETZ (INTEGER P; REAL X,Y);

```

```

BEGIN;

```

```

      RETURN ((-POLY!A[P]*X-POLY!B[P]*Y-POLY!D[P])/POLY!C[P])

```

```

END;

```

```

*****
***** "PROCEDURE TO LOAD THE VALUES OF NX,NY,NZ AND PX,PY,PZ
***** WITH COORDINATES OF THE ENDPONTS OF THE NEXT EDGE
***** OF THE CURRENT POLYGON.
***** RETURNS 0 IF NO MORE EDGES.
***** RETURNS -1 IF IT HAS FOUND ANOTHER EDGE."
*****

```

```

      "GET INDEX TO POINT NUMBER"

```

```

      "FILL UP PX, PY, PZ"

```

```

      "SAME FOR THE OTHER ENDPONIT"

```

```

      "PREPARE FOR NEXT CALL"

```

```

*****
***** "RETURN THE DEPTH OF THE POLYGON P AT SCREEN
***** COORDINATES X, Y"

```



```

DELTA THETA <= 0;
C1 <= WCODE (XX1 <= NX, YY1 <= NY);
C2 <= WCODE (XX2 <= PX, YY2 <= PY);
WHILE NOT (C1 = C2 = 0) DO BEGIN "CLIP";
  IF C1 LAND C2 THEN BEGIN INTEGER A1, A2, A3;
    "THE LINE IS BEING REJECTED. WE MUST COMPUTE
    THE 'ANGLE' SUBTENDED BY THE LINE.
    WE DO THIS BY CONSIDERING XX1, YY1, THE
    COORDINATES OF SOME INTERMEDIATE POINT."
    A1 <= ANGLE(NX, NY); A2 <= ANGLE(XX1, YY1); A3 <= ANGLE(PX, PY);
    IF ABS(A1 <= A1-A2) > 3 THEN IF A1 < 0 THEN A1 <= A1+8 ELSE A1 <= A1-8;
    IF ABS(A2 <= A2-A3) > 3 THEN IF A2 < 0 THEN A2 <= A2+8 ELSE A2 <= A2-8;
    DELTA THETA <= A1+A2;
    "RETURN 0 IF LINE DOES NOT INTERSECT WINDOW"
    RETURN (0);
  END;
  IF C1 = 0 THEN BEGIN C1 SWAP C2;
    XX1 SWAP XX2;
    YY1 SWAP YY2;
  END;
  IF C1 LAND '1 THEN PUSH (0, WLX) ELSE
  IF C1 LAND '10 THEN PUSH (0, WRX) ELSE
  IF C1 LAND '100 THEN PUSH (1, WLY) ELSE
  IF C1 LAND '1000 THEN PUSH (1, WTY)
  "RETURN -1 IF THE LINE INTERSECTS THE WINDOW"
  END "CLIP";
  RETURN (-1);
END "CLIP";

COMMENT
*****
***** DISPLAY PROCEDURES *****
*****
EXTERNAL PROCEDURE BEGIN FRAME;
EXTERNAL PROCEDURE END FRAME;
EXTERNAL PROCEDURE GEN DOT (INTEGER X, Y);
EXTERNAL PROCEDURE GEN LINE (INTEGER X, Y, X1, Y1);
REQUIRE "DIS" LOAD MODULE;

```

```

PROCEDURE DISPLAY!START;
BEGIN
  BEGIN!FRAME;
  END;
PROCEDURE DISPLAY!END;
BEGIN
  END!FRAME;
END;

PROCEDURE SHOWDOT (INTEGER X,Y);
BEGIN
  GEN!DOT (X,Y);
END;

PROCEDURE SHOWLINE (INTEGER X,Y,X1,Y1);
BEGIN
  GEN!LINE (X,Y,X1,Y1);
END;

"CALL THIS SUBROUTINE TO START A DISPLAY FRAME"
"CALL THIS SUBROUTINE TO END A DISPLAY FRAME
AND TO PUT IN UP ON THE SCREEN"
"GENERATE AN INTENSIFIED DOT AT X,Y"
"GENERATE A LINE FROM X,Y TO X1,Y1"

COMMENT
*****
READ - I N   D A T A   S T R U C T U R E
*****
PROCEDURE TO FILL UP DATA STRUCTURE FROM A FILE WITH COORDINATE INFORMATION, ETC. IN IT. THE DETAILS
OF THIS PROCEDURE ARE NOT IMPORTANT, AND ARE LISTED HERE JUST SO THAT WE CAN BE SURE THE PROGRAM
WORKS.

