Hidden Feature Elimination and Visible Polygon Return in UNIGRAFIX 2

Paul Wensley

Master's Project Report
Under Direction of
Prof. Carlo H. Séquin

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
April, 1984

ABSTRACT

UNIGRAFIX is a graphics rendering system that runs under the UNIX * operating system. It consists of a descriptive language and several programs that allow a user to create, modify, and display scenes consisting of polyhedral objects. UNIGRAFIX 2 is an enhanced and refined implementation of the original UNIGRAFIX system created by Paul Strauss. †

UNIGRAFIX 2 relies on a new hidden surface algorithm based on the CROSS algorithm by Hamlin and Gear. In this algorithm visibility information is passed from vertices to edges, and through those to other vertices. At crossings of contour edges, visibility is reexamined. This algorithm is ideally suited for efficient computation of all visible polygons. A new option, -vf, has been added to Unigrafix, that will write all visible polygons to a specified file.

* UNIX is a trademark of Bell Laboratories
† See The Unigrafix System: User's Manual

1. INTRODUCTION

Hidden surface removal in the original UNIGRAFIX system was accomplished by a scanline algorithm that maintained a list of edges that were crossing the current scanline. All faces were implicitly described by bounding edges on the left and right. When a new edge was encountered, the corresponding face was put on a depth sorted stack, referenced by the active edge. When an ending edge was encountered on the scanline, the next visible face was determined by popping the ending face from the stack; the new top stack face was then used to draw the next span. Readers are referred to the UNIGRAFIX User's Manual and Implementation Guide for further information about UNIGRAFIX.[1]

After much experimentation, it was determined that a new hidden surface algorithm was needed for expanding UNIGRAFIX. What was desired was an algorithm that exploits more of the object's coherence and is able to return explicitly all visible faces with high precision in object space. There are a number of hidden surface algorithms that can be called *edge-intersection algorithms* that are object space algorithms that can be used to determine the visible portions of faces. The first such algorithm was that of Appel[2] which defines a *quantitative invisibility* of a point as the number of faces that lie between the point and the viewpoint. The solution to the hidden surface problem requires computing the quantitative invisibility of every point on each contour edge. The amount of computation is reduced by using *edge coherence:* the quantitative invisibility of a contour edge can only change when the projection of that edge crosses another contour edge. At such intersections, the quantitative invisibility can only increase or decrease by one. The quantitative invisibility of an initial vertex can be found by an exhaustive search to see how many faces hide the vertex.

Loutrel's[3] algorithm is very similar to that of Appel, with the comparison at the intersections of contours being done in the two dimensional projection of the scene onto the picture plane. Galimberti and Montanari[4] also have a similar algorithm except that instead of computing the number of faces that obscure an edge, they compute the set of obscuring faces.

The CROSS algorithm by Hamlin and Gear[5] is also an edge-intersection algorithm that exploits the edge coherence of the object but is more suitable for rendering the object on a raster device. The algorithm, like Loutrel's, calculates intersections in the two dimensional projection of the scene to the picture plane. The algorithm is a scanline algorithm: it maintains an ordered list of the edges that fall under the current scanline and limits the search for obscuring faces to the faces that are bordered by the neighboring edges in the list. Visibility information is passed from edges to vertices and from them to other edges so that the amount of computation is limited.

Since all visibility calculations are incremental, a number of unpleasant singularities can occur with these algorithms that will propagate errors to other portions of the picture.

Galimberti and Montanari have reported such problems and reported ad hoc solutions in their paper. We describe our solutions to such problems as coincidence of points and warped polygons.

The algorithm as stated by Hamlin and Gear seemed well suited to our application but was too constrained. However, it was an ideal starting point for both a rendering algorithm as well as an algorithm for recognizing visible portions of faces.

2. OVERVIEW OF RENDERING ALGORITHM

The new hidden surface algorithm in UNIGRAFIX 2 also uses a scanline algorithm as an ordering principle to find and process all vertices and edges. The scene is processed from the uppermost scanline to the lowest scanline (from max y to min y in viewing space) determining visibility of each "interesting" point that is encountered. An interesting point is either a vertex or a crossing between two edges.

When the UGPLOT program is invoked, the plotting options are read in from the command line and checked for consistency. The various options allowed in UNIGRAFIX 2 are summarized in the appendix. The scene file is then read and the internal data structure created. The viewing transformation matrix and its inverse are generated at this time.

Back faces, i.e. those faces which are turned away from the viewer, are removed after computing the plane equation for each face in viewing space. If the z component of its face normal is non negative, then the face will be flagged invisible. For visible faces, the illumination value is computed, taking into account the face normal in world coordinates and all light sources.

All of the vertices in the scene are transformed to *viewing coordinates* by multiplication by the *viewing transformation matrix.* If the view is a perspective view, then the perspective transformation is applied at this time. The vertices are then bucket sorted according to the scanline in which they will appear. The number of buckets is the same as the number of scanlines in the output device.

The interesting points in the scene are then processed from top to bottom, generating the type of output specified by the command line options.

In pseudo code, the main loop is:

- *Read command line options and set viewing transforms*
- *Read data file and build topologically invariant structure*
- *Flatten all arrays and instances*
- *Convert all vertices to device coordinates*
- *If a perspective view, do the appropriate division*
- *Bucket sort all vertices according to their maximum y coordinate*
- *For scanline = MAX to MIN do*
  - *While a vertex or crossing occurs on this scanline:*
    - *Choose object with largest y value*
    - *If vertex chosen, call PROC_VERTEX*
    - *If crossing chosen, call PROC_CROSSING*
  - *Output the spans for this scanline from the ACTIVE EDGE LIST*

3. INTERNAL SCENE STORAGE

As with the first version, UNIGRAFIX 2 constructs an internal representation of the scene consisting of structures and pointers which can be more efficiently managed than their original ASCII representation. However, the internal scene storage in UNIGRAFIX 2 is very different than that of UNIGRAFIX 1.

3.1. Vertices

UNIGRAFIX 2 is vertex oriented and the vertex structures will be referenced by many other types of structures. As each *vertex* command in the scene file is read in, a new VERTEX structure is created and linked to the others in two ways. An ordered linking of the vertices is kept so that the vertices may again be referenced in the same order as they appeared in the original file. Moreover, to speed up lookups when other objects reference these vertices, they are also entered into a hash table, with the key being the hashed value of the vertex identifier. The VERTEX structure contains the three world coordinates of the vertex, a pointer to the list of edges that this vertex belongs to, a pointer to the name string that was entered into a global string table, and various flags.

3.2. Edges

An internal EDGE structure is created for each pair of adjacent vertices that occur in a face or wire command. Edges are unique; if more than one face or wire share an edge, then they all point to the same internal EDGE structure. From each edge there are pointers to the two vertices that form the edge as well as pointers to all wires and faces that make use of this edge.

3.3. Wires

A *wire* statement in UNIGRAFIX 2 describes a sequence of line segments. A WIRE structure is created when a wire command is read from the scene file. Since a wire is specified by the vertices that it goes through, as each pair of vertices is read, a lookup is done for the vertices specifying the new edge. The list of edges belonging to the two vertices is checked to see if an edge already exists between them. If so, then mutual references are added to the EDGE and WIRE structures. If the edge does not already exist, then a new one is created. Stored with the WIRE are a possible wire identifier, a possible color value, and various flags. Each WIRE structure is also linked up in a list of all wires in the order they are encountered in the input file.

3.4. Faces

Face statements in UNIGRAFIX are treated analogously to wires, except that the FACE structure has additional room for the plane equations in world coordinates and in viewing coordinates as well as an illumination value.

As an example, consider the UNIGRAFIX file:

```
v A 0 0 1;
v B 0 1 0;
v C 1 0 0;
v D 1 1 0;
f face (A B C);
w wire (D C B);
```

A slightly simplified view of the internal structure of the various constructs discussed so far is shown in figure 1.

4. THE CROSS ALGORITHM

4.1. Classification of edges

The hidden surface removal algorithm passes visibility information from edges to the attached vertices and from them to the attached edges. The visibility of an edge gets checked when it starts from a visible vertex or when edges cross. For efficient handling of the checks at edge crossings, edges are classified as either *LEFT, MIDDLE, RIGHT* or *WIRE*. For each edge inserted into the active edge list, the "direct" and "reversed" usage lists are scanned to find the top-most visible face on either side of the edge. If there are no visible faces and the edge does not belong to a wire, then the edge is not treated in any further analysis. Because the vertices of

