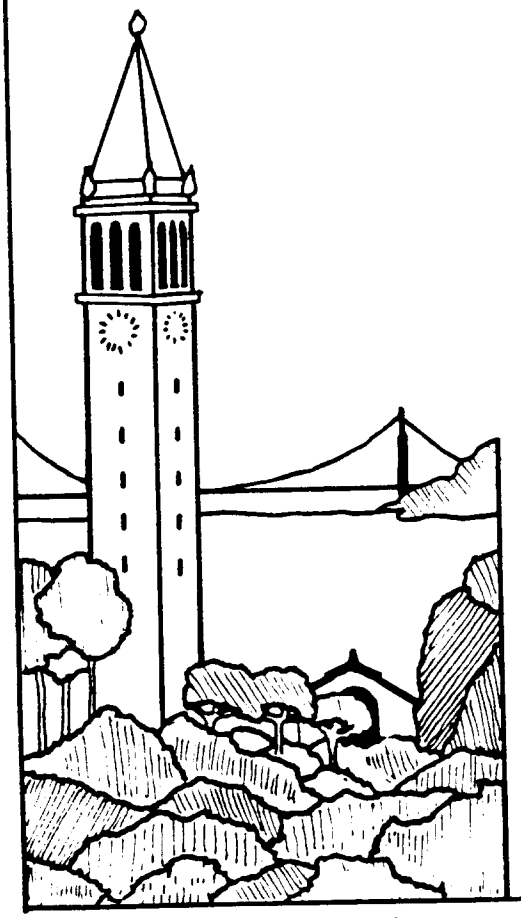
Hidden Feature Removal and Display of Intersecting Objects in UNIGRAFIX

Nachshon Gal



Report No. UCB/CSD 86/282

January 1986

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Hidden Feature Removal and Display of Intersecting Objects in UNIGRAFIX

Nachshon Gal

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, California 94720
U.S.A.

ABSTRACT

UNIGRAFIX is a graphics modeling and 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.

A new hidden feature algorithm in UNIGRAFIX was inspired by Hamlin and Gear's STACK algorithm. It is a scan-line, image space algorithm that exploits depth ordering of faces to produce the visible spans for each scanline. Detection and correct display of intersecting objects is efficiently achieved by checking *only* visible faces, and comparing each one with a minimal set of potential penetrators.

The resulting rendering program also features smooth shading of faces. It is robust and fast, and this makes it a good choice for interactive design where other renderers might fail or take too much time.

† UNIX is a trademark of Bell Laboratories

Acknowledgments

This work could never have been completed without the continuous support and encouragement of my wife Michal, and it is dedicated to her.

Professor Carlo H. Séquin, my research advisor, provided me with a very generous and pleasant work environment. He was always able to restate my vague descriptions of my algorithms in a way that gave me new insight into them, and I thank him for that. Mark Segal was invaluable in help and advice, and never complained when I made him again and again dig into old code to explain why and how. Eric Bier, H.B. Siegel and Paul Wensley also helped a lot, and Professor Larry Rowe agreed to read this work on a very short notice. I thank them all.

This work was supported by the Semiconductor Research Corporation.

Contents

Page

1. Introduction 1

2. The Hidden Surface Algorithm 4

2.1. Static Data Structure 4

2.1.1. Vertices 5

2.1.2. Edges 5

2.1.3. Faces 5

2.1.4. Wires 5

2.2. Terminology and Overview of Rendering Process 5

2.3. Dynamic Data Structure 8

2.3.1. Active Edges 8

2.3.2. Active Faces 10

2.4. Preparing the Next Scanline 11

2.4.1. Adding a New Vertex to the AEL 12

2.5. The **STACK** Algorithm 14

2.5.1. Processing the leftFaces 16

2.5.2. Processing the rightFaces 17

2.5.3. Producing Output 18

2.5.4. Coherence and Depth Comparisons 18

2.5.5. Examples 21

2.5.6. Optimizations 27

3. Display of Intersecting Faces 29

3.1. Modifications to the **STACK** algorithm 29

3.2. Detecting Intersections 30

3.2.1. Adding a Potential Penetrator to the PPL of f_1 31

3.2.2. Finding if f_1 is Penetrated 34

3.3. Incorporating the Intersection into the Data Structure 35

3.4. Special Processing of an Intersection Edge 36

4. Border Enhancement 39

4.1. Crossing Edges 44
4.1.1. Finding the Crossing Point 44
4.1.2. Cutting Edges 45
4.2. Display of Borders 46
4.3. Borders with Intersecting Objects 49
5. Smooth Shading and Embedded Text 52
5.1. Smooth Shading 52
5.1.1. Smooth Shading with Intersections 54
5.1.2. Edge Enhancement 54
5.2. Embedded Text 55
6. Evaluation and Conclusions 58
6.1. Run-Time Statistics 58
6.1.1. Comparisons with Other Renderers 58
6.1.2. Evaluation of Intersection Detection 60
6.2. Conclusions 63
References 65
Appendix A. The UNIGRAFIX Language 67
Appendix B. The UGDISP manual 68

1

Introduction

The UNIGRAFIX system already has two different scanline based renderers, both with different strengths and weaknesses. A third renderer has been implemented which tries to combine as many of the strengths of the previous two algorithms, and to add some new features such as handling intersecting objects and providing smooth shading.

The first renderer, *UGSHOW* [10], is a scanline algorithm that performs in image space and relies on stacks of faces. Edges that cross the current scanline are kept in an X sorted *Active Edge List* (AEL), and faces are implicitly described by their left and right bounding edges. Each *span* between two adjacent edges in the AEL has a depth sorted stack of all the faces that lie under it. The top face is the visible face in that span. The face stacks are retained from one scanline to the next. They are updated whenever old edges leave the AEL, new edges are inserted, or any two edges swap places. UGSHOW is a robust algorithm and copes successfully with scene inconsistencies such as warped faces and coinciding edges. It does however suffer from big (and at times huge) dynamic space requirements: All the faces that are touched by the sweep plane are represented in the face stacks, and there are such stacks for all edges in the AEL.

The second renderer, *UGPLOT* [18], is an edge-intersection scanline algorithm, following the CROSS algorithm by Hamlin and Gear [3]. UGPLOT employs a more refined data structure than that of UGSHOW and extracts valuable topological information from the scene description (written in the UNIGRAFIX language [16]). It also maintains an active edge list, but limits the search for obscuring faces to the faces that are bordered by the neighboring edges in the AEL. Visibility information is passed from edges to vertices and from them to other edges. The core of the algorithm is a set of rules that define how visibility should be updated at edge crossings. Since the major computation is done at "interesting y-locations" i.e., vertices and edge crossings, the output is not bound to physical device resolution; the algorithm is capable of recognizing the visible polygons in the two-

dimensional image plane, as well as in three-dimensional object coordinates [13]. UGPLOT is a fast renderer, and the visible polygon return enables it to exploit new display devices that scan-convert polygons in hardware. However, since all visibility calculations are incremental, it is very sensitive to minor topological inconsistencies, and errors due to numerical inaccuracies may propagate far into other portions of the picture.

UNIGRAFIX serves research areas such as Computer Graphics, Geometric Modeling [11,14,15,5] and Topology [9], as well as purely artistic endeavors by people. Scenes are thus not limited to well behaved polyhedral solids; users often take the freedom to create any imaginary scene that can be described with the UNIGRAFIX primitives. Thus a need developed for a robust, yet still efficient, renderer that would try to do its best to give a decent picture even when the scene is not completely legal. Some "illegalities" are quite abundant and should be displayed correctly; among those are coinciding faces, warped faces, self intersecting contours, and most important for any interactive design - intersecting faces. Many hidden surface algorithms restrict themselves to non-intersecting objects, and require a special pre-processor to remove all intersections [9]. This is a reasonable demand, and it leads to efficient rendering when the same scene is rendered many times. But the price of pre-processing may be too high for interactive design which involves many intermediate scenes. A rendering algorithm that detects only the visible intersection lines in a scene is then much faster than the repeated sequence of pre-processing and display.

The new hidden surface algorithm, *UGDISP*, tries to take on those challenges. It follows the STACK algorithm by Hamlin and Gear [3], but since UNIGRAFIX objects are not as constrained as their objects (convex polygons that do not intersect and do not overlap one another cyclically), it is more sophisticated than the original elegant idea. A single stack of visible faces is built, maintained and finally discarded on each scanline. The basic algorithm employs scene and scanline coherence to achieve fast rendering of shaded faces. Special care is given to edge enhancement (correct enhancement of horizontal edges turned out to be the most difficult task in the project). Command-line options tell the program to detect and display face intersections and to handle warped faces. Intersection lines are found by testing each visible face against a minimal set of potentially penetrating faces. When an intersection is found, the data structure is modified to reflect the "cut" in the current visible face such that it looks like a regular edge on the next scanlines.

Chapter 2 describes the data structures and the new STACK algorithm for shaded faces with hidden surface removal. Chapter 3 describes the algorithm for intersection detection. Chapter 4 covers the extra work that is required to achieve border enhancement. Chapter 5 briefly discusses two useful extensions to the STACK algorithm: *Gouraud* shading and embedding text in the picture. Chapter 6 evaluates the program performance and draws conclusions about this work.

2

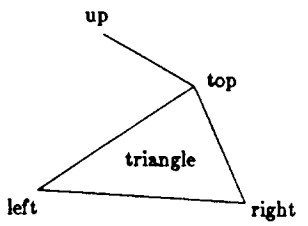
The Hidden Surface Algorithm

This chapter describes the data structures used by the program, and the hidden surface algorithm in its simplest form, i.e., without border enhancement and without intersections checks.

2.1. Static Data Structure

The primitives of the UNIGRAFIX language are *vertices*, *faces* and *wires*. A vertex statement consists of a unique identifier and *x,y,z* coordinates. A face statement has an optional identifier and a list of *contours*, each of which is a list of vertex identifiers. *Edges* are *not* part of the language but are implicitly introduced by any pair of adjacent vertices in the face statement, including the last and first vertex in every contour. A wire statement is similar to a face statement, except that no edge is implied between the last and first vertices of a contour.
As an example, consider the UNIGRAFIX file:

```
v top   0 1 0;
v right 1 0 0;
v left -1 0 0;
v up    0 1 1;
f triangle (top right left);
w       (top up);
```



For further details on definitions, arrays and instances see Appendix A and refer to the UNIGRAFIX Manual [16].

An internal representation of the scene is constructed while reading the scene file. This structure has multiple cross references between the primitive elements, and it efficiently encodes all data and topology that can be extracted from the ascii representation. In this section I describe the *static* data structure. Once

built, it can be re-used for multiple renderings with varying viewing parameters and options (See [1] on a system that makes effective use of this fact). Figure 2.1 shows the basic elements in the structure (for simplification, global linked lists are not shown in the figure).

2.1.1. Vertices

Each vertex is represented by a *VERTEX* structure which contains a pointer to its string identifier, its coordinates, and a pointer to a list of all edges that use this vertex. Additional fields hold certain flags and an illumination value (if smooth shading is performed). The vertex coordinates are either in *world* coordinate system or in *viewing* coordinate system, depending on the stage of the renderer. All vertices are linked in the order they appear in the original file to allow easy access for various general operations. Since faces and wires refer to vertices by name, we need a fast name lookup; therefore the same VERTEX structures are also linked in hash table entries, with the key being the hashed value of the unique identifier.

2.1.2. Edges

Each pair of adjacent vertices in face or wire statements is represented by an *EDGE* structure which contains pointers to the VERTEX structures of its two end-vertices, and a list of all faces and wires that make use of this edge. Edges are unique, so all those faces and wires will use the same edge. The EDGE is a major junction in the data structure between vertices on one hand, and faces/wires on the other hand.

2.1.3. Faces

Each face is represented by a *FACE* structure which contains an optional string identifier, a list of contours (each contour being a list of edges), and plane equations of the face in world and in viewing coordinates. Additional fields hold color and illumination data and flags. All FACE structures are linked in a global list in the order they appear in the original file.

2.1.4. Wires

Each wire is represented by a *WIRE* structure which is similar to the FACE except for the illumination and plane equations which do not apply to wires.