PROCEDURE READFILE;
BEGIN
  INTEGER POINTNUM, EDGENUM, POLYNUM, I, J, K, L; REAL R;
  DEFINE G(X) = "ARRAYIN(1,R,1); X <= R";
  INTENDED PROCEDURE FINDERGE;

```

```

OUTSTR ("FILE NAME:");
OPEN ("DSK", 13, 2, 0, 200, L, L);
LOOKUP (1, INCHWL, L);

G (POINTNUM);
G (EDGENUM);
G (POLYNUM);

"READ NUMBER OF POINTS IN FILE"
"NUMBER OF EDGES"
"NUMBER OF POLYGONS"

IF POINTNUM > POINTMAX OR EDGENUM > EDGEMAX OR POLYNUM > POLYMAX THEN
  OUTSTR ("TOO MUCH DATA"& 15& '12')
ELSE BEGIN
  FOR I <- 1 STEP 1 UNTIL POINTNUM DO BEGIN
    G(XS[I]); G(YS[I]); G(ZS[I])
  END;
  FOR I <- 1 STEP 1 UNTIL EDGENUM DO BEGIN
    G(ED1[I]); G(ED2[I]); EDLINK[I] <- -1
  END;
  POLYPTR <- 0;
  "INITIALIZE POINTER TO ALL OF POLYGONS"
  FOR I <- 1 STEP 1 UNTIL POLYNUM DO BEGIN
    POLYLINK[I] <- POLYPTR; POLYPTR <- I; "LINK ON LIST OF POLYGONS"
    G (J); "READ AND DISCARD SHADING VALUE"
    G (J); "READ NUMBER OF EDGES IN THIS POLYGON"
    POLYEDGE[I] <- K <- FINDEGE; "INDEX OF FIRST EDGE"
    WHILE J NEQ 1 DO BEGIN
      K <- EDLINK[K] <- FINDEGE; "LINK EDGES TOGETHER"
      J <- J-1
    END;
    EDLINK[K] <- 0;
    "TERMINATE LIST"
    "NOW RUN DOWN LIST OF EDGES
    MAKING SURE THEY ARE IN CORRECT ORDER"
    "HEAD OF LIST"
    K <- POLYEDGE[I];
    J <- ED2[K];
    IF J NEQ ED1[EDLINK[K]] AND J NEQ ED2[EDLINK[K]] THEN ED1[K] SWAP ED2[K];
    WHILE K DO BEGIN
      J <- EDLINK[K];
      IF J AND ED1[K] NEQ ED1[J] THEN ED1[J] SWAP ED2[J];
      K <- J
    END
  END
END;
END;

```

COMMENT

INITIALIZATION OF POLYGONS

```

PROCEDURE INITPOLYGONS;
BEGIN REAL X1, X2, X3, Y1, Y2, Y3, Z1, Z2, Z3;
      INTEGER CHANGE, OLDP, J;

      "PROCEDURE TO INITIALIZE POLYGON INFORMATION:
      (1) COMPUTE PLANE COEFFICIENTS OF EACH POLYGON
      (2) COMPUTE ZMIN FOR EACH POLYGON
      (3) SORT POLYGON LIST BY ZMIN VALUES"

```

```

      OLDP ← 0;
      P ← POLYPTR;
      WHILE P DO BEGIN "EACH POLYGON"
        NEXTEDGE ← POLY!EDGE[P];
        GET!NEXT!EDGE;
        X1 ← PX; Y1 ← PY; Z1 ← PZ;
        GET!NEXT!EDGE;

```

"INITIALIZE GET!NEXT!EDGE ROUTINE"

"SAVE COORDINATES OF ONE POINT"

"NOW GET TWO MORE POINTS"

"WE NOW HAVE COORDINATES OF THREE POINTS:

```

      X1 Y1 Z1
      PX PY PZ
      NX NY NZ

```

WE WILL USE THESE POINTS TO COMPUTE THE PLANE
COEFFICIENTS. THIS COMPUTATION ASSUMES THAT THE
3 POINTS ARE NOT COLINEAR."