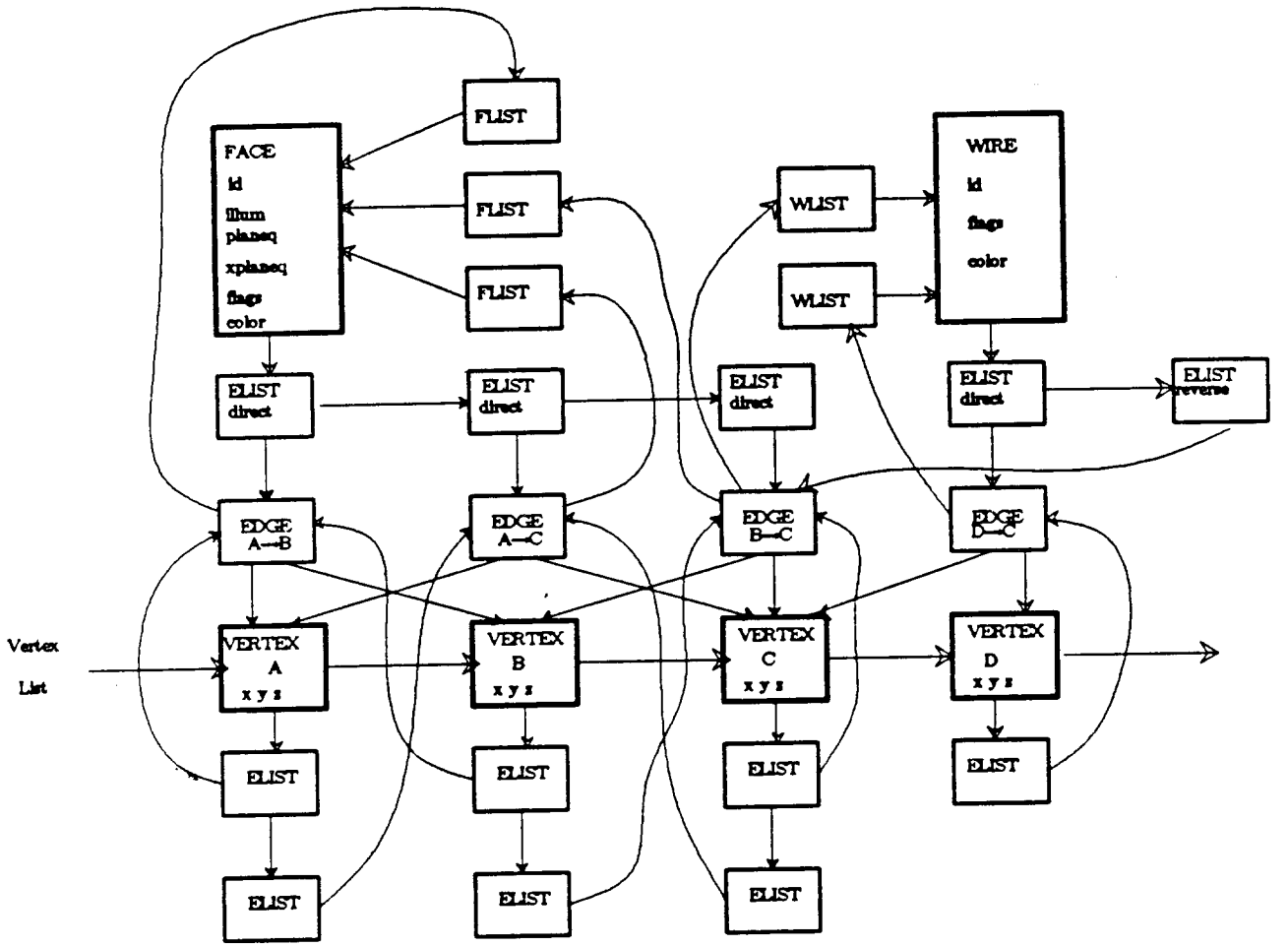


Figure 1. *UNIGRAFIX Data Structure*

visible faces and thus the bounding edges are specified in clockwise order, the face will always be on the right side of the edge when going from the starting vertex to the ending vertex and this makes the classification simple. Edges that are horizontal are treated as if they were sloping down to the right.

As an example, consider the UNIGRAFIX file:

```
v t   0   10 0;
v a  -5    0  0;
v b   5    0  0;
v c  -5  -10 0;
v d   5  -10 0;
v wl -10 -10 0;
v wr  10 -10 0;

f f1 (a b c);
f f2 (b d c);
w w1 (a t b);
w w2 (wl c d wr);
```

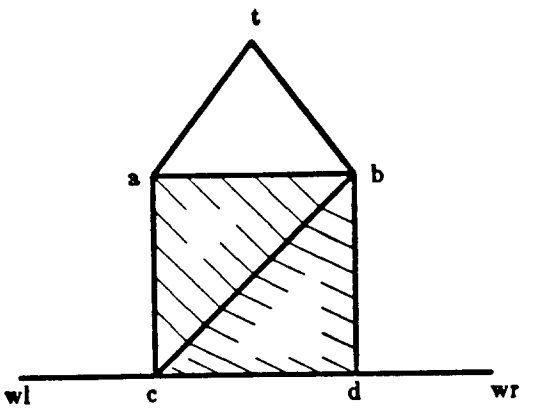


Figure 2. *Classification of edges*

This file produces the picture shown in figure 2 and the classification shown in table 1.

Member of face/wire	Starting vertex	Ending vertex	Edge classification
f1	a	b	RIGHT
f1 f2	b	c	MIDDLE
f1	c	a	LEFT
f2	b	d	RIGHT
f2 w2	d	c	LEFT
w1	a	t	WIRE
w1	t	b	WIRE
w2	wl	c	WIRE
w2	d	wr	WIRE

Table 1. *Classification of edges for figure 2.*

When edges are classified as LEFT, MIDDLE or RIGHT, the face that is visible on the left and the right of the edge is determined. In the case of a "book" of faces (see figure 3), a calculation needs to be done to determine which face is visible.

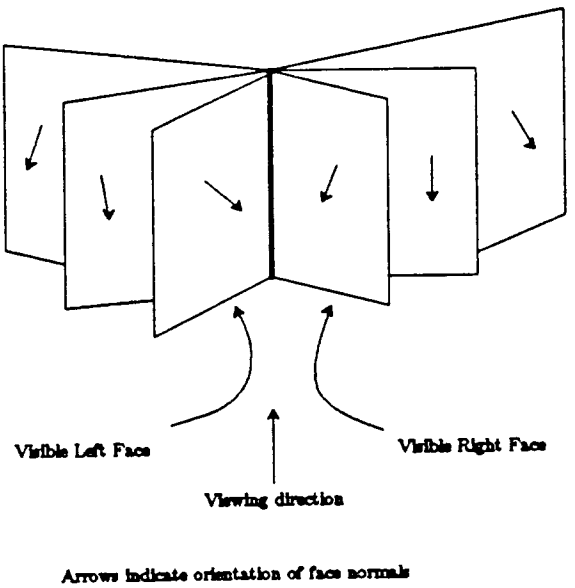


Figure 3. *Determination of visible faces for an edge.*

To determine which face is visible, the X-Y plane is divided into three sections originating at the starting vertex of the edge in question (see figure 4).

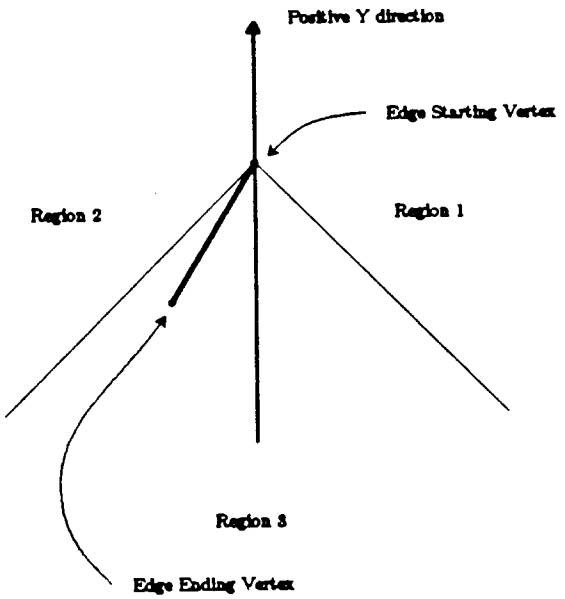


Figure 4. *Case analysis for visible face determination.*

The plane normal of two competing faces is checked according to the tests below:

Region 1

If finding the RIGHT face of the edge, the face with the **smaller** Y component of the face normal is in front. If finding the LEFT face of the edge, the face with the **larger** Y component of the face normal is in front.

Region 2

If finding the RIGHT face of the edge, the face with the **larger** Y component of the face normal is in front. If finding the LEFT face of the edge, the face with the **smaller** Y component of the face normal is in front.

Region 3

If finding the RIGHT face of the edge, the face with the **smaller** X component of the face normal is in front. If finding the LEFT face of the edge, the face with the **larger** X component of the face normal is in front.

As we progress from the uppermost scanline to the lowest, a list of edges that intersect the current scanline is maintained. This *Active Edge List* is ordered from left to right and contains ACTIVE EDGE structures that point to the real edge from which they came, a flag to mark whether the edge is visible or not, and pointers to the faces that are visible on the left and right side of the active edge. The *Active Edge List* is doubly linked for easy insertion and removal of active edges.

4.2. Calculation of crossing points

Every time the active edge list changes, we need to check for new crossing points for any newly created pair of neighboring edges. Because we rely so heavily upon the topology of the scene to propagate visibility information, we must guarantee that crossings are calculated reliably. In particular, we never want to miss crossings that are necessary for a consistent treatment of the scene. The edges are first checked to see if they have a vertex in common; if they do, then they don't cross. Next, the slopes of the edges are checked to see if the edges diverge, thus making future crossings of this pair impossible. Otherwise, if the two edges are E1 and E2 with corresponding endpoints "top" and "bottom",

$$E1\text{: } (X_{1_{top}}\ ,\ Y_{1_{top}})\ ,\ (X_{1_{bottom}}\ ,\ Y_{1_{bottom}})$$

and

$$E2\text{: } (X_{2_{top}}\ ,Y_{\ 2_{top}})\ ,\ (X_{2_{bottom}}\ ,\ Y_{2_{bottom}})$$

we form the parametrization of the lines as:

$$X_1 = X_{1_{bottom}} + t_1 * (X_{1_{top}} - X_{1_{bottom}})$$

$$Y_1 = Y_{1_{bottom}} + t_1 * (Y_{1_{top}} - Y_{1_{bottom}})$$

$$X_2 = X_{2_{bottom}} + t_2 * (X_{2_{top}} - X_{2_{bottom}})$$

$$Y_2 = Y_{2_{bottom}} + t_2 * (Y_{2_{top}} - Y_{2_{bottom}})$$

The crossing point is the point where:

$$X_1 = X_2$$

$$Y_1 = Y_2$$

To find this point, we solve for the first edge's parameter t (the edge on the left). Let:

$$\delta_{1_x} = X_{1_{top}} - X_{1_{bottom}}$$

$$\delta_{1_y} = Y_{1_{top}} - Y_{1_{bottom}}$$

$$\delta_{2_x} = X_{2_{top}} - X_{2_{bottom}}$$

$$\delta_{2_y} = Y_{2_{top}} - Y_{2_{bottom}}$$

Then:

$$t_1 = \left| \frac{\delta_{2_y} * X_{1_{bottom}} - \delta_{2_x} * Y_{1_{bottom}} - \delta_{2_y} * X_{2_{bottom}} - \delta_{2_x} * Y_{2_{bottom}}}{\delta_{2_x} * \delta_{1_y} - \delta_{2_y} * \delta_{1_x}} \right|$$

If the denominator is zero, then the lines are parallel and there is no crossing. If t is less than zero, then there is no crossing. If t is greater than one, then we force t to be one, knowing that since the lines do not diverge, then they must cross, and we create an immediate crossing. Next, calculate the second parameter by substitution into one of the above equations, i.e.

$$t_2 = \left| \frac{X_{1_{bottom}} + \delta_{1_x} * t_1 - X_{2_{bottom}}}{\delta_{2_x}} \right|$$

or

$$t_2 = \left| \frac{Y_{1_{bottom}} + \delta_{1_y} * t_1 - Y_{2_{bottom}}}{\delta_{2_y}} \right|$$

Which equation to use is chosen by which denominator has the greatest magnitude. Again if t is less than zero, then there is no crossing and if t is greater than one, then force it to be one. X and Y are calculated by substitution into one of the above equations.