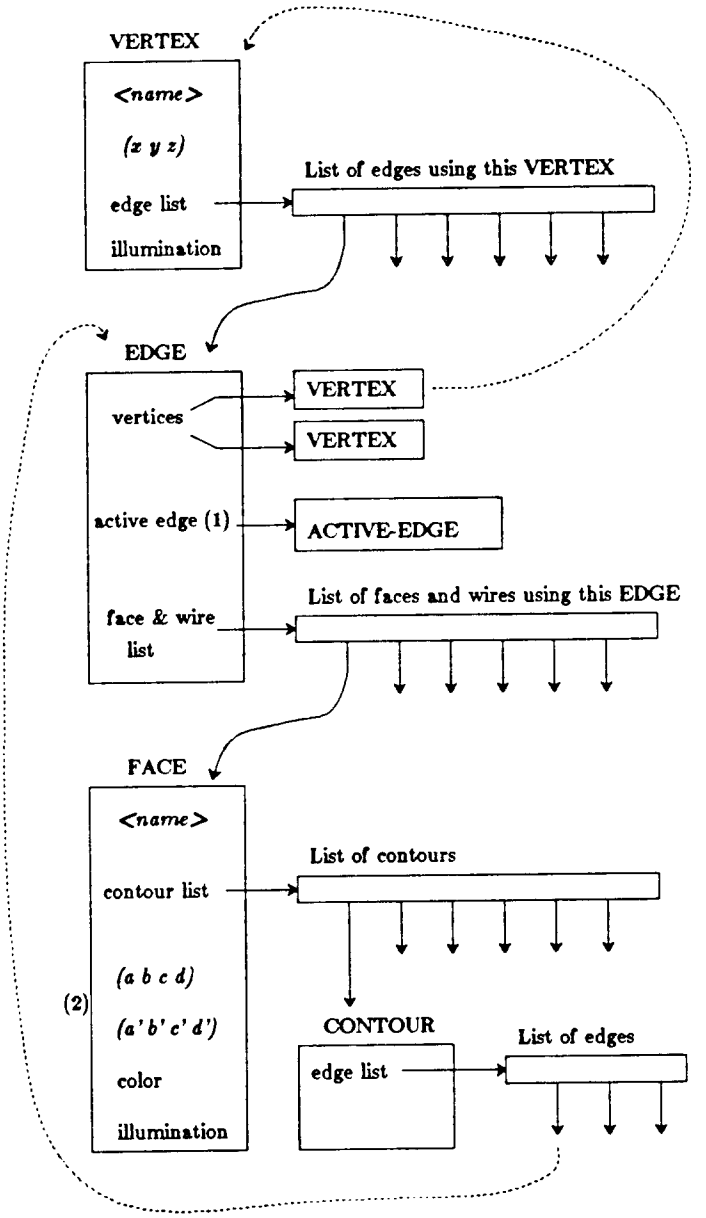


Figure 2.1. *Static Data Structure.*

(1) When the edge is a member of the Active Edge List, this is a pointer to the corresponding active structure (see section 2.3).

(2) The plane equation is represented by its four coefficients. (a' b' c' d') is the plane equation in viewing coordinates.

2.2. Terminology and Overview of Rendering Process

UGDISP starts by reading the command line parameters and options (see Appendix B), and calculating the *viewing transformation matrix*. It then reads the input file and builds the static data structure as described above. All vertices are transformed by the viewing matrix to viewing coordinates, and *bucket sorted* by the Y coordinate into buckets, one for each scanline. Within each bucket the sort is first by X and then by Y. Plane equations of all faces are transformed, and faces with normals pointing away from the viewer are culled out (unless specifically requested by the $-ab$ view option to keep them). Illumination (shading) values for faces are calculated by summing up the contributions of all light sources. Each ambient source contributes its full intensity, and each directional sources contributes its intensity factored by the dot product between its direction and the face normal; if the dot product is positive (i.e., light reaches that face), then the intensity is multiplied by it, otherwise the contribution is zero.

Now let's imagine a horizontal plane that sequentially cuts through the scene image at each scanline from top to bottom. This plane defines *segments* where it intersects faces†. Our task is to decide which segments are visible (fully or partially), and to generate their correct projection onto the output scanline. (See [17] on Segment Comparisons and Scanline Algorithms). This task is carried out, starting with the topmost non-empty bucket and going down until the lowest non-empty bucket.

The *Active Edge List* (AEL) is the list of all edges that intersect the *current* scanline. It is kept sorted by the intersection points from left to right. The current scanline is the line being processed now, and the *previous* scanline is the one just above the current. Processing line y means dealing with everything that happens in the *swath* between $y+1$ and y (However, the output for this line will reflect the visible faces at exactly y, so faces that ended on this swath are not represented in the output).

Processing a scanline is done in two major steps: updating the AEL, and running the STACK algorithm on it. Output is produced during the second step. It should be noted that the output commands of the algorithm are device independent (except for the obvious dependence on device resolution), and actual

† Concave faces and faces with multiple contours may have more than one segment in a single scanline.

output will be produced in different forms for different devices (e.g., monotonous raster scan for simple raster plotters, shaded polygons for smarter devices, etc.).

2.3. Dynamic Data Structure

The dynamic structure is maintained as long as there are vertices in the array of buckets. Its skeleton is the AEL, which is a list of *ACTIVE-EDGE* structures. The list is doubly linked to allow easy movement in both directions, and easy insertion and deletion of entries. Figure 2.2 shows the main components of the list.

2.3.1. Active Edges

Each ACTIVE-EDGE represents an EDGE from the static data structure that intersects the current scanline. It contains the following information:

- *fromEdge* is a pointer to the corresponding static edge.
- *vTop* is a pointer to the top vertex of the edge.
- *vBot* is a pointer to the bottom vertex of the edge.
- *slope* is $\Delta x/\Delta y$ i.e., the increment in X for one scanline.
- *xPrev*, *xCur* are the X intersections of the edge with previous and current scanlines.
- *rightFaces* is a pointer to a list of *FACE-START* entries, each corresponding to a face that spans to the right of the edge (and therefore bordered by our edge from the left).
- *leftFaces* is a pointer to a similar list of *FACE-END* entries, each corresponding to a face that spans to the left.
- *rightFacesPointer* (not shown in figure 2.2) points initially to the beginning of the *rightFaces* list, and during the STACK algorithm moves along that list. See 2.5.2 for its use.

At this point it may seem that storing the two vertices is redundant once we have the pointer to the edge, but in chapter 4 we shall see that an active-edge may correspond to a *partial* piece of the original edge, so the top and bottom vertices would not necessarily be the end vertices of the edge.

The lists of faces are sorted in an *open book* manner, as if the faces were pages in a book and the page numbers were used as the sort key. That means that in the *leftFaces* list the first item is bottom-most face, and the last item is the topmost

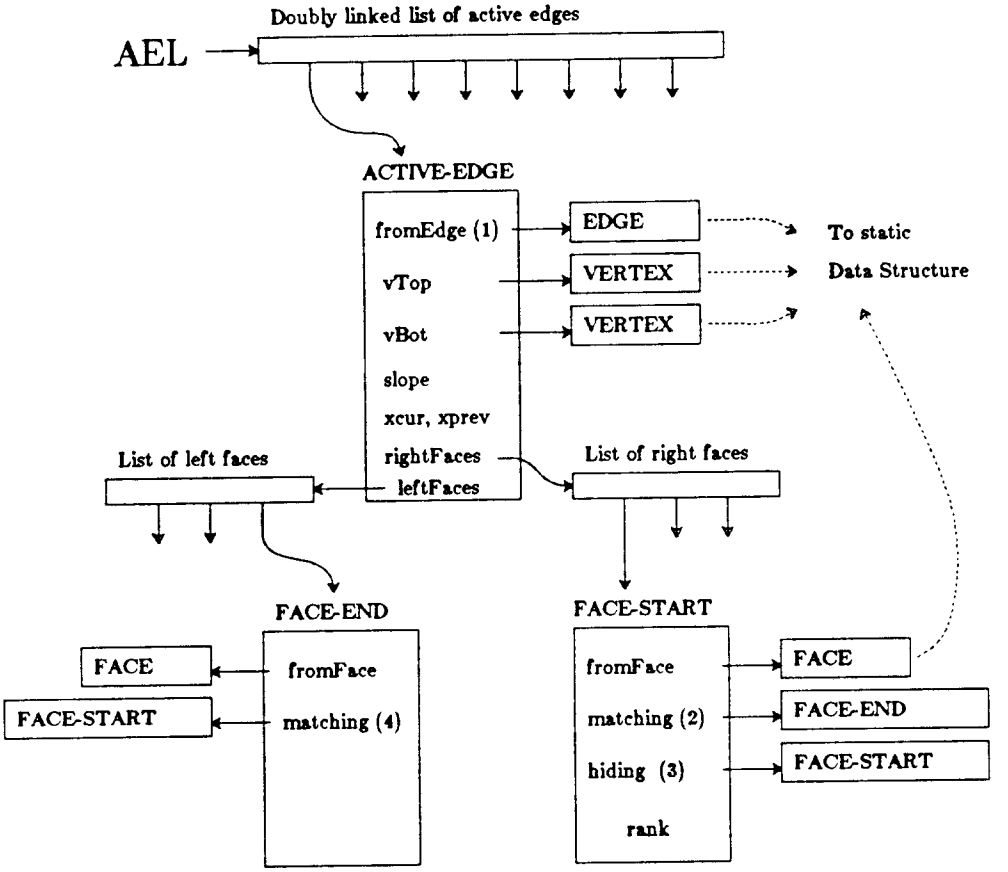


Figure 2.2. *Dynamic Data Structure.*

(1) Pointer back to the corresponding edge in the static data.

(2) A link to the edge that borders this face from the right. Specifically it is a pointer to the matching leftFace entry in that edge.

(3) Pointer to a face that hides this face from the viewer. Relying on depth coherence, this pointer may save depth calculations on following scanlines.

(4) Similar to (2), it is a link to the edge that borders this face from the left.

face. Similarly in the *rightFaces* list the first item is the topmost face, and the last is the bottom-most. See Figure 2.3. An easy way to compare the depths of two such faces is described in [18]. An active edge with two non-empty face lists is termed a *seam* edge. If only one of the lists is non-empty then the edge is termed a *contour* edge (not to be confused with the CONTOUR data structure). The reason behind the terminology is that in the final picture the *contour* edges

outline the silhouettes of the rendered bodies, while the *seam* edges help to grasp the shapes of the surfaces.

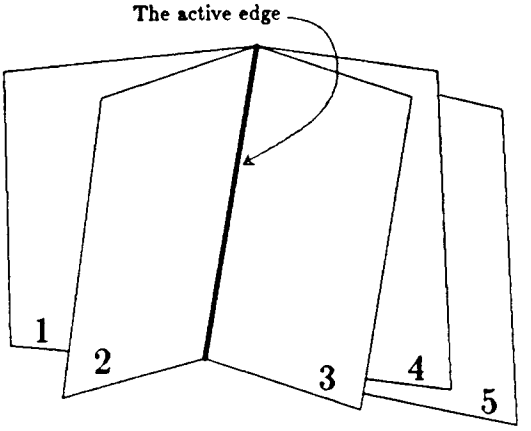


Figure 2.3. *"Book of faces" connected to an active edge.*
leftFaces list: [1] ⟶ [2]
rightFaces list: [3] ⟶ [4] ⟶ [5]

2.3.2. Active Faces

There are two structures for every face segment on the current scanline. The first one is a FACE-START item in the *rightFaces* list of the edge that borders the segment from the left, and the second is a FACE-END item in the *leftFaces* list of the edge that borders the segment from the right. Both structures point to the same static FACE entry with the *fromFace* pointer. They also point to each other with the *matching* pointer. *Matching* pointers are initially *NULL*. They are set and reset by the STACK module. FACE-START's and FACE-END's, as may be seen in figure 2.2, are equivalent data structures; they do however have different names to distinguish their use with regard to faces in the AEL. They are also called *active-faces* by analogy to edges and active-edges.

The next two fields are relevant only for FACE-START entries, and carry two forms of depth coherence:

- *hiding* is a pointer to a FACE-START whose face hides this segment from the viewer. Every time a depth comparison occurs between two faces, the result

is saved by setting this pointer in the hidden face.

- *rank* is a small integer. Ranks are set by the STACK module and used for depth comparisons. They imply a partial depth ordering on the subset of potentially visible face segments.

In the following sections I will sometimes use the term "face" instead of FACE-START or FACE-END; the exact meaning will always be clear from the context.

2.4. Preparing the Next Scanline

The AEL carries all the dynamic information from one scanline to the next. To update it for scanline y, the following is done for every active-edge in the list:

- If the edge is flagged as *ending*, it is removed from the AEL.
- If the Y coordinate of *vBot* is greater than y, it is an *ending* edge. The edge gets flagged as such, and it will be removed from the AEL on the next scanline.
- *xPrev* is set to *xCur*, and *xCur* incremented by *slope* (the increment in X per scanline).
- Temporary flags from last scanline are reset in the active-edge and its active-face lists.
- If there are still vertices in bucket y, and the first vertex in the bucket falls to the left of our edge, that vertex is removed from the bucket and incorporated into the AEL (see 2.4.1). This step is repeated if possible.

If there are still vertices in bucket y after all the edges have been processed, they are removed from the bucket and placed at the end of the AEL. At this point the AEL contains all (and only) edges which intersect y, but they are not necessarily ordered. Since they *were* ordered on the previous scanline, and since all changes were local, a variant of bubble-sort is used to reorder the list. In that sort, we traverse the list from left to right, and in turn propagate each edge to the correct position among the edges to its left. This guarantees that when we work on the edge at position j, all edges at positions 0 to $j-1$ are already ordered. Propagating an edge to its final position is achieved by a series of single-step *swaps*. Besides exchanging the positions of the two neighboring edges, the swap provides important topological information: faces that are connected to those edges and did not overlap on the previous scanline may now be overlapping, and vice versa. We use that information for erasing ranks that have become irrelevant (when non-

overlapping faces become overlapping. See 2.5), and in border enhancement mode for finding the edge intersections (See chapter 4).