```

      X3 ← NX-X1; Y3 ← NY-Y1; Z3 ← NZ-Z1;
      X2 ← PX-X1; Y2 ← PY-Y1; Z2 ← PZ-Z1;

```

```

      POLY!A[P] ← Y3*Z2 - Y2*Z3;
      POLY!B[P] ← X2*Z3 - X3*Z2;
      POLY!C[P] ← X3*Y2 - X2*Y3;

```

```

      POLY!D[P] ← -(POLY!A[P]*X1+POLY!B[P]*Y1+POLY!C[P]*Z1);

```

```

"NOW COMPUTE ZMIN FOR THIS POLYGON"
"GREATER THAN LARGEST POSSIBLE SCREEN COORDINATE"
ZMIN <= 0;
NEXTEDGE <- POLY:EDGE[P];
WHILE GET:NEXT:EDGE DO IF ZMIN > PZ THEN ZMIN <- PZ;
POLY:ZMIN[P] <- ZMIN;

IF POLY:IC[P] = 0 THEN BEGIN
"CAN'T POSSIBLY SEE THIS POLYGON"
"THE Z COEFFICIENT IS ZERO, WHICH MEANS THE
POLYGON IS EDGE-ON TO THE VIEWER"
DELETE THIS POLYGON FROM THE LIST OF POLYGONS"
IF OLDP = 0 THEN POLYPTR <- P ELSE POLY:LINK[OLDP] <- POLY:LINK[P]
END ELSE OLDP <- P;
P <- POLY:LINK[P];

END "EACH POLYGON";

DO BEGIN "SORT"
CHANGE <- FALSE;
OLDP <- 0; P <- POLYPTR;

WHILE P DO BEGIN
"GET INDEX TO NEXT POLYGON IN LIST"
J <- POLY:LINK[P];
IF J NEQ 0 AND POLY:ZMIN[P] > POLY:ZMIN[J] THEN BEGIN
"INTERCHANGE P & J IN THE LIST"
IF OLDP = 0 THEN POLYPTR <- J ELSE POLY:LINK[OLDP] <- J;
POLY:LINK[P] <- POLY:LINK[J];
POLY:LINK[J] <- P;
CHANGE <- TRUE;
"PREVIOUS CELL"
OLDP <- J;
END ELSE BEGIN
OLDP <- P;
P <- POLY:LINK[P]
END
END
END "SORT" UNTIL NOT CHANGE;

LEVEL <- 1;
FOR P <- 1 STEP 1 UNTIL POLYMAX DO HISTORY[1,P] <- -1; "INITIALIZE HISTORY TO LOOK AT EVERY POLYGON"

##
##
END;

```

 COMMENT

 MAIN RECURSIVE PROCEDURE

 THIS IS THE MAIN PROCEDURE. IT EMBODIES THE LOOKER, THE THINKER, AND MOST OF THE CONTROLLER.
 ARGUMENTS TO THE PROCEDURE SPECIFY THE LOCATION AND SIZE OF THE WINDOW UNDER CONSIDERATION. IF THE
 WINDOW MUST BE SUBDIVIDED, THIS PROCEDURE CALLS ITSELF RECURSIVELY.

RECURSIVE PROCEDURE WARNOCK (INTEGER LEFT,BOTTOM,SIZE);
 BEGIN
 "FIRST, INITIALIZE THE WINDOW LIMITS FOR THE
 CLIP ROUTINE. THE TOP AND RIGHT EDGES OF THE
 WINDOW SHOULD BE JUST LESS THAN THE BOTTOM AND
 LEFT EDGES OF THE WINDOWS ADJACENT TO THIS ONE."

DEFINE EPSILON = ".00001";

WLX < LEFT; WRX < WLX+SIZE-EPSILON;
 WBY < BOTTOM; WTY < WBY+SIZE-EPSILON;

 ##
 FOR P < 1 STEP 1 UNTIL POLYMAX DO HISTORY[LEVEL+1,P] < HISTORY[LEVEL,P];

"INITIALIZE THINKER LISTS"

SURROUNDERS < INTERSECTORS < 0;
 ZMINMAX < 0;

 ##
 P < POLYPTR;
 WHILE P DO BEGIN "LOOKER"
 IF POLY!ZMIN[P] > ZMINMAX THEN DONE; "DO NOT LOOK FARTHER"
 THETA < HISTORY[LEVEL,P];
 IF THETA = -1 THEN BEGIN "MUST EXAMINE"
 THETA < 0;
 NEXTEGE < POLY!EDGE[P];
 "INITIALIZE CUMULATIVE ANGLE AROUND WINDOW"
 "INITIALIZE GETNEXT!EDGE ROUTINE"

WHILE GETNEXT!EDGE DO BEGIN "LOOP FOR ALL EDGES"
 IF CLIP THEN BEGIN "EDGE PASSES THROUGH THIS WINDOW"
 POLY!LIST[P] < INTERSECTORS;
 INTERSECTORS < P; "PUT ON INTERSECTORS LISTS"
 THETA < -1; "THE POLYGON INTERSECTS THE WINDOW"
 DONE; "NO USE CLIPPING OTHER EDGES"

 TUEETA,DELTA,THETA, "IPDATE,ANGLE"