4.3. Scanline processing

We now want to process the next interesting point in the scene in descending y order. An interesting point is either a vertex or a crossing point of two edges. Because vertices and crossing points have been bucket sorted, this is very fast, just comparing the top two structures in the y sorted vertex list and crossing list corresponding to this scanline (see Figure 5). If two structures have the same y value, then the ordering is from left to right. If both y and x are the same, then preference is given to the crossing point.

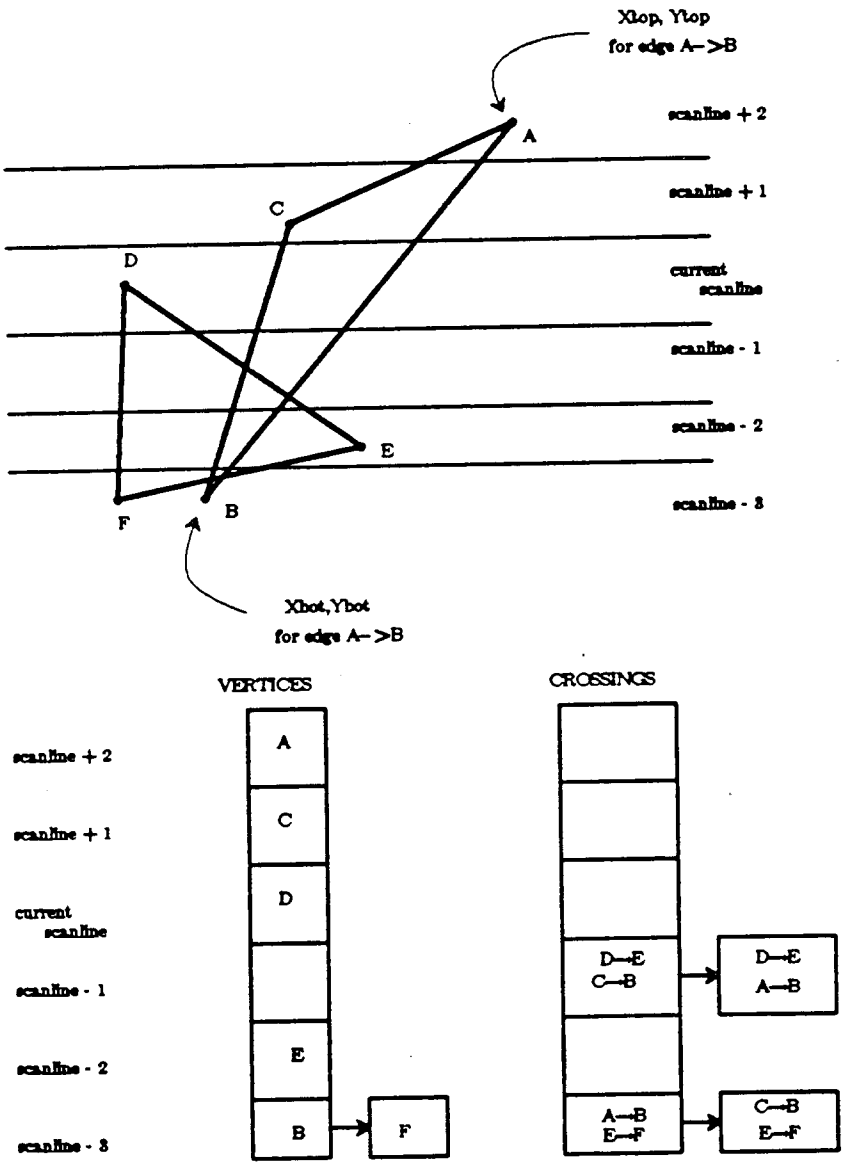


Figure 5. *Bucket sorted vertices and crossing points*

4.3.1. Vertex Processing

If a vertex is chosen as the next interesting object to process, we want to remove all edges that end on the vertex, determine the visibility of the vertex, and add all new emanating edges. In order to propagate the visibility and left and right face information correctly, all edges ending on a vertex must be adjacent in the *Active Edge List*. To this end, we must group the edges by forcing crossings to the left until all terminating edges are adjacent. The position of the ending edges in the *Active Edge List* is maintained as the position of any new edges starting from this vertex. All of the ending edges are now removed from the list. If any of the ending edges were visible, then the vertex is determined to be visible, too.

If no edges ended on this vertex, then the position and visibility of the vertex must be determined by scanning the *Active Edge List* from left to right, until the proper position is found where the vertex and all associated starting edges should be inserted into the active edge list. The z component of the visible face at the location of the new vertex is calculated and compared with the z component of the vertex and the visibility of the vertex is set accordingly.

All emanating edges are added to the active edge list at the position determined, from left to right, setting the edge's visibility and the left and right face pointers (see figure 6). If the vertex was determined to be invisible, then all new edges must also be invisible. If, however, the vertex was visible, then the visibility of each new edge must be calculated, because one of the new faces could obscure other edges from the vertex.

If a *LEFT* edge is being added, then the edge will be visible if the z coordinates of the new edge are less than those of the currently visible face on the left. If the z coordinates are the same or the vertex belongs to the face on the left, then the ending z coordinate is checked to see if the edge is in front of or behind the face to the left. If it is in front, then the edge is visible.

If a *MIDDLE* edge is being added, then the edge will be visible if its left face is the currently visible face on the left. Anytime a *RIGHT* edge is added, the right face must be determined by scanning the active edge list, from left to right, recording entries and exits from polygons until the edge in question is reached. The z components of the polygons at the vertex are compared and the one in front is the right face of the new *RIGHT* edge. If two polygons have the same z coordinate, then the x and y coordinates for the bottom of the edge are inserted into the plane equations. If this test does not result in a clear determination of which polygon is visible, then we know that the polygons have an edge in common and the visible polygon can be found from the same set of tests that were used to determine right and left visible faces of edges.

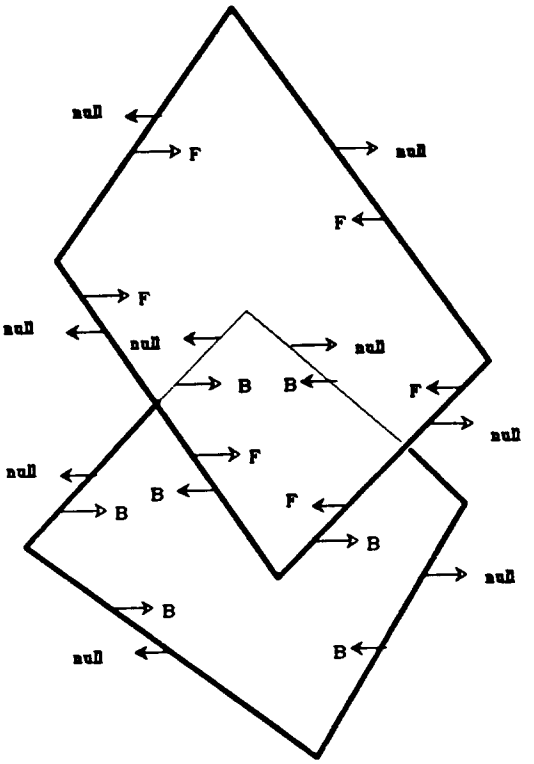


Figure 6. *Left and right polygon pointers*

Procedure to process a vertex (PROC_VERTEX)

- *Force adjacency of all edges that end on this vertex by forcing crossings to the left*
- *Remove all ending edges from the active edge list*
- *If any ending edge was visible, mark the vertex as visible*
- *If there were no ending edges, calculate visibility of vertex and position of any new edges:*
  - *First check if the vertex is part of the currently visible polygon on the left, in which case the vertex is marked visible.*
    Otherwise, the Z coordinate of the vertex is compared against the currently visible face on the left.
- *Add all new edges, from left to right, for each edge:*
  - *Classify edge as LEFT, MIDDLE, RIGHT or WIRE and determine left and right visible faces.*
  - *If the edge being added is a LEFT edge, then if the vertex belongs to the face on the left or has the same Z coordinate, then the slopes of the line and the face must be checked to see if the edge should be visible.*
    The slope is checked by inserting the X and Y coordinates of the endpoints of the edge into the plane equation and comparing Z values.
  - *If the edge being added is a MIDDLE edge, it will be visible if its left face is the currently visible face.*
    The currently visible face on the right will be the edge's right face.

- *If the edge being added is a RIGHT edge, it will be visible if its left face is the currently visible face.*
  - *The currently visible face on the right of the edge is determined by scanning the active edge list recording entries and exits from faces, and comparing the depths of the faces in the manner described above.*
- *Insert all new edges from this vertex as a group into the active edge list checking for crossings to the left for the leftmost edge and checking for crossings to the right for the rightmost edge.*

4.3.2. Crossing Point Processing

When the next interesting point in the scene is a crossing between two edges in the *Active Edge List*, the visibility on the edges must be redetermined. Since the *Active Edge List* must be maintained from left to right, when two edges cross, they must be exchanged in the list. If the edges are currently not adjacent in the list, then crossings must be forced to the left to make them adjacent. The visibility of the edges and the left and right faces need to be updated according to the matrix in Table 2.

Procedure to process a crossing (PROC_CROSSING)

- *If the two edges that cross are not adjacent in the active edge list, then force crossings to the left*
- *Swap the edges in the active edge list*
- *Set visibility of edges and left and right face pointers according to Table 2.*

1st Edge	2nd Edge							
	LV	MV	RV	WV	LI	MI	RI	WI
RV	C2	I2b	I2a	C4b				
MV	I1b	Ec	Ed	P				
LV	I1a	Ea	Eb	P	C1a	C1b	C1c	C5a
WV	C4a	P	P					
RI			C3c					
MI			C3b					
LI			C3a					
WI			C5b					

L = Left edge, M = Middle edge, R = Right edge, W = wire,
V = Visible, I = Invisible

Table 2. *Edge crossing cases*

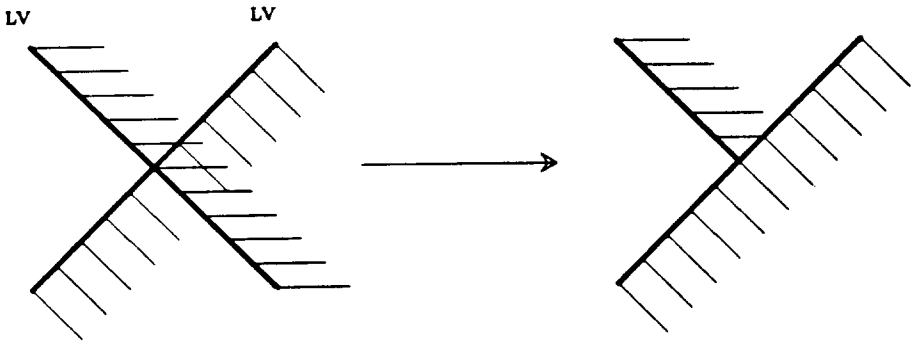
Blank entries indicate no action need be taken; otherwise, apply the specified operation as described below.