If new vertices were added at this scanline, the STACK module is instructed to reset the *matching* information in the AEL. It is possible to try and adjust all *matching* pointers to the changes that were introduced by the new vertices, but the computation required is considerable (consider for example a saw-shaped face with more than one "tooth" ending on the current scanline). The *matching* information is constantly renewed as a side effect of running the STACK module.

2.4.1. Adding a New Vertex to the AEL

Each edge has two end-vertices. We will say that the edge is a *starting* edge of the first end-vertex to be incorporated into the AEL, and an *ending* edge of the second vertex. Usually this means that edge e_1 is an *ending* edge of V if V is the lower end of e_1, and that edge e_2 is a *starting* edge of V if V is the upper end of e_2. If both ends of an edge lie between the the same two scanlines, then the leftmost (or topmost, if the edge is vertical) of its end-vertices is added first to the AEL; therefore, by way of construction, these edges start at their leftmost vertex and end at their rightmost. Such edges are called *semi horizontal edges.* They do not participate in the STACK algorithm because, as was mentioned before (in 2.2), the STACK deals only with edges and faces that intersect the current scanline. They do however play a major role in border enhancement as we shall see in chapter 4.

Each vertex has two (possibly empty) sets of *ending* and *starting* edges. Consider vertex V waiting to be processed in bucket y. When the time comes for V to be removed from the bucket, all its ending edges are already represented in the AEL by their corresponding active-edges, and their *actedge* pointers are therefore non-nil (Thus the *actedge* pointer provides a quick way to differentiate starting from ending). A new active-edge is created for each starting edge, and:

- *vTop* and *vBot* are set to the edge vertices.
- *xPrev* is the X coordinate of *vTop*.
- *xCur* and *slope* are calculated from the edge vertices and the current scanline.
- The face-list† of the edge is sorted into the *rightFaces* and *leftFaces* lists (see 2.3.1 on the order of faces in each list), and an active-face is created for each face segment.

† Wires are ignored at the moment. They are displayed only in border enhancement mode.

- The new active-edge is then inserted into the AEL in the position of V (i.e., just before the edge that is closest to V from the right). If there is more than one starting edge, all are inserted at that position, correctly sorted by slope between themselves.

The next actions are dependent on the number of edges in each set. If there are exactly one ending edge and one starting, we know for sure that they belong to the same contour(s), and use this fact to transfer as much information as we can from the old to the new (see Figure 2.4):

- The *matching* pointers are copied from the ending active-faces to the respective starting active-faces, and in the "matched"‡ active-faces the *matching* pointers are shifted from the ending to the starting entry.
- Ranks are transferred from ending *rightFaces* to the respective starting *rightFaces*.

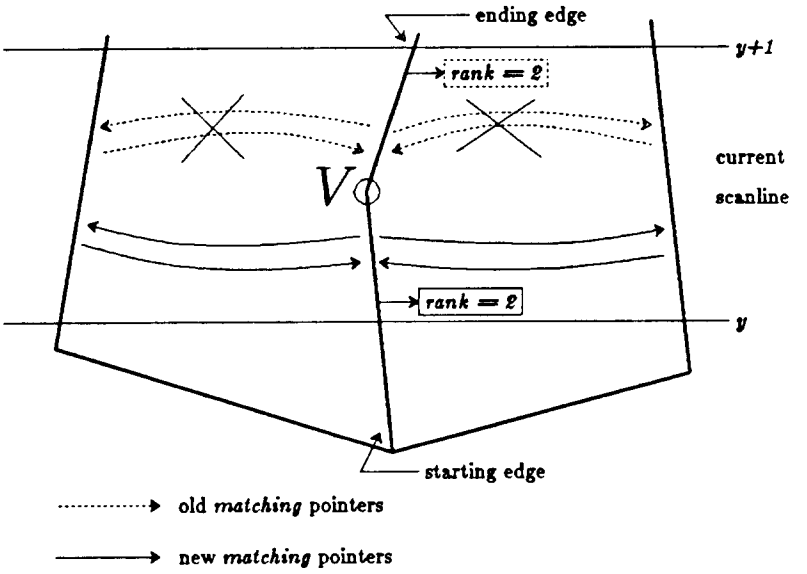


Figure 2.4. *New vertex V, with one ending edge and one starting edge.*

- *"matching" pointers adjusted.*
- *Ranks copied from ending to starting "rightFaces".*

‡ If the *matching* pointer in FACE-START A points to FACE-END B, we say that B is "matched" by A, and vice versa.

The general case of dealing with the edges attached to a vertex is quite complicated, and the STACK algorithm can handle it much more efficiently then any special processing here. Therefore in all other cases, ranks and *matching* information is simply erased in the faces that use our vertex and in the "matched" faces.

2.5. The STACK Algorithm

Once the AEL is updated and ordered, the STACK algorithm starts at the first (leftmost) edge in the AEL, with an empty stack, A *NULL current* face, and a rank counter which is set to 0. The first (topmost) face segment† in the *rightFaces* list of this edge is declared to be the *current* face, and processing begins on it. To process a *current* face means to scan the AEL from the left boundary of the face to its right boundary, and determine when (if ever) its visibility changes.

As our "scanning ray" moves to the next edge in the AEL, certain actions are done on the *leftFaces* list and then on the *rightFaces* list of that edge. Those actions depend on the *current* face and on temporary states of the edge and items in the above lists. When processing is finished on *all* items in the two lists, the edge is *marked* and it will not be considered any more on this scanline. The "scanning ray" is actually a pointer to the AEL. Whenever it advances to the next edge in the AEL it checks to see whether the edge is marked. If so, it immediately goes on to the next edges until it gets to an unmarked one.

If the next edge contains in its *leftFaces* list the FACE-END of the *current* face, and its *rightFaces* list is empty (a *contour* edge), then processing is terminated on this face. The rank counter is incremented by one and copied to the *rank* of the *current* face. Since the algorithm causes the frontmost face of any overlapping faces to have its processing terminated first, the ranks of two overlapping faces may be used on the next scanline to indicate their relative depths. (Ranks of two non-overlapping faces are meaningless, so if such faces become overlapping on the next scanline, we make sure that their ranks are erased to prevent wrong depth ordering based on those ranks. see 2.4). After ranking the

† To simplify the description from this point onward, I will refer to "face segments" simply as "faces", and ignore the fact that a single UNIGRAFIX face may be represented by multiple segments on the current scanline. It does not reduce the generality of the algorithm because every segment is uniquely identified and referenced by its FACE-START and FACE-END entries.

face, the stack is popped, the popped face becomes the *current* face, and processing is resumed on it. When this occurs, the ray jumps back to the position in the AEL it had when that face was pushed onto the stack. Thus, the algorithm does not always follow a strict left-to-right processing order.

If the next edge terminates the *current* face as above, but the *rightFaces* list of the edge is not empty (a *seam* edge), then the first FACE-START item in that list becomes *current* face without any depth comparisons, and without popping the stack. This transfer of the "*current*" title from face to face is called a "same level" transfer because the scanning ray doesn't have to jump up to a new hiding face, or to fall back to a face that was just popped from the stack. "Same level" transfers are repeated as long as the edge that terminates the *current* face has at least one *rightFace*, and the processing of the *current* face is not suspended by some hiding face. The collection of such contiguously connected faces creates a long pseudo "*current* face" which has all the properties of a regular *current* face except that it is piecewise. Whenever each face in that collection is terminated, its *rank* is set to the rank counter, but the counter is incremented only once, so all of them get the same rank.

If the first face in the *rightFaces* list of the next edge hides the *current* face, processing of the *current* face is suspended, and it is pushed on the stack together with the current position in the AEL. The hiding face becomes *current* and processing is started on it. The first step in processing a new *current* face is to find its right boundary (i.e., the corresponding FACE-END entry). In most cases this information is already stored in the *matching* field of the FACE-START entry. Otherwise we find it by a forward linear search across the AEL, and once found we store the relevant pointers in the *matching* fields of the new face and its FACE-END. If however our *current* face hides the faces in the *rightFaces* list of the next edge, nothing happens and processing continues.

If a face in the *leftFaces* list of the next edge is not the FACE-END of the *current* face, then it is the FACE-END of a face that is currently invisible. The algorithm marks that it has seen the right end of this face. If that face was *current* at some previous point, then it must have been pushed on the stack, and it still waits there to be popped out. When eventually it gets popped off the stack, the mark will cause the algorithm to immediately rank it and pop the next face from the stack. If however the face was invisible all the way, and never got to be *current*, then the mark has the effect of removing the face from further consideration on this scanline. A major advantage of the above mentioned big "piecewise *current*

face" is that all the faces that start and end within its span are practically ignored without spending any computation time on them.

Notice that two different *marks* were mentioned here. To distinguish between them we'll call them *e-mark* and *f-mark*; an active-edge is *e-marked* when all its left and right faces have been processed, and a FACE-START is *f-marked* when the scanning ray passes its matching FACE-END. The marks are kept in the corresponding data structures, although not shown in figure 2.2.

The following two sections restate and summarize in an orderly way what is done on each list of the next edge during the processing of the *current* face.

2.5.1. Processing the leftFaces

All the items in the *leftFaces* list are FACE-END's of some faces. One of them may be the FACE-END of the *current* face, in which case it *must* be the last item in the list, because of the way the list is sorted (see 2.3.1). To realize that, suppose f_1 is a FACE-END in the *leftFaces* list, and it ends the *current* face. f_2 is another item that comes after f_1 in the list; since f_2 succeeds f_1 in the list, the face it ends *hides* the *current* face, and therefore would have become *current* itself!

The algorithm traverses the list from head to tail. All the list items, except possibly the last one, are right ends of invisible faces which have to be f-marked. The actual f-marking is done on the left border of the face (i.e., the matching FACE-START entry of the face), and we find the pointer to it in the *matching* field of our item. If *matching* is not set then we find it by a backward linear search on the AEL, and once found record our findings in the relevant *matching* pointers.

If the last item in the list ends the *current* face, then we rank the face and terminate its processing. If the face got to be *current* in a "same level" transfer, then we want it to have the same rank as the previous *current* face, therefore the *rank* is simply set to the rank counter. Otherwise the rank counter is incremented by one before the setting. If the current edge is a *contour* edge, we pop the stack; if the popped face is f-marked, we increment the rank counter and rank the face. Then we pop more faces in a similar way until the popped face is unmarked. That face now becomes *current*, and the position in the AEL is reset to the position it had when this face was pushed onto the stack. If, on the other hand, the edge is a *seam* edge then a "same level" transfer will occur, so the stack is not popped.

When the algorithm is done with all the *leftFaces*, it sets a flag in the active-

edge to remember that. This is necessary because this active-edge may be visited again and again until all its *rightFaces* are processed, and we do not want to repeat the processing of the *leftFaces*.

2.5.2. Processing the rightFaces

The items in the *rightFaces* list are FACE-START's (left boundaries) of some faces. The first item in the list is the topmost and hides all the others.

When a *rightFace* is processed, we determine its depth relation to the *current* face. If the *current* face is *NULL* (e.g., at the beginning of the scanline, or in open spaces between objects) or is farther than the *rightFace*, then it is pushed on the stack with the current position in the AEL (i.e., the edge we're processing right now), and the *rightFace* becomes *current*. Otherwise nothing happens and processing continues on the *current* face.

The second way a *rightFace* could become *current* is via a "same level" transfer, i.e., the edge it belongs to is a *seam* edge and the *current* face had just terminated on the last item of its *leftFaces* list. In that case no depth comparison is needed and nothing is pushed on the stack.

When a *rightFace* becomes *current* in one of those two ways, we find its corresponding FACE-END. Finding that right boundary is not a necessary requirement of the STACK algorithm; rather it is a means of insuring scene integrity and optimizing other parts of the algorithm which require one boundary to know where the other boundary is (e.g., f-marking the *rightFace* of a segment when passing its right boundary). If no right boundary is found, then either the FACE-START entry is ignored, or some error recovery takes place. After doing that, the scanning ray moves to the next unmarked edge and continues processing of the new *current* face.

The *rightFacesPointer* of the edge initially points to the first item in the *rightFaces* list. It advances to the next item whenever the face it points to becomes *current*, or *f-marked*. When it gets to the end of the list, we know that all the faces that began on this edge have been accounted for, and that we can finally *e-mark* the edge. Consider for example an edge E with the *rightFaces* list: f_1, f_2, f_3. A typical scenario involving E can be the following:

- When the ray arrives at E for the first time, f_1 is compared with the *current* face, found to be closer, and becomes *current*. The *rightFacesPointer* advances to f_2.

- The ray advances to the next unmarked edge and processing continues. At some point the ray passes the matching FACE-END of f_2, so we *f-mark* f_2. Processing continues until at some later point the stack is popped, and the ray jumps back such that E is again the next unmarked edge.
- Since f_2 is *f-marked* the *rightFacesPointer* moves to f_3.
- f_3 is found to hide the *current* face, and becomes *current*. The *rightFacesPointer* is now at the end of the list, so E can be *e-marked*.

2.5.3. Producing Output

The algorithm outputs pixel spans in a monotonous left-to-right fashion, and puts a *marker* on the rightmost edge for which output has been produced. Each face is potentially visible whenever it is *current*, but if the face became *current* after the ray jumped to the left, then no output is produced until the rightmost marker is passed.