```

Z1 ← GETZ (P.WLX.WBY);
Z2 ← GETZ (P.WLX.WTY);
Z3 ← GETZ (P.WRX.WBY);
Z4 ← GETZ (P.WRX.WTY);

IF Z2 > Z1 THEN Z1 ← Z2;
IF Z3 > Z1 THEN Z1 ← Z3;
IF Z4 > Z1 THEN Z1 ← Z4;
ZMINMAX ← Z1;

"AND STORE IN ZMINMAX"

"NOW COMPUTE THE MAXIMUM OF THESE"

"GET DEPTHS AT CORNERS OF WINDOW"

END "FOUND SURROUNDER";
P ← POLYLINK[P]
END "LOOKER";
BEGIN "THINKER"

"NOW WE CAN BE A SMART THINKER OR A NOT-SO-SMART THINKER;
BOTH THINKERS HANDLE PENETRATING PLANES CORRECTLY.
FIRST, WE FIND ONE CRITICAL SURROUNDER OF THE WINDOW. IF
TWO SURROUNDERS PENETRATE, WE GO TO FAIL.
THEN, EACH INTERSECTOR IS COMPARED TO THE CRITICAL SURROUNDER:
1. THE INTERSECTOR IS HIDDEN BY THE SURROUNDER ---
   REMOVE INTERSECTOR FROM LIST
2. THE INTERSECTOR IS COMPLETELY IN FRONT OF SURROUNDER ---
   LEAVE IN LIST
3. OTHERWISE, INTERSECTOR MUST PENETRATE SURROUNDER; FAIL

WHEN THIS PROCESS IS FINISHED, WE EXAMINE THE INTERSECTOR LIST:
SMARTER THINKER,
IF # OF POLYGONS IN INTERSECTOR LIST = 0, DO NOTHING
   = 1, DISPLAY VISIBLE
   > 1, FAIL

NOT-SO-SMART THINKER;
IF # OF POLYGONS IN INTERSECTOR LIST = 0, DO NOTHING
   > 0, FAIL

WE WILL SHOW THE SMART THINKER"

```

```

ZMIN1 ← ZMIN2 ← ZMIN3 ← ZMIN4 ← 0 ;
HIDER ← PENETRATE ← 0 ;

WHILE SURROUNDERS DO BEGIN
    Z1 ← GETZ (SURROUNDERS,WLX,WBY) ;
    Z2 ← GETZ (SURROUNDERS,WLX,WTY) ;
    Z3 ← GETZ (SURROUNDERS,WFX,WBY) ;
    Z4 ← GETZ (SURROUNDERS,WFX,WTY) ;

    "INITIALIZE DEPTHS OF HIDER"
    "GET DEPTHS OF SURROUNDER AT CORNERS OF WINDOW"

    "NOW SEE IF THE DEPTHS OF THIS SURROUNDER ARE CLOSER
    TO THE EYE THAN ANY OTHER SURROUNDERS"

    IF Z1 < ZMIN1 AND Z2 < ZMIN2 AND Z3 < ZMIN3 AND Z4 < ZMIN4 THEN
        BEGIN "NEW HIDER WHICH IS CLOSER THAN PREVIOUS ONES"
            HIDER ← SURROUNDERS ;
            ZMIN1 ← ZMAX1 ← Z1 ;
            ZMIN2 ← ZMAX2 ← Z2 ;
            ZMIN3 ← ZMAX3 ← Z3 ;
            ZMIN4 ← ZMAX4 ← Z4 ;
        END ELSE
        IF Z1 > ZMAX1 AND Z2 > ZMAX2 AND Z3 > ZMAX3 AND Z4 > ZMAX4 THEN
            BEGIN "NEW POLYGON IS DEEPER THAN THE PRESENT HIDER -- DO NOTHING"
            END ELSE
            BEGIN "POLYGONS PENETRATE"
                "COMPUTE MINIMUM AND MAXIMUM DEPTHS OF
                THE PENETRATING PLANES"
                PENETRATE ← TRUE ;
                IF Z1 < ZMIN1 THEN ZMIN1 ← Z1 ;
                IF Z2 < ZMIN2 THEN ZMIN2 ← Z2 ;
                IF Z3 < ZMIN3 THEN ZMIN3 ← Z3 ;
                IF Z4 < ZMIN4 THEN ZMIN4 ← Z4 ;