- P

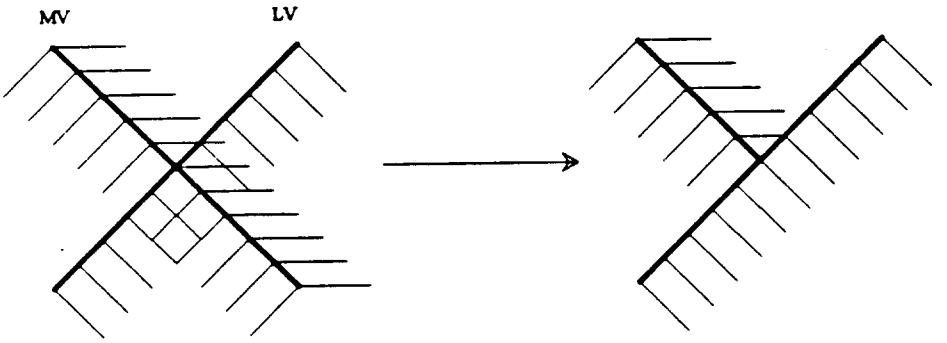
  Only the edge's left and right polygon pointers need be updated.

- I1

  In this case, we want to make the first edge invisible. The second edge's left face is now the first edge's left face.



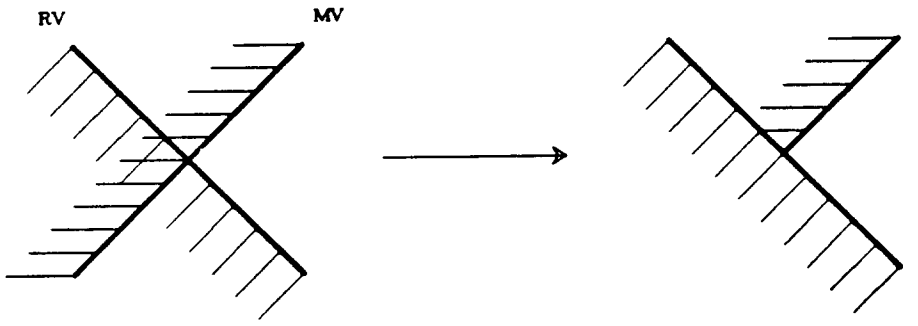
Case I1a



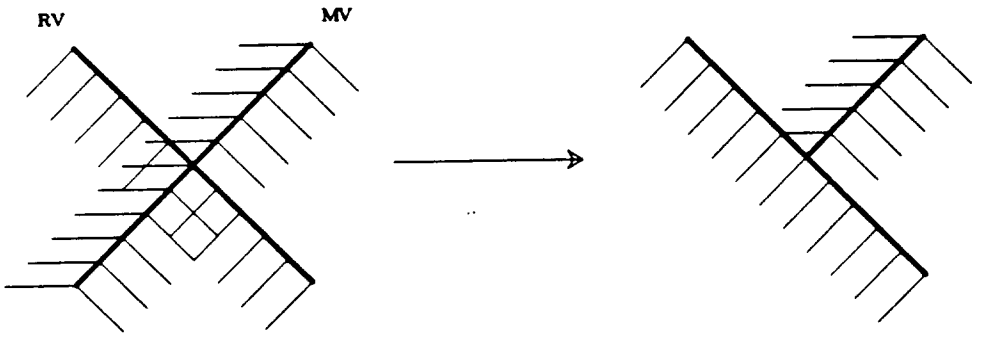
Case I1b

- I2

We want to make the second edge invisible. The first edge's right face is now the second edge's right face.



Case I2a

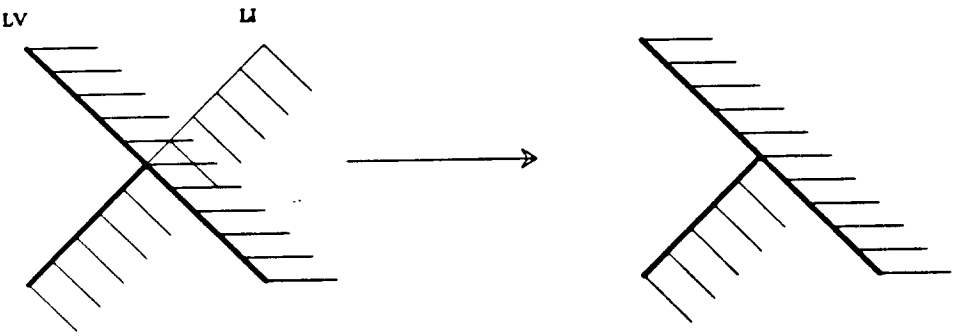


Case 12b

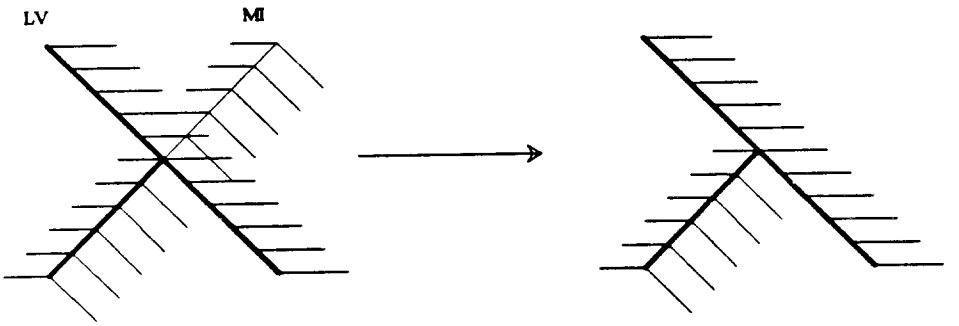
- C1

If the second edge is in front of the polygon visible to the left of the first edge, make the second edge visible. For case C1a, the z component of the face to the left of the first edge is compared to the z component of the second edge at the intersection. If they are the same, then the depth of the ending points of the emerging edge is compared against the old face on the left. If this edge lies in the left face, the slope of the two faces on the right are compared by inserting the x and y coordinates of the endpoint of the first edge into both plane equations. If all these tests yield no definite determination of who is in front, then the faces are in the same plane and the visible face is chosen arbitrarily. If the depth ordering of the two faces is redetermined at a later time in a manner inconsistent with this choice, the faces become logically intersecting and great trouble will result.

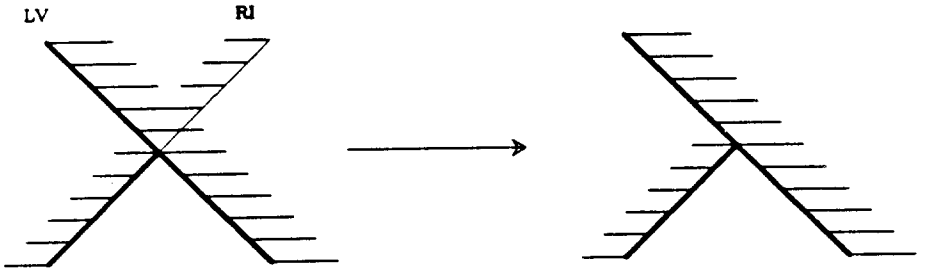
In cases C1b and C1c, the second edge will be visible if the left face pointer is the currently visible face on the left. For case C1c, the visible face between the two edges is determined by scanning the active edge list from left to right until the edge in question is reached. The visible face is chosen by the same set of tests as used to determine a new *RIGHT* edge's right face pointer, i.e. scanning the Active Edge List.



Case C1a



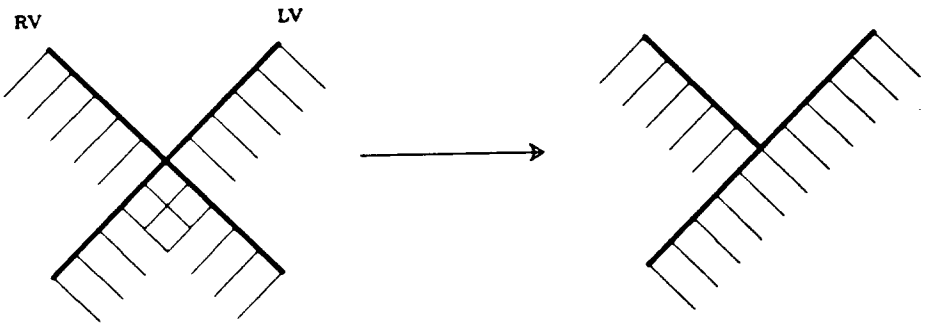
Case C1b



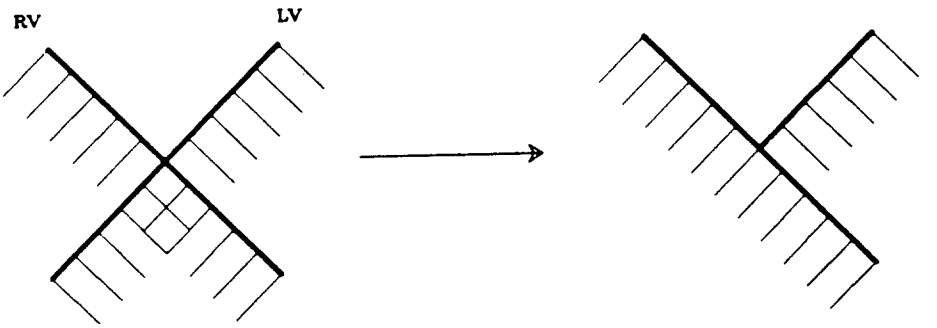
Case C1c

- C2

Compare the depths of the polygons to the left of the first edge and to the right of the second edge. The one with the closer polygon remains visible. The closer polygon is determined by the same set of tests as described under case C1.