The display module collects sub-spans of the current visible face into a single unified span, and this span is sent to the output device when it ends. We know it ends when the current visible face changes.

More specifically, the display module keeps pointers to the currently visible face (*VisFace*), the edge where it started to be visible (*Start*), and the right limit i.e., the edge where it ceases to be visible (*Limit*). When more contiguous spans of the same face are sent from the STACK module, the right limit is updated. When a span from a new face arrives, the *VisFace* is sent to the output device spanning from *Start* to *Limit*. Then the *VisFace* is set to the new face, *Start* is set to the *Marker*, and *Limit* to *Next*.

Notice that when the *current* face is *NULL*, the *NULL* face is sent to the display module. This face will not be sent to the output device, but serves to flush out the current span.

2.5.4. Coherence and Depth Comparisons

Hidden surface algorithms use depth comparisons to select the visible face from a set of overlapping faces. This operation is carried out many times, so we try to optimize it and to minimize the number of floating point calculations. Once the depth relations among a set of faces have been established, we would like to use it to prevent re-calculation of known relations, and to infer more relations.

We have a set of assumptions about our environment and exploit any form of *coherence* that can be deduced from this set; see Sutherland *et al* [17] for an excellent discussion about types of coherence and their use. The more assumptions we make on the environment, the more types of coherence we can exploit, but correspondingly the set of legal scenes gets smaller. The UNIGRAFIX language supports concave faces with multiple contours and holes, and those faces may cyclically overlap each other. Therefore the only assumptions about faces are that they are planar (to a reasonable degree), do not have self intersecting contours, and do not intersect each other. There is no notion of a *solid body*, but by using named vertices the language allows the user to define faces with shared edges, and imply face continuity across those edges.

Given the above assumptions, UGDISP exploits *depth* coherence of non-intersecting face *segments* rather than of whole faces. *Scanline*, *edge* and *face* coherence are exploited in updating the AEL for the next scanline and by passing the *matching* information across scanlines. *Area* coherence guarantees that the same shading level can be used for an output span between its left and right limits.

Whenever two faces† are compared, the depth relation is remembered by storing the pointer to the hiding face in the *hiding* field of the hidden FACE-START. The implementation of the STACK algorithm on a system with limited virtual memory (such as UNIX), requires deallocation and reuse of space for data structures. Suppose that for some face-starts f_1 and f_2, we set $f_2 \rightarrow hiding = f_1$ on some scanline; when we try to reference $f_1 \rightarrow hiding$ a few scanlines later, it may happen that the space pointed to was deallocated and is now used by another face! So storing just the pointer is not enough. To avoid such dangling reference problems, a *validity* stamp is attached to that pointer. This stamp is the number of the scanline in which the comparison had taken place. The program guarantees a gap of two scanlines between deallocation and reallocation of the same space for two different face entries; therefore a *hiding* pointer with a stamp higher than the previous scanline is invalid. As a matter of fact, the *hiding* pointer is updated only if the validity stamp is *older* (i.e., higher) than the current scanline. If f_1 hides f_2, but the stamp in f_2 equals the current scanline then f_2 was hidden by some f_3 earlier on this scanline, and we want to keep the pointer to f_3 because it is most likely that on the next scanline f_2 will be compared with f_3 before it will be compared with f_1.

The following tests are applied when the algorithm has to answer the question

† As in section 2.5, I will refer to "face segments" simply as "faces".

"which face is closer, f_1 or f_2 ?".

- If $f_1 \rightarrow hiding$ is valid and points to f_2, then f_2 hides f_1.
- If $f_2 \rightarrow hiding$ is valid and points to f_1, then f_1 hides f_2.
- If $f_1 \rightarrow rank$ and $f_2 \rightarrow rank$ are both non-zero, then the face with the smaller rank hides the other face.
- If none of the above worked, we have to calculate the depths of the two faces at the point of interest and to compare those depths. Remember that depth comparisons are done when during the processing of the *current* face we encounter an edge with a non-empty *rightFaces* list. Let's denote the next edge by E, the *current* face by $Current$, and the *rightFace* of E by New. The coordinates of the point of interest are:

$$x = E \rightarrow xCur,$$
$$y = scanline.$$

The depth of $Current$ is found from the plane equation of the face. If the plane equation is:

$$a * X + b * Y + c * Z + d = 0,$$

then the Z coordinate of (x, y) on that plane, which we shall call $zCur$, is:

$$zCur = -\frac{a * x + b * y + d}{c}.$$

The depth of New could be found in the same way from the plane equation of the new face, but since we know that (x, y) lies on the edge E, we can use a more precise method:
Let v_1 and v_2 be the end vertices of E, and let

$$\Delta x = v_2 \rightarrow x - v_1 \rightarrow x,$$
$$\delta x = x - v_1 \rightarrow x,$$
$$z_1 = v_1 \rightarrow z,$$
$$z_2 = v_2 \rightarrow z.$$

Then we can find the parametric position t of (x, y) on E, and use it to find the Z coordinate of that point, which we shall call $zNew$:

$$t = \delta x / \Delta x,$$
$$zNew = t * (z_2 - z_1) + z_1.$$

If Δy is greater than Δx we shall use Y instead of X to gain more precision. Moreover we have to make sure that t is in the range $[0,1]$, and clip to that range otherwise.
Once *zCur* and *zNew* have been determined, we compare them, and the face with the smaller depth hides the other face.

- If *zCur* and *zNew* are too close to make a safe decision, we choose another point on E which hopefully will result in a clearer distinction between the faces. This point is the vertex of E which is farther from (x,y) than the other vertex. *zCur* is found, as before, from the plane equation, and *zNew* is readily given by the Z coordinate of the vertex. They are compared, and the face with the smaller depth hides the other face.
- If the new values of *zCur* and *zNew* are still too close then E lies in the plane of *Current*. The faces's slopes are then compared with the "open book" sorting method, as if both of them are *rightFaces* of E (see section 2.3.1), and the face in front is the face that is closer to the viewer.
- If the slopes are equal then the two faces are co-planar. Co-planar faces may occur in UNIGRAFIX scenes that were created by generators like UGSWEEP, or simply by attaching two objects together. The mutual orientation of the faces is determined, i.e., whether their normals point to the same or to opposite directions. Opposite orientation will occur if one face is a back-face and the other is a front-face (a *back-face* is a face with a normal that points away from the viewer, but was included in the rendered scene because of the *-ab* display option).
  - If the faces have the same orientation then I assume that the topology of the faces that are attached to our compared faces is such that *Current* belongs to the visible surface, and that *New* belongs to the hidden surface. Therefore *Current* stays *current*, and *New* is hidden.
  - If the faces are back-to-back then the back-face hides the front-face. This may seem counter-intuitive at first, but to realize why it is true consider the following example: Cube A is mounted on top of cube B such that the top face of B and the bottom face of A coincide, the top face of A is removed, and we view the scene from above, so we can see the insides of A. From this view point, the top face of B is a front-face while the bottom face of A is a back-face, but we expect to see the latter![†]

[†] It is possible to create an "adversary" UNIGRAFIX scene such that in a back-

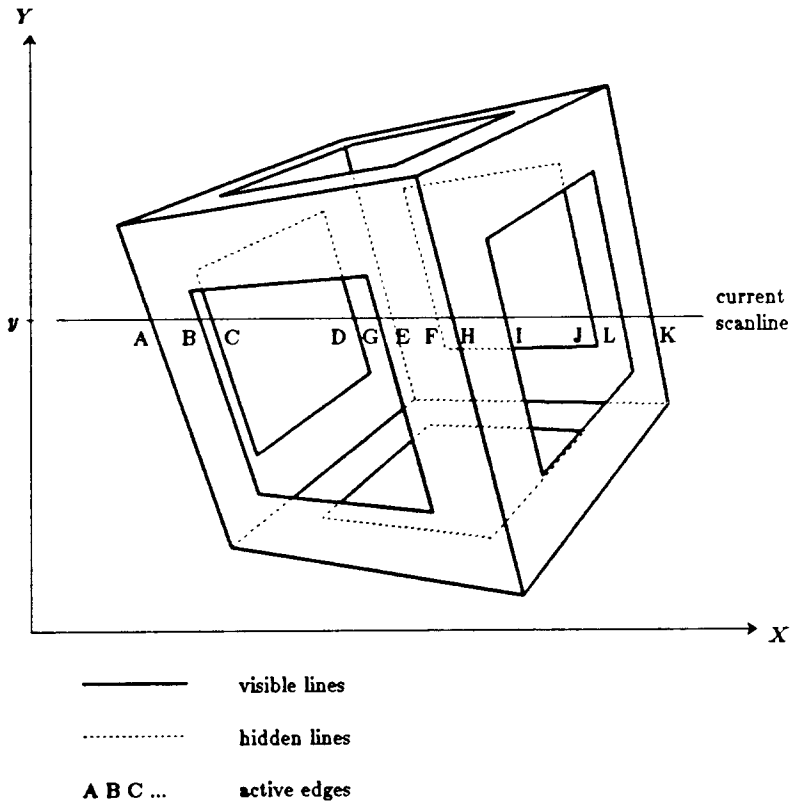


Figure 2.5. *A cube with rectangular holes, projected on the view plane. Edges A to K intersect the current scanline. (viewing parameters: -ep 3.2 1.5 -4 -vr 15)*

2.5.5. Examples

Figure 2.5 shows a perspective view of a cube with a rectangular hole in each infinitely thin face. Hidden lines are shown dashed. Let's follow the execution of the algorithm on the marked scanline during the display of this cube. Figure 2.6 shows the intersection of the horizontal sweep plane at the current scanline with our scene. Edges are denoted with capital letters (e.g., A), and face segments are denoted by their bounding edges (e.g., (A, B)). Notice the following facts:

- Each face is represented on this scanline by two segments; e.g., (A, B) and

to-back case the *front-face* is the one that should be visible. Nevertheless, such scenes are relatively rare, and the practical solution for a correct rendering then would be to slightly separate the co-planar faces from each other.

(G, H) belong to the same face.

- Edges E and H are *seam* edges, and all the others are *contour* edges.
- Edge A has a $rightFaces$ list of two items. Following the sort rules from section 2.3.1 we see that (A, B) should be the first, and (A, C) is the second. Similarly, K has a $leftFaces$ list of two items.

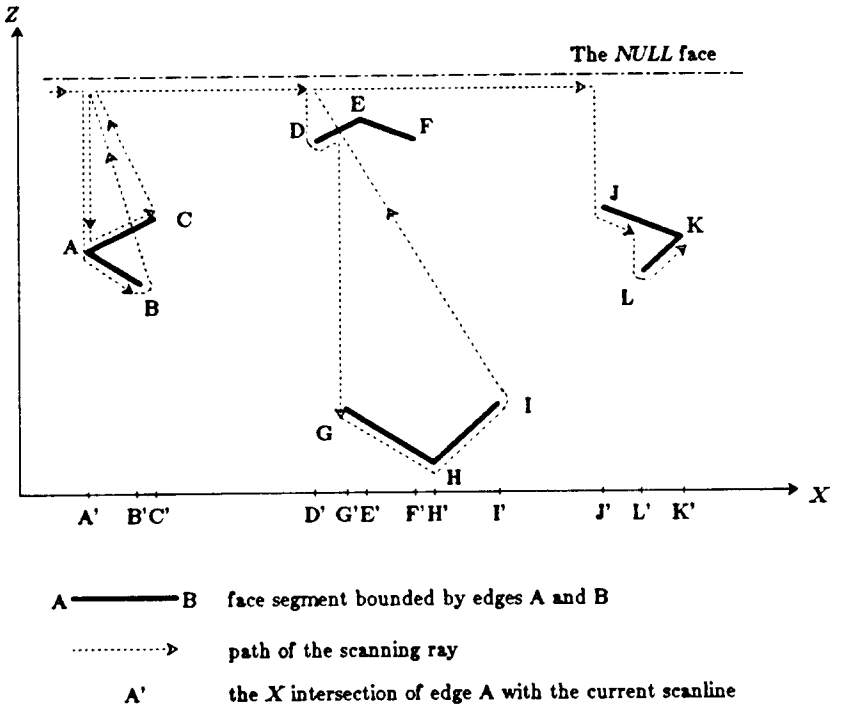


Figure 2.6. *Trace of the STACK algorithm on the current scanline.*

The sorted AEL is illustrated in 2.7. The *matching* pointers were set in previous scanlines. The stack is initially empty, the *current* face is $NULL$, and $RANK = 0$. When some face-start (X, Y) and edge Z are pushed onto (or popped off) the stack, We'll use the notation: $((X, Y), Z)$.