                IF Z1 > ZMAX1 THEN ZMAX1 ← Z1 ;
                IF Z2 > ZMAX2 THEN ZMAX2 ← Z2 ;
                IF Z3 > ZMAX3 THEN ZMAX3 ← Z3 ;
                IF Z4 > ZMAX4 THEN ZMAX4 ← Z4 ;
            END ;
        END ;
    END ;
    "LOOK AT NEXT ONE"
END ;

```

```

IF NOT PENETRATE THEN BEGIN "NO SURROUNDERS PENETRATED"
    "NOW REMOVE FROM INTERSECTORS LIST ANY POLYGONS
    HIDDEN BY THE HIDER POLYGON"
    OLDP ← 0; P ← INTERSECTORS;
    WHILE P AND HIDER DO BEGIN
        Z1 ← GETZ (P.WLX.WBY);
        Z2 ← GETZ (P.WLX.WTY);
        Z3 ← GETZ (P.WRX.WBY);
        Z4 ← GETZ (P.WRX.WTY);
        IF Z1 > ZMAX1 AND Z2 > ZMAX2 AND Z3 > ZMAX3 AND Z4 > ZMAX4 THEN
            BEGIN "COMPLETELY HIDDEN -- REMOVE FROM LIST"
                BEGIN "COMPLETELY HIDDEN -- REMOVE FROM LIST"
                    J ← POLY!LIST[P]; "NEXT ENTRY ON LIST"
                    IF OLDP = 0 THEN INTERSECTORS ← J ELSE POLY!LIST[OLDP] ← J
                END ELSE BEGIN "ONLY NEEDED BECAUSE SMART THINKER"
                    "SAVE FOR NEXT TIME THROUGH"
                    OLDP ← P;
                IF Z1 < ZMIN1 AND Z2 < ZMIN2 AND Z3 < ZMIN3 AND Z4 < ZMIN4 THEN
                    BEGIN "INTERSECTOR COMPLETELY IN FRONT OF SURROUNDER --- DO NOTHING"
                    END ELSE
                    BEGIN "INTERSECTOR PENETRATES SURROUNDER(S)"
                        PENETRATE ← TRUE; "FLAG THAT PENETRATION OCCURRED"
                    DONE; "NO USE LOOKING AT MORE INTERSECTORS."
                    END
                END;
            END;
        P ← POLY!LIST[P]
    END;

```

```

IF NOT PENETRATE THEN BEGIN "NO INTERSECTORS PENETRATE SURROUNDER(S)"
    "NOW LOOK AT INTERSECTORS REMAINING IN THE LIST AND
    DECIDE WHAT TO DO:
    1. IF NO INTERSECTORS OR ALL HAVE BEEN REMOVED
    FROM THE INTERSECTOR LIST BECAUSE THEY ARE
    HIDDEN BY SURROUNDERS WE CAN RETURN -- THE
    SCREEN IS BLANK IN THIS AREA"
IF INTERSECTORS = 0 THEN RETURN;
    "2. IF THERE IS JUST ONE POLYGON WHICH INTERSECTS
    THE WINDOW, THEN WE MAY DISPLAY ANY
    PORTIONS OF EDGES WHICH INTERSECT THE
    WINDOW, NOTICE THAT THERE MAY ALSO
    HAVE BEEN A SURROUNDER, BUT WHICH IS DEEPER
    THAN THE INTERSECTOR."
IF POLY!LIST[INTERSECTORS] = 0 THEN BEGIN "JUST 1 POLYGON"
    NEXTEDGE < POLY!EDGE[INTERSECTORS];
    WHILE GET!NEXT!EDGE DO
        IF !CLIP THEN SHOWLINE (XX1,YY1,XX2,YY2);
    RETURN;
END "NO INTERSECTORS PENETRATE SURROUNDER(S)"
END "NO SURROUNDERS PENETRATED"
END "THINKER";

"COME HERE IF PROCESSING OF THIS WINDOW
HAS FAILED."

IF SIZE < 2 THEN SHOWDOT (LEFT,BOTTOM) ELSE
    BEGIN
        SIZE <= SIZE % 2;
        LEVEL <= LEVEL+1;
        WARNOCK (LEFT,BOTTOM,SIZE);
        WARNOCK (LEFT+SIZE,BOTTOM,SIZE);
        WARNOCK (LEFT,BOTTOM+SIZE,SIZE);
        WARNOCK (LEFT+SIZE,BOTTOM+SIZE,SIZE);
        LEVEL <= LEVEL-1;
    END
END "WARNOCK";

"HERE BEGINS MAIN EXECUTION"

```