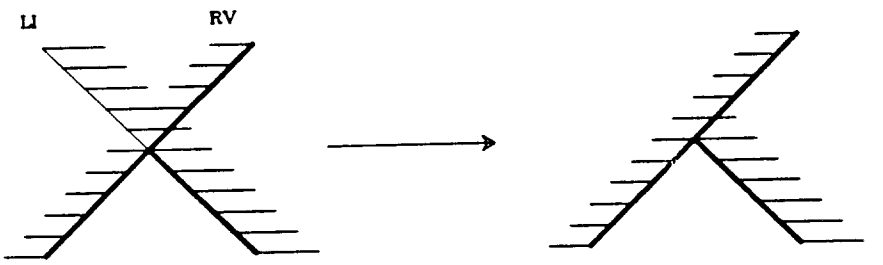
Case C2a



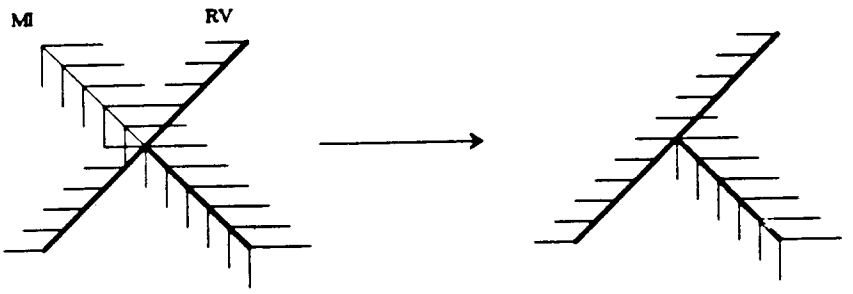
Case C2b

- C3

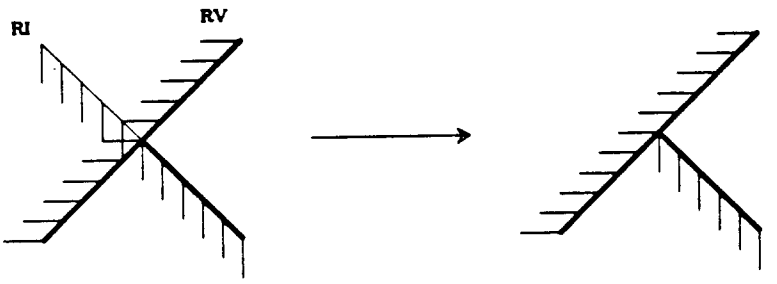
If the first edge is in front of the polygon visible to the right of the second edge, make the first edge visible. The determination of the visible face is done the same way as described under case C1. In case C3c, the face that is visible to the right of the edges is determined by the same procedure described for determining a *RIGHT* edge's right face pointer.



Case C3a



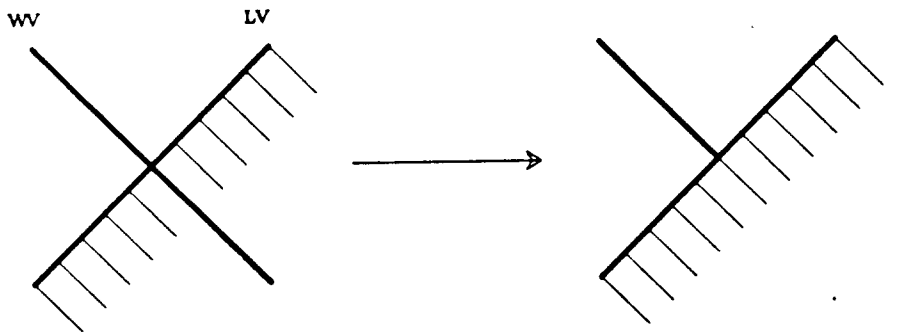
Case C3b



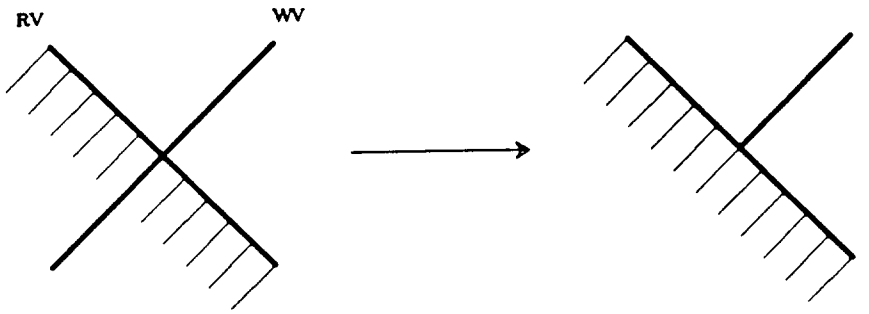
Case C3c

• C4

See if the wire should become invisible by comparing the z coordinate of the wire and the polygon.



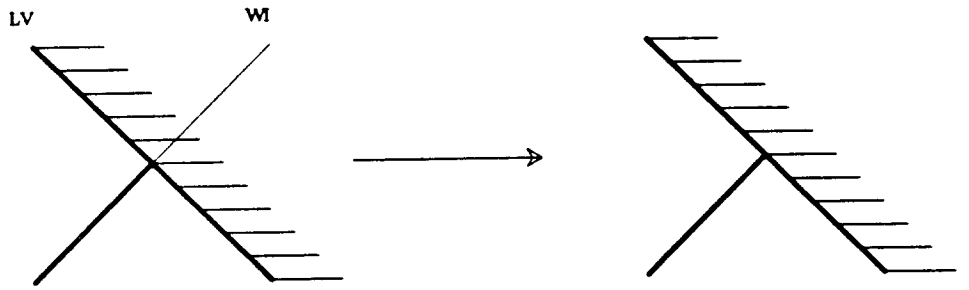
Case C4a



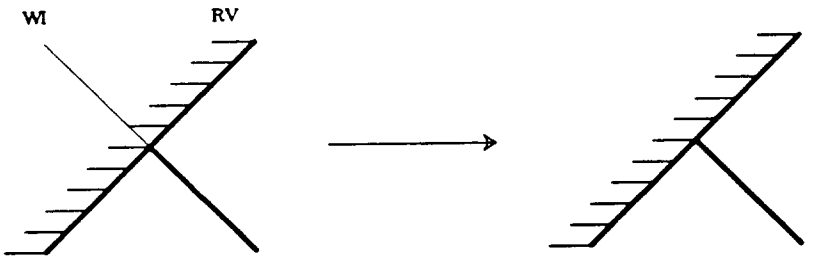
Case C4b

• C5

See if the wire should become visible by comparing the z coordinate of the wire and the corresponding polygon.



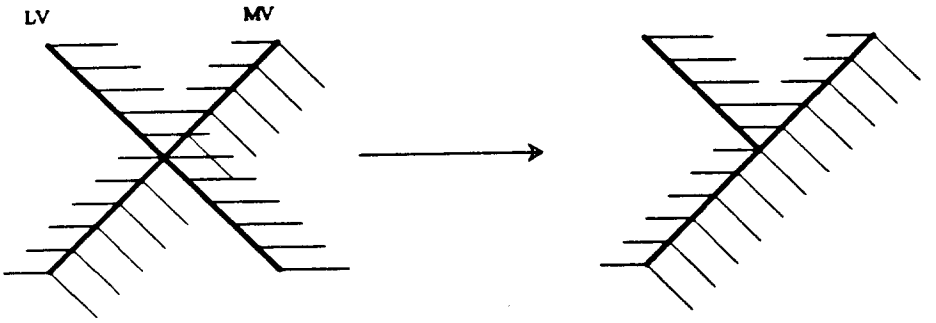
Case c5a



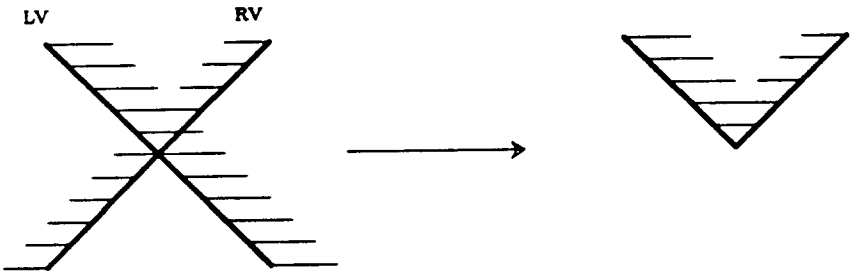
Case C5b

• E

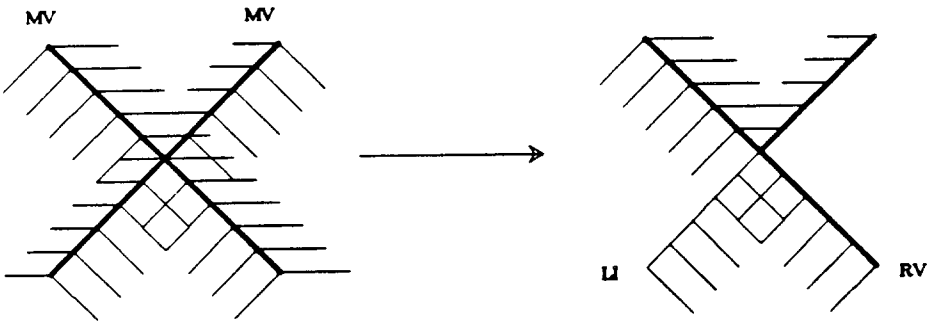
"Error" condition because the edges belong to the same polygon. This can only happen for non-planar self-intersecting polygons that may arise due to floating point calculations for the plane equation or non-planar input faces.