- The first unmarked edge is A, so face (A, B) becomes *current* face, and the $rightFacesPointer$ is advanced to (A, C). The $NULL$ face is pushed on the stack together with position A.
- At B, processing of (A, B) is terminated. $RANK$ is incremented to 1, and (A, B) is ranked 1, and f-marked. B is e-marked. The stack is popped and the *current* face is set to $NULL$. The ray jumps back to A.
- The $rightFacesPointer$ is on (A, C) now. The *current* face is $NULL$ so (A, C)

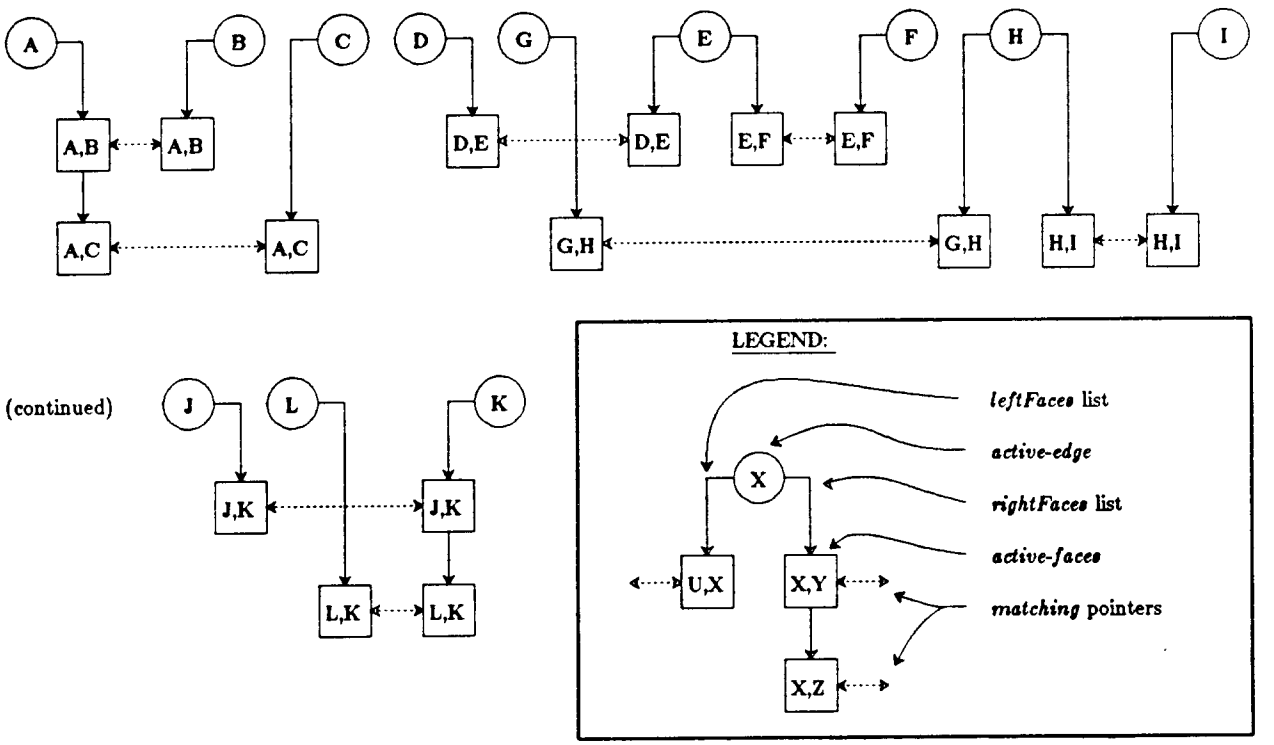


Figure 2.7. *The active edge list.*

becomes *current*. It is the last item in the *rightFaces* list, so A is e-marked. $(NULL, A)$ is pushed onto the stack.

- At C, processing of (A, C) is terminated. $RANK$ is incremented, and (A, C) is ranked 2, and f-marked. C is e-marked. $(NULL, A)$ is popped from the stack and the *current* face is set to $NULL$. The ray jumps back to A.
- The next unmarked edge is D. (D, E) becomes *current*, and D is e-marked.
- At G, we compare the relative depths of (D, E) and (G, H); the latter is closer so $((D, E), G)$ is pushed, and (G, H) becomes *current*. G is e-marked.
- At E, we see the right end of (D, E) so it is f-marked. (G, H) wins the depth comparison with (E, F) and stays *current*. E is not e-marked yet because (E, F) is still potentially visible.
- At H, the *current* face terminates. $RANK$ is incremented to 3, and (G, H) is ranked and f-marked. Face (H, I) becomes *current* in a "same level" transfer because H is a *seam* edge. H is e-marked.

- At F, we see the right end of (E,F), so it is f-marked. The *rightFacesPointer* of E is advanced to the end of list, so E is e-marked. F is also e-marked.
- At I, processing of (H,I) is terminated. It started as a "same level" edge, therefore it is ranked 3 (without incrementing $RANK$) and f-marked. I is e-marked. Face (D,E) is popped from the stack. Since it is already f-marked we immediately give it rank 4 ($RANK$ incremented), and pop again. The *current* face is set to $NULL$, and the ray jumps to D.
- Next unmarked edge is J. $(NULL,J)$ is pushed, and (J,K) becomes *current*. J is e-marked.
- At L, face (J,K) is compared with (L,K) and loses. So $((J,K),L)$ is pushed, and (L,K) is *current*. L is e-marked.
- At K, we see the right end of (J,K), so it is f-marked, and J is e-marked. The *current* face also terminates at K. It is f-marked and ranked 5. $((J,K),L)$ is popped from the stack, but it is already f-marked, so it is immediately ranked 6, and the stack pops again. The ray was supposed to jump to J to look for the next unmarked edge, but since K is the last edge in the AEL, we *know* that by now all edges are e-marked and all faces are f-marked, so processing of this scanline is terminated.

  Spans sent to the display module are:

  $(A'-B')$ from face (A,B).

  $(B'-C')$ from face (A,C).

  $(D'-G')$ from face (D,E).

  $(G'-E')$ and $(E'-H')$ from face (G,H).

  $(H'-F')$ and $(F'-I')$ from face (H,I).

  $(J'-L')$ from face (J,K).

  $(L'-K')$ from face (L,K).

  Figure 2.8 shows the actual output the algorithm produced from our scene.

Figure 2.9 illustrates more complicated cases which did not appear in the previous example. The reader is encouraged to trace the algorithm's execution on this scene, and verify the path of the scanning ray and the resulting ranks of the faces. To clarify a few fine points, note the following remarks:

- Processing of (A,B) is suspended and resumed three times. Whenever it is popped from the stack, processing is resumed at the position in the AEL it had when it was pushed.

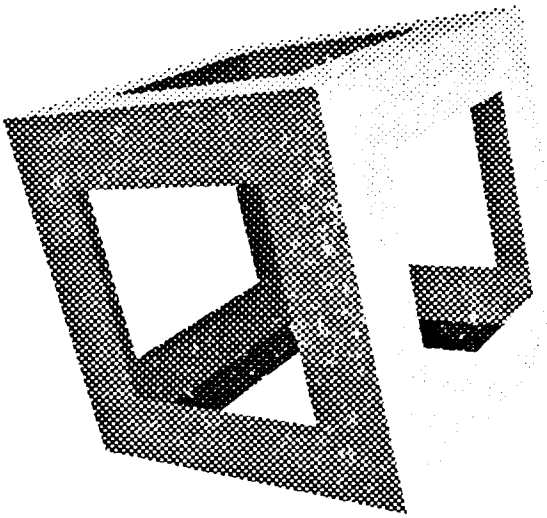


Figure 2.8. *The final result: hidden surfaces are removed, visible faces are shaded.*
(display options: -sf -ab)

- No processing is done on (G, H). Consequently it gets no rank.
- When the ray gets to N the state of the algorithm is:

Stack: **Current:**

(J, K)	M	f-marked	(M, N)
(E, F)	I	f-marked	
(A, B)	E		
$NULL$	A		

face	**position**	**status**

After processing of N the state becomes:

Stack: **Current:**

$NULL$	A	(M, N)

face	**position**	**status**

and the following ranks are assigned: $(M, N) \leftarrow 3$, $(J, K) \leftarrow 4$, and $(E, F) \leftarrow 5$.

- Although J and K are *seam* edges, faces (I, J), (J, K) and (K, L) all get different ranks! The reasons are:
  (J, K) starts in a "same level" transfer, but ends as hidden while (M, N) is the *current* face; therefore it gets ranked when it is popped from the stack.
  (K, L) becomes *current* during the processing of (A, B), and not by a "same level" transfer.

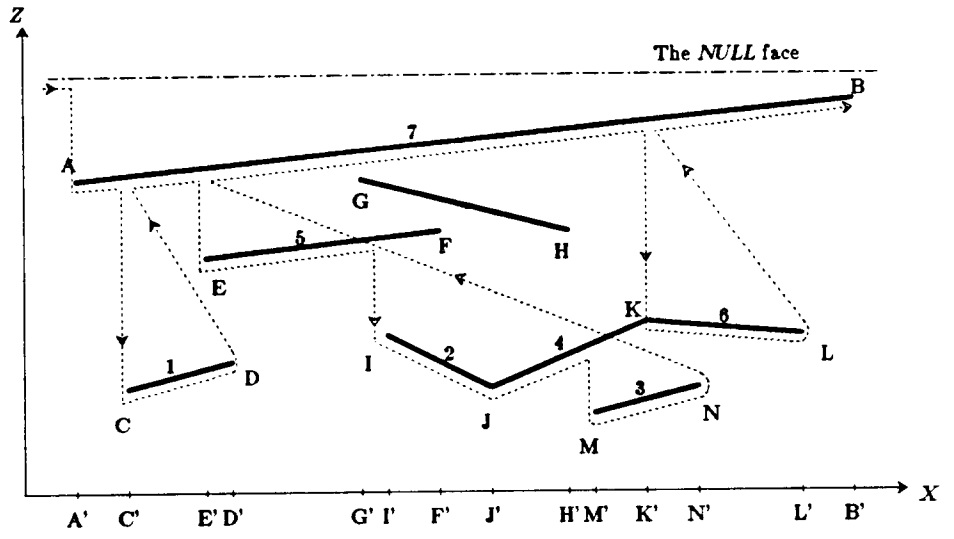


Figure 2.9. *Another example of the STACK algorithm.*
The numbers above the face segments are the resulting ranks.

2.5.6. Optimisations

If no new vertices were added to the scene at the current scanline and no two edges swapped their positions in the AEL, then there is no need to run the STACK algorithm; depth coherence of faces guarantees that the same faces which were visible on the previous scanline will also be visible on this scanline, and in the same order. Therefore if we keep a list of all the output spans, with pointers to their left and right bounding edges, this list could be used to produce the output spans for the next scanline, with the updated X intercepts of the edges in the AEL. This observation was first made by Romney (see the section on Scanline Algorithms in [17]); its implementation is simple and it saves a lot of computing time in constant parts of the scene.

Another type of optimization involves the capabilities of the output device. If the device is capable of accepting polygon descriptions, then the above list could be used to group several spans of a visible face into a single trapezoid. Communication to the device usually takes a significant amount of I/O and system time, so this greatly reduces total execution time.

3

Display of Intersecting Faces

This chapter describes the modifications and extensions to the basic algorithm that are required to detect intersecting faces and correctly display them.

By allowing faces to intersect, we violate one of our basic assumption, so we have to identify the parts of the algorithm that depend on the non-intersection assumption and remove that dependency. Then we have to add the modules to detect intersections and incorporate them into the general framework. Detection and construction of intersection lines are performed on the fly during the STACK algorithm, and they make effective use of its features to minimize the extra computation. Once the intersection edges are inserted, they act very much like regular edges, and require very little special processing.

3.1. Modifications to the STACK algorithm

When we review the ways in which the basic algorithm exploits coherence we realize that most of them stay valid in the presence of intersecting faces, provided of course that the intersection lines are detected and made into ordinary edges. Some of them are valid most of the time except in the close vicinity of the intersection lines. More specifically:

- The AEL still represents exactly those edges that intersect the current scanline, provided that intersection lines are added and removed at the correct points.
- The X-intercept of an edge on the next scanline can still be found from the current intercept incremented by $\Delta x/\Delta y$, because intersections do not affect edge linearity.
- An output span between two consecutive edges in the AEL is guaranteed to belong to a single face (again, after the new edges are present in the AEL).

- The *hiding* fields in active-faces (see 2.5.4) are valid across scanlines, provided that when an intersection is detected between faces f_1 and f_2, their *hiding* pointers are mutually adjusted. No other depth relation is affected by that intersection.
- *Ranks* of active-faces behave the same way as *hiding* pointers. Making sure that at least one of the *ranks* of the two intersecting faces is zero, guarantees that no order is implied between them. The partial order of the set of all ranked faces stays valid.

One use of coherence that we have to give up is the use of *matching* pointers. Since intersection edges may come into existence at any point during execution of the STACK module, we cannot just use the FACE-END from the previous scanline when we look for it on the current scanline; instead, we have to perform a forward linear search. In fact, as described in the next section, we get the FACE-END as a by-product of detecting intersections. Once the *matching* relation has been found for some face on the current scanline, we can use it to f-mark the FACE-START when we pass the FACE-END of that face, because our method guarantees that when the ray is at some position E in the AEL, all the intersection lines to the left of E have already been discovered and added to the AEL.

Since intersections are detected during execution of the STACK algorithm, we have to run it on every scanline. Thus we cannot reuse the image of the previous scanline on the current scanline, even if no new vertices were added and the order of edges was not changed in the AEL (see section 2.5.6).