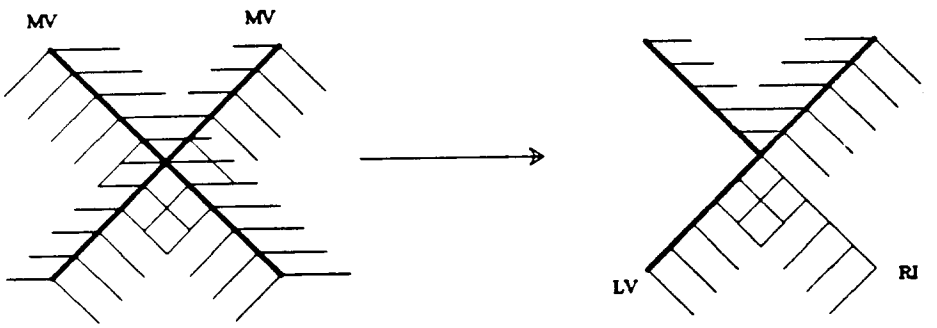
Case Ea



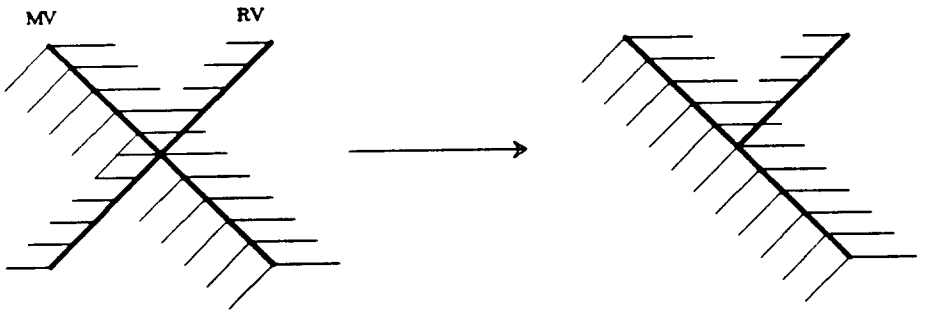
Case Eb



Case Ec1



Case Ed2



Case Ed

Even though, in this manner, something "reasonable" can be done for the display of these polygons, warped polygons may lead to irrecoverable errors when they emerge in their twisted state from behind a contour edge.

For all cases, any edge that is visible after the crossing calculation may need to have its right or left face pointers changed.

4.3.3. Coincidence of points

The scanline algorithm requires that there be an ordering of interesting points from top to bottom and from left to right in the scene. When points have exactly the same X and Y coordinates, an arbitrary ordering is chosen. To avoid ambiguities, the topology is fixed by the order in which the points are processed. The first point processed is considered also to be the geometrical ordering of the points, and edge crossings will have to be introduced correspondingly once the edges diverge in directions incompatible with the original assumption.

In order to have a consistent analysis of the scene, this implied ordering must be carried to all other operations in the analysis. A typical case where several crossing points must be generated for the edges to propagate visibility information correctly is shown in figure 7.

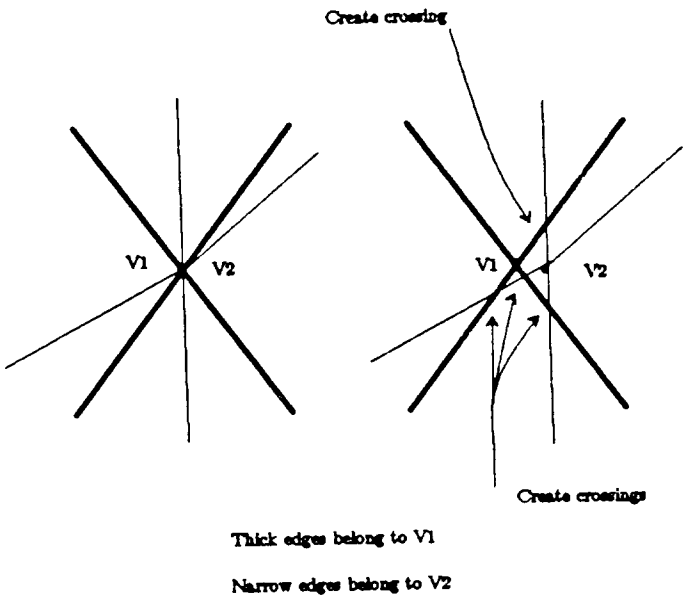


Figure 7. *Treatment of coincident points.*

This is the reason that when a vertex is being analyzed, the edges ending on the vertex must be grouped together or when a crossing point is being analyzed, the crossing edges must first be made adjacent in the *Active Edge List*.

This grouping of edges at vertices and crossing points means that other crossing points will be generated in order for the edges to again be in left to right order. These crossing points will again be at the same X, Y and will be executed as the next interesting point. Note that this approach completely avoids the introduction of any "special case code" for such coincidences.

4.4. Warped Polygons

A particular problem is caused by non-planar faces. Such faces may arise due to the way the scene is originally constructed, or they may result from arithmetic inaccuracies when parts of the scene are subjected to multiple transformations. If such a face is seen almost "edge-on", the projection of its contour may be self-intersecting. This leads to crossings between edges that belong to the same polygon, a situation that Hamlin and Gear [5] labeled as an *error* condition.

However, if this crossing is simply ignored, it leaves the scene description in an inconsistent state from which the algorithm may not recover for a long time. The wrong edges are then flagged visible, leading to incorrect rendering of the scene.

To alleviate this situation and to make the rendering algorithm more robust to such imperfections in the scene description, the plane equations of the faces are calculated by the

formula given by Newman and Sproull [6] which determines a best fit through a set of points. All faces that are close to being viewed edge-on are rejected like backwards-facing ones. The threshold for this rejection can be adjusted; rejecting all faces with z-component values of the normalized face-normal greater than -0.01 has worked well for many "warped" scenes.

In addition, something "reasonable" can be done when such "error" crossings do occur. The edges that cross are reclassified to maintain the consistency of the contour description. This still does not leave a completely consistent description, since such inconsistencies would require additional flags on the edges to propagate this information until the face causing the problem has been completely processed. In order not to burden the algorithm by checking the topological consistency every time when adding or crossing edges, we simply print a warning message when such a warped face is recognized, and then continue as well as we can with the information available in the data structure.

5. VISIBLE POLYGON RECOGNITION

The enhanced edge crossing algorithm detects all visible segments of the original polygon edges. It is thus a natural starting point for the reconstruction of all visible polygons. Visible polygon return consists of two tasks: finding the coordinates of all the corner vertices of the visible polygon pieces, and proper collection and assembly of all the contour pieces of each visible part of a polygon.

Because when we process the scene, we know exactly the points where lines begin, end, and cross, and because we always know the face between every pair of active edges, we can recognize the vertices that make up all visible polygons.

The coordinates of the end-points of all visible edge segments in viewing coordinates can be readily calculated. From the x-y-coordinates of contour edge crossings and from their line equations in viewing space, the viewing-space z-coordinates of the corresponding two points in the line of sight of an edge crossing can be determined. The actual coordinates of these contour points in world coordinates can then be found by subjecting them to the inverse viewing transformation. The world-coordinates of the original vertices can be saved if they are needed later.

The visible parts of polygons are assembled with the same sweep-plane pass that was used to determine visibility of the edge segments. The *Visible Span List* (VSL) (Figure 8) contains an ordered list, from left to right, of all visible spans on the current scan line. No explicit background polygon is introduced. If part of the current scan line is "empty", the corresponding span pointer of the last visible active edge is *nil*. Each visible span element points to the original polygon of which the span is a part and to two ends of one or two vertex chains representing the

partial contour of the corresponding visible polygon. In turn, each visible active edge points to the visible span on the right. Every time a vertex or an edge crossing is encountered by the sweep plane, the VSL is updated in parallel with the updating of the AEL: an entry is made in one or more of the contour vertex chains.

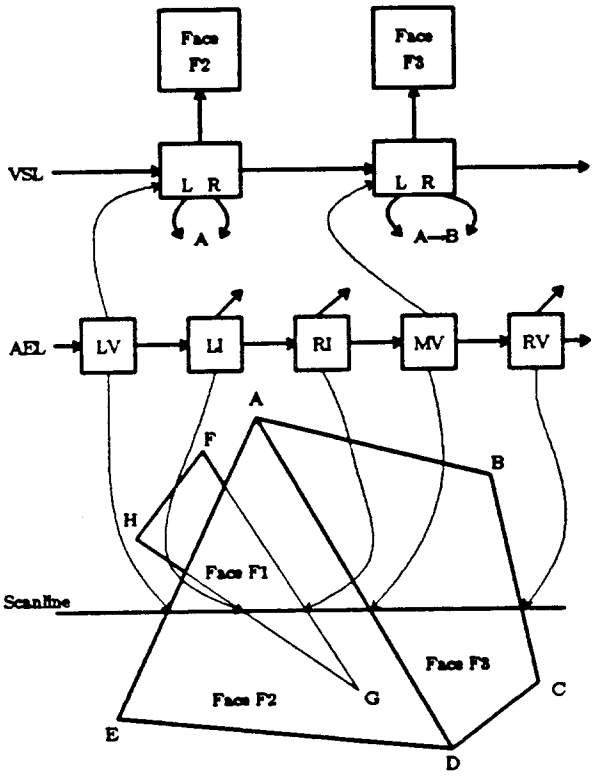


Figure 8. *Relation of Visible Span List and Active Edge List*

The visible polygons under reconstruction are implicitly represented by one or more such chains of vertices. Every time a new span must be inserted into the VSL, e.g., when the vertex that forms the top-most tip of a visible polygon is encountered, a new vertex chain is started. Each such chain has a left and right connection point, pointed to by the visible spans bounded by this contour. When a visible active edge ends in a vertex encountered by the current scan line, the vertex is appended at the proper end of the linked vertex chain(s) for the one (or two) visible polygons that border this edge. When left and right border edges of a visible span merge in a vertex, as is the case at the bottom-most tip of a visible polygon, the corresponding span is removed from the VSL and the left and right border vertex chains are joined with the insertion of the closing vertex. If this action closes off a complete contour, the corresponding contour is transfered to a stack of contours for this polygon, to be output eventually into a file containing all visible polygons.

There are only five distinct routines to handle all possible cases of arbitrarily shaped polygons with multiple contours. These are the basic operations:

Open_face

A new span appears on the VSL. A new vertex chain is started with the L and R pointers of the visible span element pointing to the two ends of this new "chain".

Split_face

This operation is often needed in conjunction with the *Open-face* case when the tip of an object appears in front of a visible polygon. The old polygon span is split into two separate spans with the new visible span element in between, and a new vertex chain is started with the new span element. The two existing pointers of the original span element continue to serve as the L pointer of the new left span and as the R pointer of the new right span. The R pointer of the left span and the L pointer of the right span point to the appropriate ends of the new vertex chain consisting so far of just the one vertex V (Figure 9).

Continue_face

A vertex is encountered on the current scanline but the sequence of spans on the VSL is unchanged. The vertex is appended to the proper one or two vertex chains, and the pointer to the visible span on the right is readjusted (Figure 10).

Close_face

A span disappears from the VSL. The two ends of the vertex chains representing its left and right border are joined. If those two ends belonged to the same chain, the completed, closed-off contour is saved on a contour stack for the corresponding face.