3.2. Detecting Intersections

The main advantage of handling intersections as part of the display process, over pre-processors for intersection removal, is that the former has to find only the visible intersection lines, while the latter has to find and remove *all* the intersections in the scene, regardless of viewing direction. Those pre-processors usually test each polygon in the scene against all other polygons, and may have a quadratic time and space behavior. Bounding box tests may alleviate the task for well structured scenes [9]. UGDISP, on the other hand, detects intersections only for *current* faces, and each *current* face is tested only with those faces that overlap it on the current scanline. Also, intersection edges are added to the AEL only if they are visible (i.e., to the right of the output marker).

Recall from section 2.5 that when a FACE-START entry becomes the *current*

face, we first find its matching FACE-END. When handling intersection, another important task is performed before processing continues; a list is made of all the faces that may penetrate this *current* face during its processing. This list will be inspected every time the scanning ray moves to the next edge in the AEL, to test if some intersection line becomes visible. Clearly we want to minimize the size of this list and the effort to create it each time for the *current* face.

Theoretically every face on the current scanline is a potential penetrator, except for all those faces that are parallel to the *current* face. Actually, even if the planes of two faces intersect, the two individual segments cannot intersect if they do not overlap. Therefore the horizontal span of a *current* face f_1 on the current scanline is used to limit the set of faces that will be tested: faces that have already ended (f-marked) by the time f_1 becomes *current*and faces that start to the right of f_1's FACE-END are not tested. Culling out the faces from the left is achieved by keeping a pointer to the *lim-inf* of unmarked edges, i.e., the rightmost edge in the AEL for which all the preceding edges are already e-marked. We start at that lim-inf, move along the AEL until we find the FACE-END of f_1, and on the way add each unmarked FACE-START to the *potential penetrators list* (PPL) of f_1, as described in section 3.2.1. Clearly the edge that contains f_1's FACE-END in its *leftFaces* list, is the *lim-sup* of our set, i.e., the leftmost edge in the AEL for which all the succeeding edges do not overlap f_1 on the current scanline.

When the PPL is ready, we have the matching FACE-END of the new *current* face as well, and processing of f_1 begins. Section 3.2.2 describes how the list is used to detect faces that realize their potential.

3.2.1. Adding a Potential Penetrator to the PPL of f_1

Each item in the PPL is a FACE-ISECT structure, containing the following information:

- *ppFace* is a pointer to the potentially penetrating FACE-START.
- (u, v) encode the intersection line between the planes of the *current* face and *ppFace*; u is the direction vector $\mathbf{u} = (a, b, c)$ of the line, and v is some point $\mathbf{v} = (x, y, z)$ that lies on it.
- *xIsect* is the X-intercept of the intersection line with the current scanline.

The encoding of the intersection line is derived from the plane equations of the two faces and therefore stays invariant through the whole scene. Since any two faces may overlap across many scanlines, we store that encoding, once

calculated, for easy retrieval the next time we will need it. The obvious place is the FACE static data structure; so for each FACE entry an additional linked list holds descriptions of intersection lines with other faces. Figure 3.1(a) shows part of the static data structure where intersection data has already been calculated for faces (α, β) and for faces (α, γ). Notice that this time we are talking about whole *faces*, and not just segments, because all the segments of a face have the same plane equation.

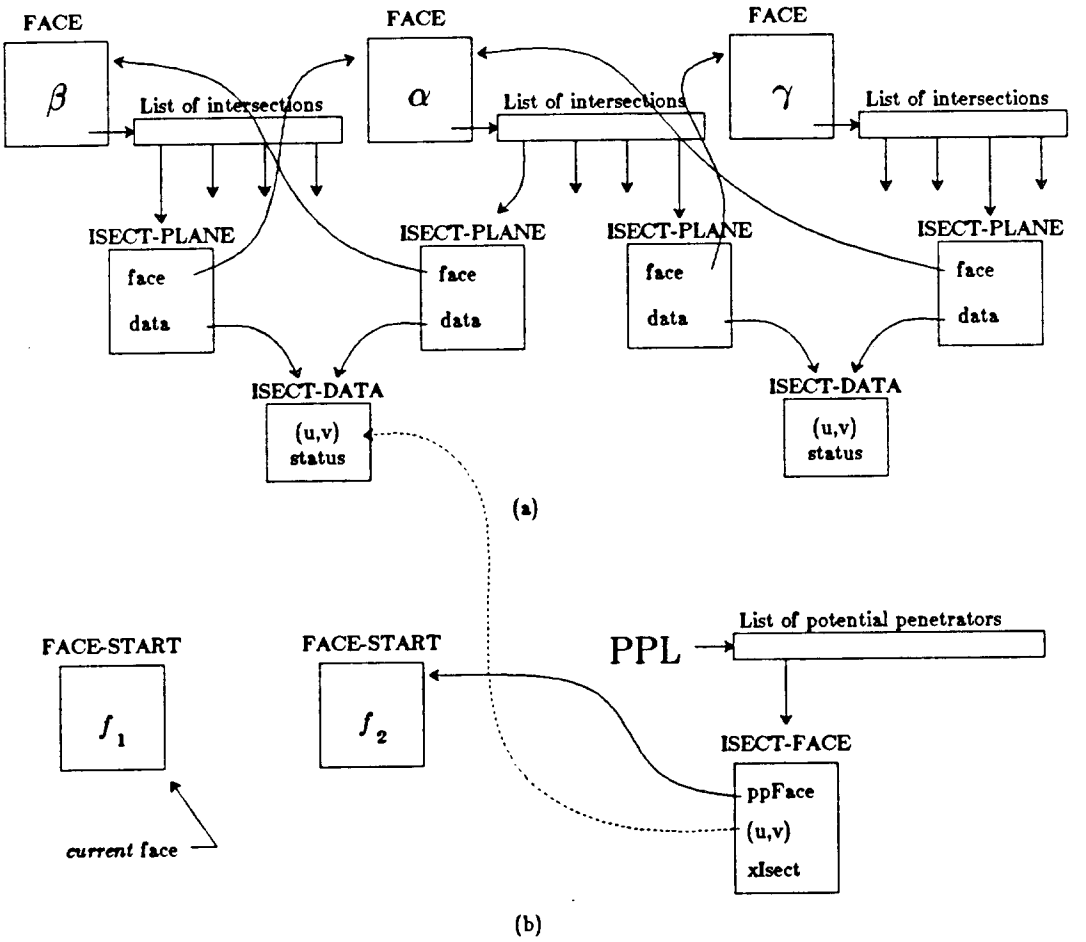


Figure 3.1.

(a) Keeping intersection line data in the static data structure.

(b) The PPL in the dynamic data structure.

f_1 is the current face, f_1's face is α, and f_2's face is β. The status flag in ISECT-DATA is set if this intersection is currently represented as an edge in the AEL.

Suppose now that we have to add FACE-START f_2 to the PPL of the *current* face f_1. First, we search the intersection list of f_1's face for an entry with f_2's face; if found, and if the *status* flag is not set, we take the intersection data from there. If the flag is set, then an intersection edge already exists for f_1 and f_2, so f_2 is not added to the PPL. If such an entry is not found, we have to calculate it. The following calculation of the intersection data is taken from [9].

From the FACE structures we obtain the normalized plane equations:

$$a_1 x + b_1 y + c_1 z = -d_1$$
$$a_2 x + b_2 y + c_2 z = -d_2$$

The vector (a_i, b_i, c_i) is the normal to the appropriate face.

The direction vector $\mathbf{u}$ of the intersection line is found by computing the cross-product of the two plane normal vectors and normalizing the result. To obtain a point $\mathbf{v}$ on the line, we introduce a third arbitrary plane equation, constructed to have normal $\mathbf{u}$ and to pass through some vertex in one of the faces, yielding:

$$\begin{pmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} -d_1 \\ -d_2 \\ -d_3 \end{pmatrix},$$

or,

$$\boldsymbol{A}\mathbf{v} = \mathbf{d}$$

The system is nearly orthogonal, since each of the rows $\mathbf{r}_i$ of $\boldsymbol{A}$ is a unit vector, and $\mathbf{r}_1 \cdot \mathbf{r}_3$ and $\mathbf{r}_2 \cdot \mathbf{r}_3$ are both zero by construction. We need only alter $\mathbf{r}_1$ so that it is orthogonal to $\mathbf{r}_2$. We apply the elementary row operation:

$$\mathbf{R}_1 \leftarrow \mathbf{R}_1 - (\mathbf{r}_1 \cdot \mathbf{r}_2)\mathbf{R}_2$$

and renormalize:

$$\mathbf{R}_1 \leftarrow \frac{1}{|\mathbf{r}_1'|}\mathbf{R}_1$$

with $\mathbf{R}_i = (a_i, b_i, c_i, d_i)$. $\mathbf{r}_1'$ is a three-vector whose components are the first three components of $\mathbf{R}_1$ after the first row operation. The resulting system is orthogonal, and we obtain our point:

$$\mathbf{v} = \boldsymbol{A}'^{T}\mathbf{d}',$$

where the primed quantities represent the result of the orthogonalizing row operations. The intersection line encoding, $(\mathbf{u}, \mathbf{v})$, is stored in an ISECT-DATA

entry, and the proper ISECT-PLANE entries are added to the intersection lists of f_1's and f_2's faces.

The X-intercept of the intersection line with the current scanline y is computed from $\mathbf{u}$ and $\mathbf{v}$. The complete item is now inserted to the PPL of f_1, which is kept sorted by the X-intercepts. Figure 3.1(b) shows f_1's PPL after the insertion.

3.2.2. Finding if f_1 is Penetrated

In processing of the *current* face f_1, the scanning ray moves from the edge it had last processed to the next unmarked edge. When intersections are not allowed, we are guaranteed that along the span between those two edges f_1 stays *current*, and hides all the yet unmarked faces. However, when handling intersections, another face may penetrate f_1 within that span and should be made the *current* face instead. Notice though, that we care about penetrating faces only if f_1 is visible (i.e., the next unmarked edge succeeds the output marker). As long as f_1 is still invisible, the intersection line would also be invisible, so we ignore it.

Before processing the next unmarked edge (which we shall call *Next*), we examine the items in the PPL from head to tail in order to determine whether an intersection line falls within the relevant interval on the current scanline. The following is done on each item:

- If *ppFace* is f-marked, then this face has already terminated, so the item is removed from the PPL.
- If *xIsect* is smaller than the output marker, or if it is larger than the right end of f_1, then the intersection line lies outside the relevant part of f_1, so the item is removed.
- If *xIsect* is larger than *Next*→*xcur*, then no penetration will occur before *Next*. Since the list is sorted by *xIsect*, the next items will also have an *xIsect* field larger than *Next*→*xcur*, so we stop examining the list. No item is removed from the list.
- We now know that *ppFace* is still active, and that the intersection line lies within the relevant part of f_1 and comes before *Next*. If the line lies outside *ppFace*, nothing happens, but if it lies within the span of *ppFace*, then a penetration occurs! We stop examining the list, and an intersection edge will be created and inserted to the AEL. In both cases the item is removed from the list.

If a penetration was not found, f_1 stays the *current* face, and *Next* is the next edge to be processed.

3.3. Incorporating the Intersection into the Data Structure

Suppose that while f_1 is *current* and *Next* is the next unmarked edge, penetration of f_2 was detected. The first step of incorporating the intersection line into the AEL, is to set the *status* flag in the corresponding ISECT-DATA entry. When f_1 later becomes *current* on the next scanline and a PPL is built for it, this flag will have the effect of excluding f_2 from the PPL. After setting the flag, we create a new ACTIVE-EDGE structure and fill it with data:

- A new VERTEX entry is created for the top vertex. Its Y coordinate is y (the current scanline), the X coordinate is *xIsect* from the PPL item, and Z is calculated from the plane equation of one of the participating faces. *vTop* is set to this vertex.
- The bottom vertex stays unspecified because we don't know yet when and where this intersection edge is going to terminate; so *vBot* is *NULL*.
- *slope* is calculated from the direction vector of the intersection line.
- *xPrev* and *xCur* are set to *xIsect*.
- *leftFaces* and *rightFaces* lists are constructed to represent a "cut" of one face segment into two segments.

Cutting just one face insures correct behavior of the STACK algorithm (Once again, this is simpler than intersection removal pre-processors which usually cut both faces). The face that gets cut is arbitrarily chosen to be the face that is hidden on the left side of the intersection line, and hiding on the right side. It is determined by the same technique that is used to do the "open book" sort on faces for ordinary edges (See 2.3.1). Figure 3.2(a) illustrates a situation where the penetrating face is cut, while in 3.2(b) the *current* face gets cut.

Let f_2 be the cut face in our example (as in figure 3.2(a)). The *rightFaces* list of the new active-edge consists of a single FACE-START entry, which we shall call *fIsect*. Its fields are set as follows:

- *fromFace* is set to f_2's face.
- If $f_2 \rightarrow$ *matching* is set, then *fIsect* $\rightarrow$ *matching* is set to it, and the *matching* pointer in the matched FACE-END is set to *fIsect*. Otherwise *fIsect*$\rightarrow$*matching* is set to NULL.