Merge_face

This occurs often in conjunction with the *Close-face* case. The two spans on either side of the disappearing span may belong to the same polygon and must therefore be merged into a single span. This is done by merging the two inner borders of the left and the right span and retaining only the L pointer of the left span and the R pointer of the right span to serve as the two new pointers of the merged visible span element. If this merger joins ends that belong to the same vertex chain, a closed hole contour is formed, which is then transferred to the contour stack for this particular face (Figure 11).

Often a visible edge segment acts as a border for two polygons. When a vertex is encountered, it becomes part of two separate contours of visible polygons. Two calls to some of the above routines may then be used to process the polygons bordering at that vertex. If the two faces do not physically share an edge, i.e. if the latter is a contour edge, then a physical vertex on this contour can belong to only one face. A new vertex in the same line of sight must then be created

for the polygon that lies behind the contour edge. This vertex has the same x-y-coordinates in viewing space, but a new z-value derived from the plane equation of the polygon to which it belongs. In some cases, e.g., when two contour edges overlap in front of a third polygon, *two* extra vertices need to be generated.

5.1. Illustration of Some Typical Cases

To enhance understanding of our approach, we will illustrate how these routines are applied in a few typical situations.

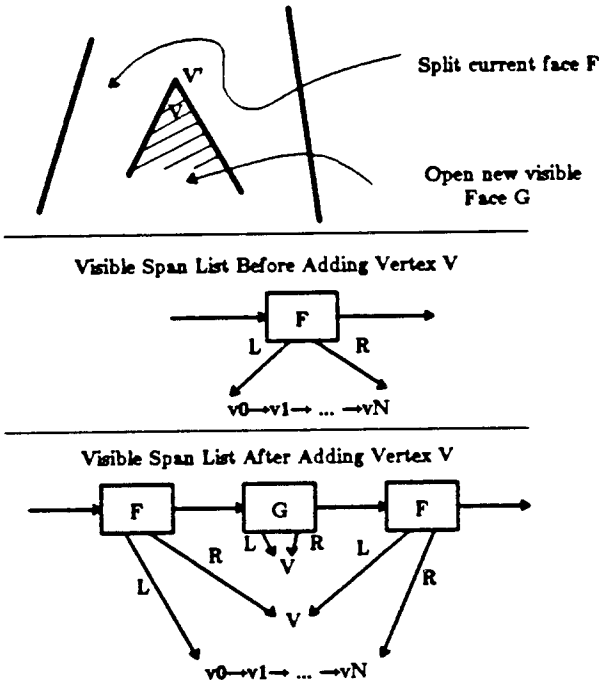


Figure 9. *New face starts in front of another one.*

The first example (Figure 9) shows the case of a new face G starting in front of another face F. The masked face F is split into two parts with the procedure *Split_face* and is now represented by two separate visible span elements on the current scan line. Two new contours are started with *Open_face*, one involving the corner V of the new face, and the other starting with the projection V' of this vertex onto the plane of the original face F. The latter contour will become a hole in face F.

The same basic situation rotated counter-clockwise by 90 degrees is shown in Figure 10. However, the routines involved in processing this case are now *Continue_face* because of the different orientation. The sequence of visible span elements on the scan line does not change as this vertex is passed. The vertex and its projection onto the face behind are simply added to the

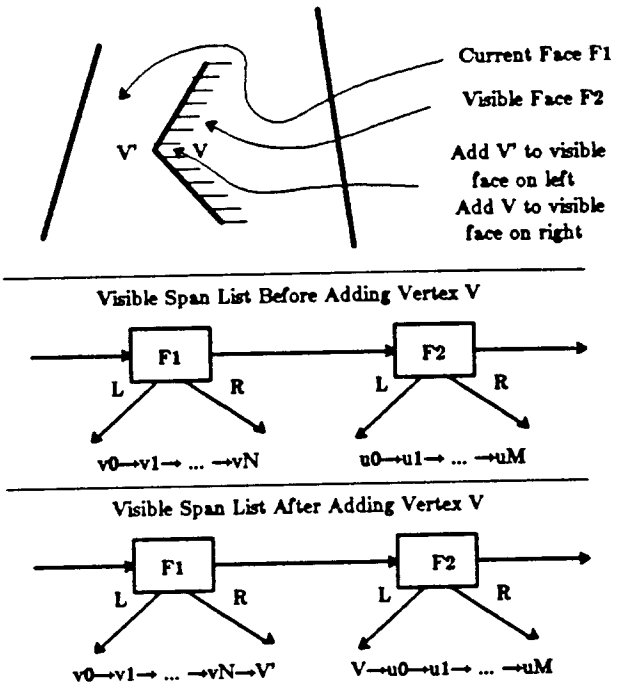


Figure 10. *Corner in a contour edge.*

respective contours, and some pointers are redirected.

A third case is shown in Figure 11. This time we assume that the scan-line approaches the end of a hole in face F. The two spans that represented different parts of face F need to be merged into a single span with *Merge_face*, while the hole contour gets closed off using *Close_face*. The hole contour disappears from the VSL and is transfered to a stack of finished contours for face F. If there is another polygon G behind F, visible through the hole, then its visible contour gets closed at the same time and is also transfered to its output stack.

A last case is shown in Figure 12. In this case two contour edges overlap. At the crossing location of the two edges the contour of face F is unaffected; no new vertex need be added to its contour vertex chain. But face G will get a new vertex corresponding to the projection of V. The background polygon behind the two faces F and G, if there is one, will be terminated at this point. It requires yet another copy of the vertex V in its own plane.

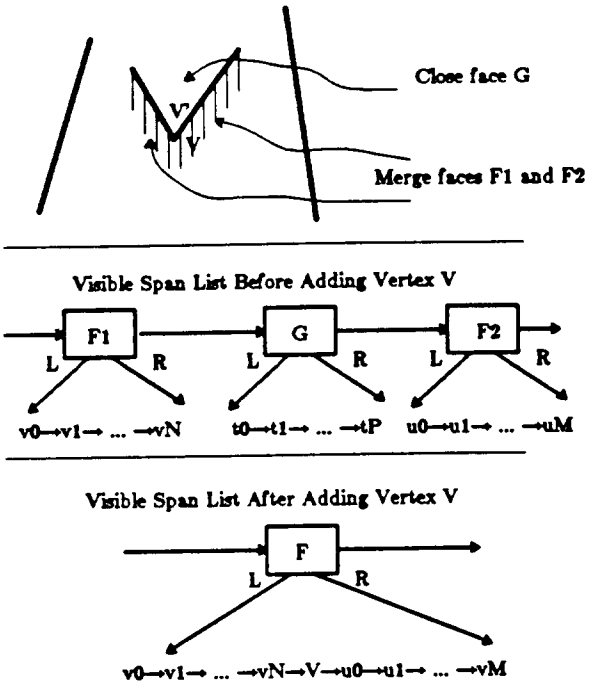


Figure 11. *End of a hole contour.*

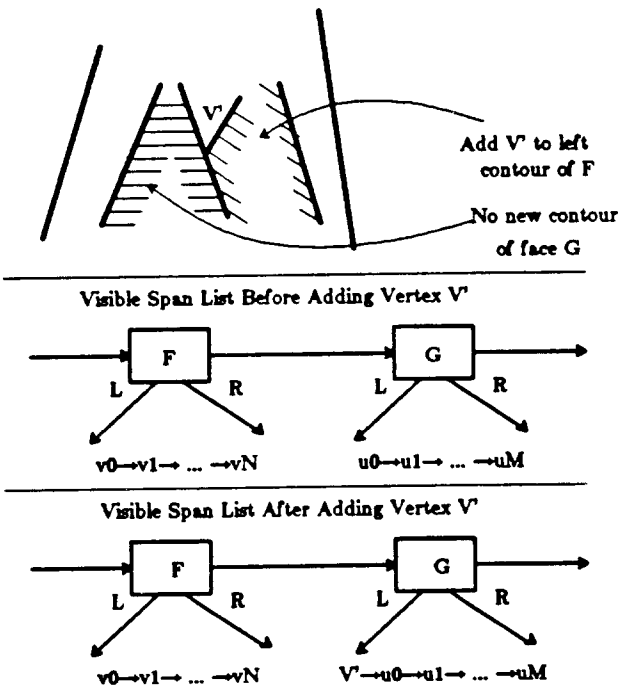
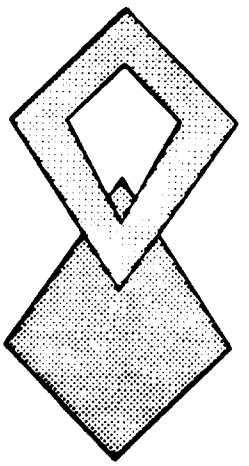


Figure 12. *Contour intersection.*

As an example, consider the picture and UNIGRAFIX file:



```
v ua  -10.  10.  0 ;
v ub  0.  20.  0 ;
v uc  10.  10.  0 ;
v ud  0.  -5.  0 ;

v ha  -5.  10.  0 ;
v hb  0.  15  0 ;
v hc  5.  10.  0 ;
v hd  0.  1.  0 ;

v la  -10.  -10.  5 ;
v lb      0.  5.  10 ;
v lc  10.  -10.  15 ;
v ld  0.  -20  10 ;

f u ( uc ud ua ub )( hd hc hb ha );
f l ( la lb lc ld );
```

The generated UNIGRAFIX file returning just visible faces is:

```
v uc  10.  10.  0 ;
v ud  0.  -5.  0 ;
v ua  -10.  10.  0 ;
v ub  0.  20.  0 ;
v hd  0.  1.  0 ;
v hc  5.  10.  0 ;
v hb  0.  15  0 ;
v ha  -5.  10.  0 ;
f u ( uc ud ua ub  )( hd hc hb ha  );
v lc  10.  -10.  15 ;
v ld  0.  -20  10 ;
v la  -10.  -10.  5 ;
v X#2  -3.3333333333  0.  8.3333333333 ;
v l:ud    0.  -5.  10. ;
v X#3  3.3333333333  0.  11.6666666667 ;
v l:X#1  1.2121212121  3.1818181818  10.6060606061 ;
v l:X#0  -1.2121212121  3.1818181818  9.3939393939 ;
v lb      0.  5.  10 ;
f l ( lc ld la X#2 l:ud X#3  )( l:X#1 l:hd l:X#0 lb  );
```

The vertices starting with **X#** are generated from the crossing of two edges of two visible polygons. A vertex beginning with a face name, **F**, followed by a ":" and then a vertex name, **V**, is the generated vertex that corresponds the vertex **V** projected onto face **F** .