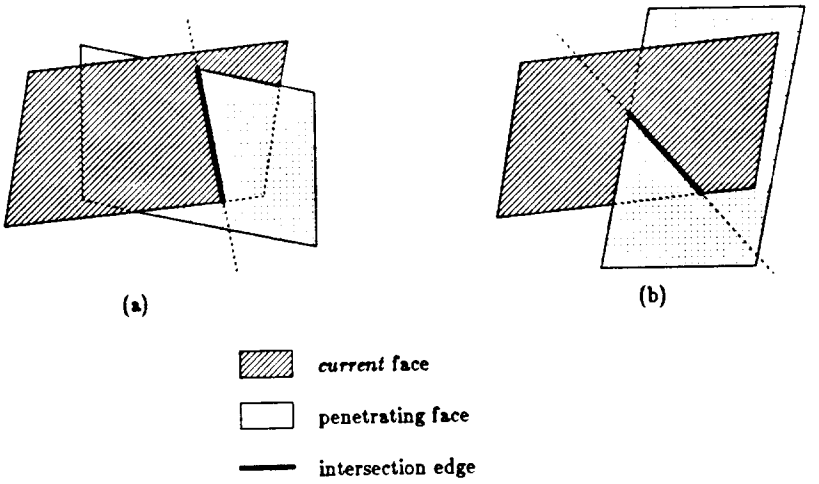


Figure 3.2. *Which face will be cut into two segments?*
(a) The penetrating face.
(b) The current face.

- *rank* is set to 0.
- If $f_2 \rightarrow hiding$ pointer is "fresh" (validity stamp equals to current or to previous scanline) and does not point to f_1, then $fIsect \rightarrow hiding$ is set to it (remember that *fIsect* hides f_1!). Otherwise it is set to *NULL*.

The *leftFaces* list of the new active-edge consists of a single FACE-END entry; its *fromFace* field is set to f_2's face, the *matching* pointer is set to f_2, and $f_2 \rightarrow matching$ is set to point back to this new FACE-END.

The *hiding* pointers of the two original faces are modified as well (unless they have fresh settings from the current scanline): $f_1 \rightarrow hiding$ is set to *fIsect*, and $f_2 \rightarrow hiding$ is set to f_1.

Finally, the AEL is searched from *Next* backwards for the correct X position, and the new edge is inserted there. The search is necessary because there may be some already marked edges between *Next* and the correct position.

3.4. Special Processing of an Intersection Edge

Once the intersection line is incorporated into the AEL, it requires no special processing in the STACK module. However, we still have to find where it terminates. The line ceases to be a real edge when it crosses another edge that

belongs to any of the two faces that had originated the intersection. Figure 3.3 illustrates a typical intersection edge. The intersection line between faces α and β starts invisible under face γ, and is made into a real edge only when it first becomes visible at point T. The edge terminates at point B; B is on the first scanline where the intersection edge crosses another edge that belongs to α or to β.

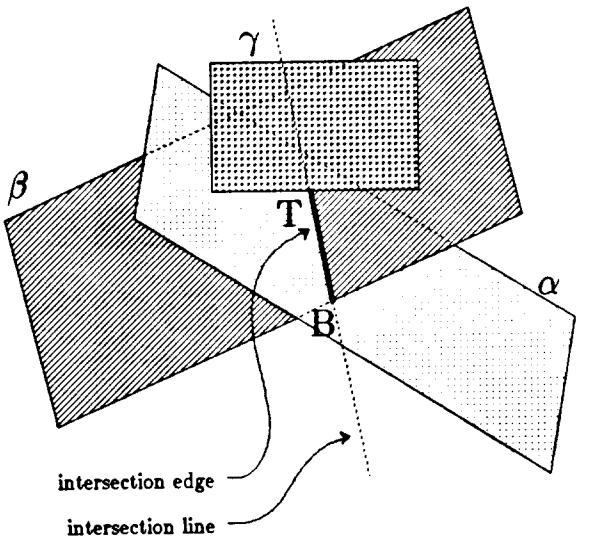


Figure 3.3. *Top and Bottom endpoints of an intersection edge. Hidden lines and the intersection line are dashed.*

Recall that when the AEL is updated for the next scanline, we bubble-sort the list with the new *xCur* values of the edges as the sort key (section 2.4). Whenever two edges are swapped we check the following conditions:

- If one edge is an intersection edge, and
- The second edge is *not* an intersection edge, and
- The second edge is used by one of the two faces that intersect on the first edge, and
- The two edges actually cross each other (they may have to be swapped to achieve correct *xCur* ordering, but if one or both of them had just ended or had just started on this scanline, they do not necessarily cross).

If all these conditions are true, the intersection edge terminates, and the following is done:

- The edge is flagged as *ending*.
- The *status* flag in the relevant ISECT-DATA entry is reset.
- If the *matching* pointer of the *rightFaces* is set, then the *matching* pointer in the matched FACE-END is reset.
- The FACE-START of the cut face (f_2 in our example) is found either by using the *matching* pointer in the *leftFaces* of the intersection edge, or by a search on the AEL if that pointer is reset. On this FACE-START we do:
  - The *rank* is zeroed.
  - The *matching* pointer is reset.
  - If the *hiding* pointer points to a FACE-START whose face is the second face of the intersection pair (α in our example), then this pointer is reset.

Notice that we do not bother to find the exact point where the intersection edge terminates. This will be important only in border enhancement mode.

4

Border Enhancement

Enhancing polygon borders with solid edges gives the picture a "crisp" look and makes complicated pictures much easier to understand. It is especially important when the output device provides a limited number of colors or just black and white stipple patterns. Basically the same work is required for producing just the line-drawing image of the scene. In that case we draw only the crisp edges and omit the shading of faces.

This chapter describes the algorithms to draw borders in scenes with and without intersecting objects.

In border enhancement mode we display the visible portions of edges with maximal contrast to the background†. Edges are used by wires and/or faces, but for the sake of simplicity I shall discuss face borders only; enhancing wires is a natural extension. The material in this chapter calls for a meticulous definition of what was freely referred to as "scanline" in the previous chapters: *Scanline y* is the horizontal line $Y = y$, and *swath y* is the horizontal strip between the previous line ($Y = y + 1$) and the current line ($Y = y$). Swath y includes scanline y, but excludes scanline $y + 1$.

The edges in the AEL that intersect the current scanline define a sequence of *spans* on it. The STACK algorithm determines for that scanline which face segment is visible along each of these spans. Using this fact, we can define an edge to be *visible* if it bounds a visible face, i.e., if the visible face on its left is in its *leftFaces* list and/or if the visible face on its right is in its *rightFaces* list. Soon we shall see that this definition is not complete, because it does not cover edges, or portions thereof, that are present on the current swath, but that do not intersect the current scanline (and therefore do not participate in the STACK algorithm). They may still be visible and must thus be enhanced as well.

---

† I assume a white background for our display, so edges are displayed in solid black.

Enhancing an edge E is an easy task on a swath where E does not start, does not end, and does not cross any other edges, because its visibility cannot change on such a swath. If E is invisible, according to the above definition, then nothing is displayed, otherwise we blacken the pixels over the horizontal extent of the edge in this swath, that is the range $[E{\rightarrow}xPrev, E{\rightarrow}xCur]$ (or $[E{\rightarrow}xCur, E{\rightarrow}xPrev]$ if the edge has a negative slope). Note that the range limits are floating-point numbers, while pixels are indexed with the integer coordinates of their bottom left corners. Therefore the above range is interpreted as all the pixels from $\lfloor E{\rightarrow}xPrev \rfloor$ (the integer part of $E{\rightarrow}xPrev$) up to and including $\lfloor E{\rightarrow}xCur \rfloor$. The resulting effect is that all the pixels the edge passes through on this swath are enhanced. See figure 4.1.

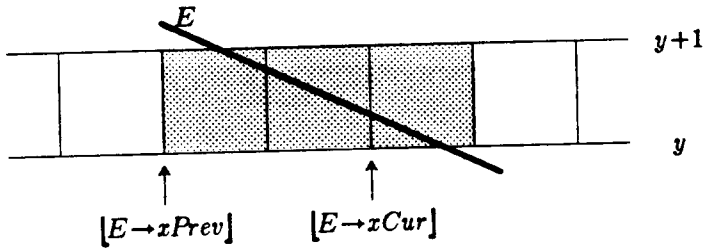


Figure 4.1. *Each square represents a pixel on the current swath. The pixels that the edge passes through are enhanced.*

In all other cases, E may experience visibility changes, and we have to find the visible portions and their exact endpoints. Figure 4.2 exemplifies four cases with increasing complexity. For cases (b), (c) and (d) there are also the corresponding (b)',(c)' and (d)' showing the final picture with the correct border enhancement.

- Figure 4.2(a) shows the simple case mentioned above. If the STACK algorithm finds that either the face to the left of the edge or the face on its right side is visible, then a span of pixels is blackened as described above.
- In figure 4.2(b) we have a vertex a with one ending edge and three starting edges. For each of these edges we already know the exact endpoints of its portion on this swath:

  The top point of E is $(E{\rightarrow}xPrev, y+1)$; the bottom point is a.

  The top point of F is a; the bottom point is $(F{\rightarrow}xCur, y)$.

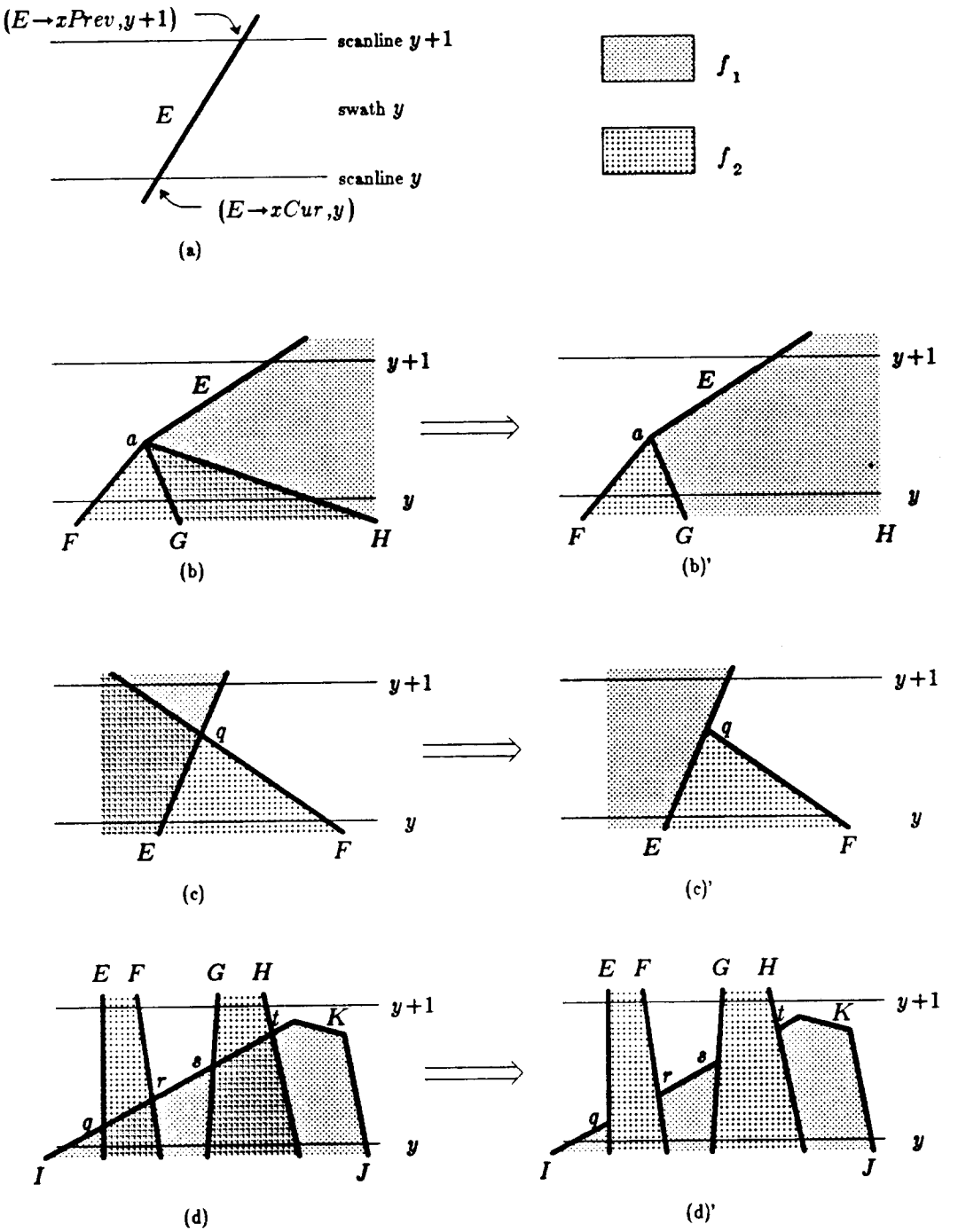


Figure 4.2. *Problems encountered in displaying borders.*

(a) Simple case, no problems.

(b) Ending and starting edges meet at a vertex.

(c) Two edges cross.

(d) Edges start and end on the same swath (SHE).

(b)', (c)' and (d)' show the corresponding scenes after the hidden feature removal.

G and H are analogous to F.

Suppose that the STACK algorithm has determined that the visible face between G and H is f_1. From this we find that F and G are visible, and that H is invisible. As for E, it was visible on the previous swath, and no other edge had crossed it on this swath, so it is visible here as well. As a result, the pixels in the range $[F{\rightarrow}xCur, E{\rightarrow}xPrev]$ are blackened. These are the X extrema in this swath for all visible edge segments attached to vertex a.