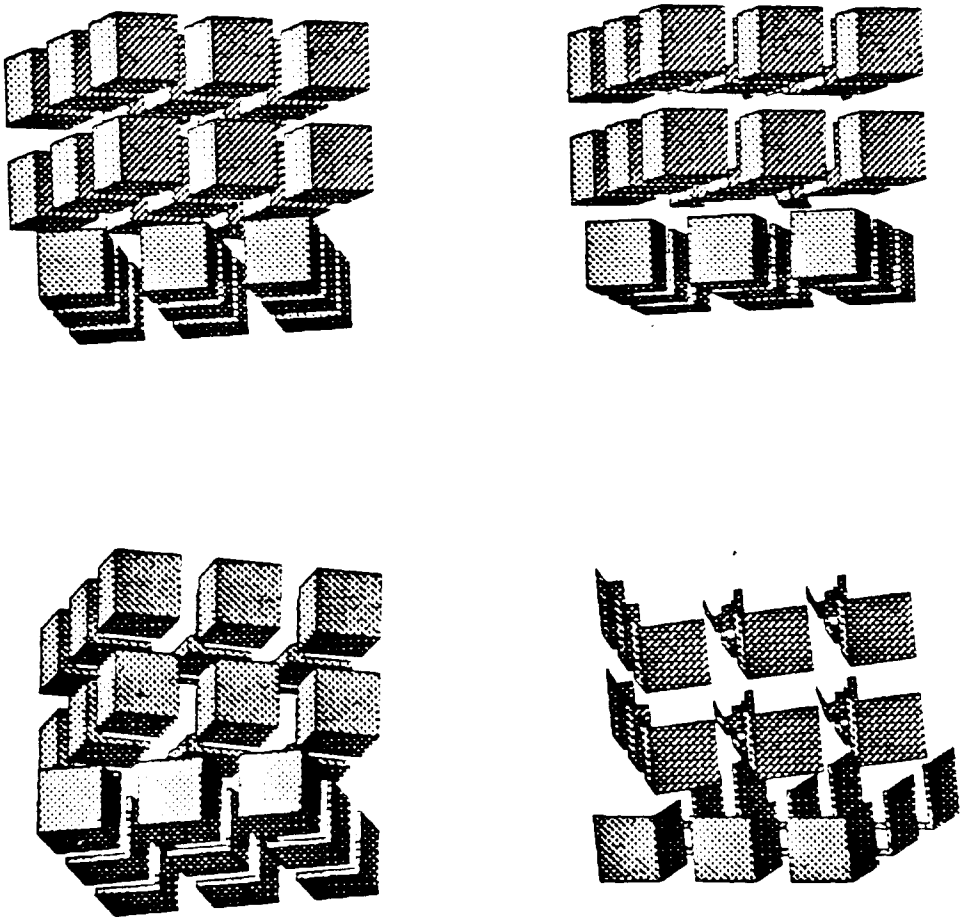


Figure 12. *Visible polygon return in world coordinates.*

Figure 12 shows the result of a visible polygon return in world coordinates. The scene consisting of only the visible polygons is then rendered four times, first from the original view point from where the visible polygons were determined, and then three more times from slightly different angles to demonstrate that the polygons are indeed properly clipped.

6. OUTPUT OF LINES, SPANS AND CONVEX POLYGONS

For line drawings, and when adding borders to shaded faces, every time a visible active edge is removed from the active edge list or a visible edge becomes invisible, the line from the starting point to the ending point can be output. Obviously, when an invisible edge becomes visible, the edge's starting point must be suitably modified to correspond to the point at which it became visible for correct output of the line.

After all vertices and crossing points in the current scanline have been analyzed, the spans and lines on the scanline can be generated from the active edge list. If shaded faces are being displayed, then the active edge list is scanned from left to right, looking for pairs of visible active edges. For each pair found, the span between them is output with the color determined by the face between them. The X coordinates of the span are determined by keeping a current value of X for each active edge. As the active edge list is scanned, a delta X value is added to the current value of X, which corresonds to the intersection of the active edge with the bottom of the output scanline, ensuring that the X value never extends beyond the bottom of the edge.

For devices that can accept higher level output primitives such as convex polygons, the visible polygon return gives the opportunity to speed up the rendering process. With a flag, output of visible elements can be restricted to triangles and trapezoids. Associated with each visible span are the X coordinates of the face contour on the left and right, as well as the corresponding Y value at the last interesting point in that face. When a face begins to be visible, these two X values are set to be the X coordinate at the top of the face, the Y value is set to the corresponding Y value of the point and a flag is set signaling that the next output will be a triangle. At the processing of each subsequent vertex or crossing point, a trapezoid can be output that has on the top the saved X and Y values and a base that has a point corresponding to the vertex or cross and the other X value found from the active edge to the right or left of the point, depending upon which side of the face the vertex lies. When a face becomes invisible or disappears, a triangle may be output that has at the top the saved X and Y values, and at the bottom the single closing point.

Unfortunately, since the generation of the polygons is now directly controlled by the output device, the order in which the polygons are output can affect the final picture because for most devices if a pixel is covered by the polygon, then it will be colored. Thus on the border of two polygons, where the intersection lies within the area of the pixel, the order in which the polygons are drawn can affect the final color of the pixel. For this reason, the borders of the polygons drawn in this fashion can "waver" by one pixel, depending upon the bordering polygons. Figure 13 below shows a generated picture and the resulting output of convex polygons that would result if rendered on a suitable device.[7]

Clipping of the lines and faces is accomplished by adding a *clipping face* to the scene that has a hole the size of the output device and extends infinitely outward in all directions. Clipping is then done automatically by the scanline algorithm.

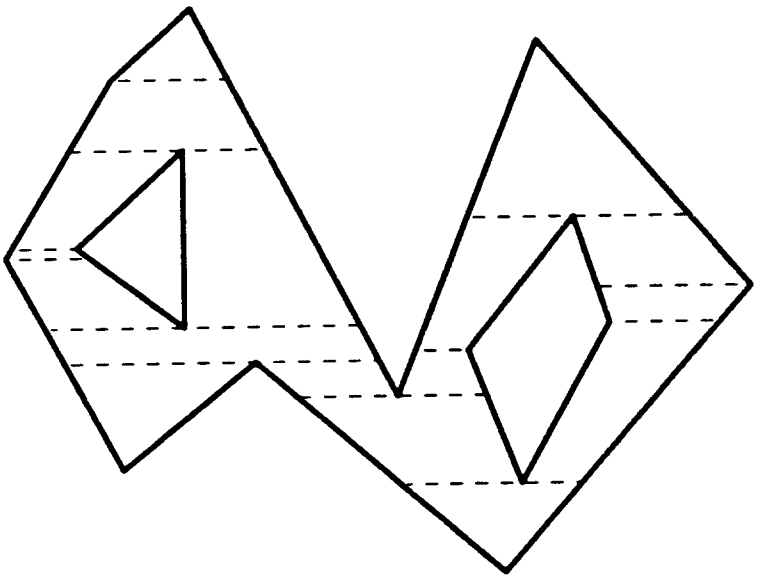


Figure 13. *Illustration of convex polygon output.*

7. CONCLUSIONS

The implementation of the extended cross algorithm, able to handle polygons with multiple contours and holes, to the point where it gives clean results of practically all legal UNIGRAFIX scene descriptions, turned out to be much harder than expected at the outset of this work. This algorithm, because it carries so much information from previously analyzed parts of the scene to later parts, is particularly unforgiving with respect to attaching wrong visibility information to edges. A single error based on an ambiguous situation can carry its consequences through the whole rest of the scene.

The possibility of encountering some "inconsistency" in scene descriptions is always present. Slightly non-planar polygons, when seen almost edge-on, can lead to self-intersecting contour projections. Two objects touching each other may result in intersections of faces, which, even though they slice off only a tiny sliver of a face, can nevertheless confuse the algorithm. Thus the algorithm needs some extra checks and built-in tolerances to give it the robustness required of a general purpose rendering algorithm

On the other hand, this approach has some very attractive features. It leads directly to line drawings of objects with hidden faces removed, and in this mode it is particularly fast and exceeds in rendering speed even the rather fast algorithm used in UNIGRAFIX 1.

In addition, the new hidden surface algorithm gives rise to a number of interesting extensions that were not easily possible with the original algorithm. Current research is involved with the generation of shadowed pictures. It is well known that the methods involved in hidden

surface removal are the same ones that can be used to determine the illuminated and shadowed portions of a scene. Our method of recognizing visible polygons is ideal for the splitting of an original scene into the visible and invisible portions. To generate a picture with shadows, one sets the eyepoint of the view to be the lighting direction and generates a file with both invisible and visible portions of the polygons, suitable marked for the current illumination.

Thus, in spite of the difficulties that the implementation of this algorithm caused, we consider it a good choice for the high-resolution rendering of polyhedral geometrical objects. A consistent description of such objects contains a lot of valuable information that can be exploited in the elimination of hidden features with a significant gain in rendering speed.

References

1. C.H. Sequin and P.S. Strauss, "UNIGRAFIX," *Proc. 20th Design Automation Conf.*, pp. 374-381, Miami Beach, FL, June 1983.

2. A. Appel, "The Notion of Quantitative Invisibility and the Machine Rendering of Solids," *Proc. ACM*, pp. 387-393, National Conference, 1967.

3. P. P. Loutrel, "A Solution to the Hidden-Line Problem for Computer-Drawn Polyhedra," *IEEE Trans. on Computers*, vol. C-19, no. 2, p. 205, March 1970.

4. R. Galimberti and U. Montanari, "An Algorithm for Hidden-Line Elimination," *Comm. ACM*, vol. 12, no. 4, p. 206, April 1969.

5. G. Hamlin and C.W. Gear, "Raster-Scan Hidden Surface Algorithm Techniques," *Computer Graphics*, vol. 11, no. 2, pp. 264-271, Summer 1977.

6. W.M. Newman and R.F Sproull, "Plane Equations," in *Principles of Interactive Computer Graphics, 2nd Edition*, p. 499, McGraw-Hill, New York, 1979.

7. M.E. Newell and C.H. Sequin, "The Inside Story on Self-intersecting Polygons," *Lambda Magazine of VLSI Design*, vol. 1, no. 2, pp. 20-24, May 1980.