- In figure 4.2(c) two edges cross at an unknown point q. We first determine the coordinates of q, then decide which of the four edge portions are visible, and finally blacken the pixels between the extremes of the range of visible edge segments. It may seem at first that we need only two more fields in the active-edge structure, namely $xBorderPrev$ and $xBorderCur$. Then once we know that f_1 hides f_2, we can set them as follows:

$$E{\rightarrow}xBorderPrev = E{\rightarrow}xPrev,$$
$$E{\rightarrow}xBorderCur = E{\rightarrow}xCur,$$
$$F{\rightarrow}xBorderPrev = q{\rightarrow}x,$$
$$F{\rightarrow}xBorderCur = F{\rightarrow}xCur.$$

  and use them for border enhancement (see [10]). But the example in figure 4.2(d), in which one edge segment in the current swath is broken into more than one visible segments, demonstrates that two such fields do not suffice. Instead both edges must be cut at the crossing point q, resulting in a situation not unlike the one in figure 4.2(b).

- In figure 4.2(d) we have four such crossing points: q, r, s and t, and for each one we have to cut the two crossing edges. But another difficulty is introduced here: some of the edges (both original edges, and edges created by a crossing) start and end on the same swath. Recall from section 2.4.1 that such edges are termed *semi horizontal edges* (SHE's†). They don't carry visibility information from the previous swath, nor do they participate in the STACK algorithm, so their visibility must be found by some special purpose processing.

Let's review here once more the main modules of UGDISP and their interaction with each other during the processing of the current scanline. The first step is a

† No sexism implied...

pre-processing of the AEL for the STACK and display modules. It consists of two passes on the AEL: In the first pass edges that ended on the previous swath are removed from the list, the *xPrev* and *xCur* fields of the other edges are updated, and new edges are inserted to the list. In the second pass all the edges in the AEL are bubble-sorted by increasing values of their X-intercepts with the current scanline. Every pair of adjacent edges that are in the wrong order is swapped, and if those edges *cross* each other, the crossing point is found and each edge is cut into its two sub-parts. The cross-point determination and cutting are done only in border enhancement mode.

The second step is the STACK algorithm. The STACK and the display modules concurrently perform one pass on the AEL during which the STACK module determines the visible face segments, and the display module follows and displays them. Whenever the STACK module advances to the next unmarked edge in the AEL, and that next edge is farther on the list than the output right marker, a message is sent to the display module containing the *current* visible face and the boundaries of the last span. The display module collects all the contiguous spans from that visible face into one super-span and sends it to the output device. Each such super-span is bounded by *edge-clusters* from both sides. An edge-cluster can simply be the bounding edge, as in example (a) of figure 4.2; it can also be the bounding edge together with ending edges that are connected to its top vertex, as in example (b), or with ending edges that are connected to it by crossing points, as in examples (c) and (d). The display module determines the horizontal extent of the bounding edge-clusters and sends them to the output device as well.

It may be argued that in the first step there is no need to cut *both* crossing edges into four sub-edges, and that cutting just the farther edge is enough. Referring again to figure 4.2(d) it would mean that edges E, F, G and H would stay intact, and that edge I would be cut into five sub-edges. Now look at the edge-cluster that bounds span $(E-F)$ from the right; edges (r,s) and G should have been part of this cluster, but there would be no tie in our data structure between F and (r,s), and edge visibility would not be able to "propagate" to (r,s) and through it to G! It is true that G would be enhanced as the left boundary of the visible span $(G-H)$, but even then (r,s) would be isolated and unnoticed. Therefore both edges are cut and tied together for correct and efficient border enhancement.

Section 4.1 describes the processing of a crossing point, and section 4.2 describes how borders are found and discusses some of the techniques to find

visibilities of SHE's.

4.1. Crossing Edges

When we bubble-sort the AEL, we swap every pair of adjacent edges that are in the wrong order. We check if they cross each other, and if they do, we find the crossing point and cut each edge into its two sub-parts. The only exception to this procedure is when both edges are *known* to be invisible, that is both of them were present on the previous scanline and were determined to be invisible by the STACK module. In that case they are just swapped in the AEL without cutting them.

4.1.1. Finding the Crossing Point

Let the two edges be represented by their end vertices: $e_1 = (v_1, v_2)$ and $e_2 = (v_3, v_4)$. The distance of a point p from an edge e in the two-dimensional view plane is given by

$$d_{p,e} = n^T \cdot (p - v)$$

where n is the normal unit vector to the line, and v is a point on the line. For any point p on the line, $d_{p,e}$ is zero. Let n_2 be the normal to e_2. The distances of e_1's endpoints from e_2 are given by

$$d_{v_1,e_2} = n_2^T \cdot (v_1 - v_3),$$
$$d_{v_2,e_2} = n_2^T \cdot (v_2 - v_3).$$

If the endpoints of one edge lie on opposite sides of the other (distances have opposite signs), and vice versa, then the edges cross each other. The crossing point c_{e_1,e_2} is found by averaging the endpoints of e_1 with their distances from e_2

$$c_{e_1,e_2} = \frac{d_{v_1,e_2} v_2 - d_{v_2,e_2} v_1}{d_{v_1,e_2} - d_{v_2,e_2}}.$$

Note that the last equation frees us from the need to find a normal n_2 of unit length, because the non-normalized distances appear both in the denominator and the numerator. Therefore n_2 can be found simply by

$$n_2 = (v_{3\rightarrow y} - v_{4\rightarrow y} \quad , \quad v_{4\rightarrow x} - v_{3\rightarrow x}).$$

Also, we don't care which one of d_{v_1,e_2} and d_{v_2,e_2} is positive and which is negative, for exactly the same reason.

To achieve maximal accuracy we compute $d_{v_3,e_1} - d_{v_4,e_1}$, and if its absolute magnitude is larger than that of $d_{v_1,e_2} - d_{v_2,e_2}$, we exchange the roles of e_1 and e_2 in the computation. This method is more stable and accurate than directly solving the linear system of the parametric line equations, especially when an endpoint of one edge is very close to the line of the other edge.

4.1.2. Cutting Edges

The calculation in the previous section only finds the projection of the crossing point, i.e., only its X and Y coordinates are known. To find the corresponding two points on each edge, we calculate the Z coordinate from the edge endpoints in a similar way to that described in the previous section. Having found z_1 for e_1, and z_2 for e_2 we compare them and find which edge is on top (call it *topDog*), and which is below (call it *underDog*). The following is done for both edges:

- The ACTIVE-EDGE structure is duplicated. That includes the active-face structures in the *leftFaces* and the *rightFaces* lists as well. The original active-edge entry will represent the lower part of the edge, and the new entry the upper part.
- The upper sub-edge is flagged as *ending* and the lower as *starting*.
- If the original edge was not a *starting* edge (i.e., it had already existed on the previous scanline), and was visible on the previous scanline, then the upper sub-edge is flagged as *visible*. If the edge is the *topDog* then the lower sub-edge is flagged as *visible* as well.
- A VERTEX structure is allocated for the cross vertex. Its coordinates are set to the X and Y of the crossing point, and the Z coordinate calculated for this edge. The *vBot* pointer of the upper sub-edge and the *vTop* pointer of the lower sub-edge are set to this vertex. If any of the upper or lower sub-edges is a SHE, then its *vTop* and *vBot* may have to be exchanged to conform to the SHE conventions (see 2.4.1). The vertex is put in an auxiliary list besides the static data structure, and will be deallocated when not needed any more.
- The original edge is removed from the AEL, and the two sub-edges are inserted into it at the correct X-intercept positions.

Recall from chapter 2 and figure 2.1 that a vertex has a list of all the edges that use it. We now create such an edge-list for the two new vertices. Even though each new vertex is connected to only two sub-edges, the list includes all *four* of them! This is because for border enhancement purposes there is no functional difference

between a cross and an ordinary vertex with four edges. The only difference is that in the case of a crossing we know the *topDog/underDog* relations, and can use them to infer visibility of one edge from that of the others.

4.2. Display of Borders

The display module gathers contiguous spans of the same visible face into a single span that ends when a span from a new face starts. Then it determines the horizontal extent of the edge-cluster that bounds the span from the right, and both extents are sent to the output device: first the face span with the corresponding shading value, and then the edge extent with with maximal contrast to the background. Pixels that are overlapped by the span and the edge, belong to the edge. This scheme covers all edge-clusters except those that bound a span from the left, and the previous visible face is NULL. In these cases the extent of the left cluster is determined as well, and the order of output is: left cluster, face span, right cluster.

If during this process, a vertex is found to be visible (if at least one edge connected to it is visible), it is marked in the vertex structure (either in the static vertex list or in the auxiliary list of cross-vertices). Vertex visibilities are used in determining visibilities of SHE's, as will be described below.

If the edge is not a *starting* edge, it corresponds to the simple case illustrated in figure 4.2(a), and the range that must be blackened for border enhancement is set by the X-intercepts of the edge with the previous and current scanlines.

If the edge is a *starting* edge, the initial range to be blackened is determined from its $xPrev$ and $xCur$ as before. Then we check all the *ending* edges that are connected to the top vertex of that edge, and if any of them is visible, the range is expanded to include the X-extent of that edge in the current swath. Let's return once again to figure 4.2 and see how this is done.

- In example (b), when span $(F - G)$ from face f_2 is sent out, we find that its left edge, F, is visible. The initial range is thus $[F{\rightarrow}xCur, F{\rightarrow}xPrev]$. Since F is a *starting* edge, we check the *ending* edges connected to vertex a, namely edge E. E is the boundary of a face that was visible on the previous scanline (f_1), so it is visible, and the right limit of the range is expanded to $E{\rightarrow}xPrev$. When span $(G - H)$ from face f_1 is sent out, edge G determines the initial range, and E expands its right limit. Notice that there is no need to send this range to the display, because it is already included in the previous border

range. The display module can detect it and prevent the extra work.

- In example (c), when span $(? - E)$ from face f_1 is sent out, the lower part of edge E is visible and sets the initial range. Now there are two *ending* edges: the upper part of E, and the upper part of F (call them *upperE* and *upperF* respectively). Since q is a crossing, we make use of the knowledge that E is the *topDog*, and F is the *underDog*:

  upperE is the other half of an edge that is visible and a *topDog*, therefore it is visible as well, and the range is expanded to $E \rightarrow xPrev$.

  upperF belongs to an *underDog* edge, and the *starting* visible edge belongs to the *topDog* edge. In such cases we have to check if the *topDog* hides this side of the *underDog*: The distance of the second vertex of *upperF* (i.e., not q) from E is found, and from the sign of the result we find that *upperF* lies on the left side of E. Now since E has a face in its *leftFaces* list (f_1), then this face hides *upperF*, and it is invisible.

  When span $(E - F)$ is sent out, and edge F is enhanced, we already know that out of the set of edges that end at q, only *upperE* is visible. But let's assume that we don't know their visibilities, and see how they are found when the visible *starting* edge is an *underDog*: *upperE* is the *topDog*, so if the corresponding *underDog* is visible, it is also visible. *upperF* is the second half of a visible *underDog*, therefore it must be invisible because the *topDog* has at least one face which will hide it.

- In example (d), when span $(I - E)$ of face f_1 is sent out, the lower part of I is enhanced, and applying the above set of rules for crossing point q, the upper part of E is found visible.

  Similarly, when the lower part of F is enhanced, the upper part of F, and the edge between r and s are found visible. Notice that edge (r, s) is a *starting* edge, therefore we recursively apply the same procedure for it and check the visibilities of the *ending* edges that are connected to s, namely the upper part of G (which is found visible), and (s, t) (which is found invisible).

  When edges H and J are enhanced, we have to find the visibility of edge K. This time it is a SHE that is connected to the *starting* visible edge with an original vertex, so we don't have the helpful information provided by a crossing. It is easy to figure out the visibility of K here, so let's look at figure 4.3, when span $(F - G)$ is sent out. The situation is similar to case (b) in figure 4.2, but this time edge H is a SHE between the original vertices a and b.

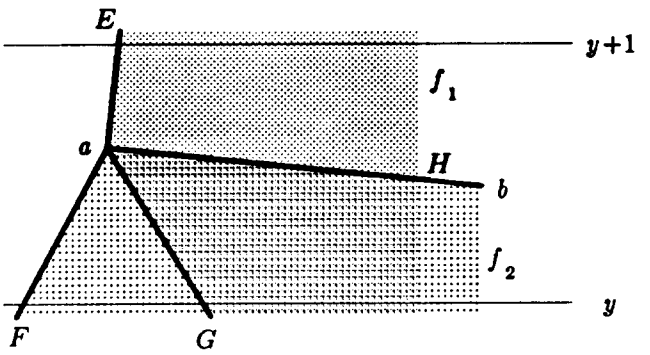


Figure 4.3. *Finding visibility of a SHE.*
F is the visible starting edge, and H is a SHE; Face f_1 overlaps H and hides it, so H is invisible.

Finding the visibility of such an edge is done in two steps. First we find whether it is *potentially visible*, which it is if it has the current visible face in any of its face lists, and/or if its two vertices are visible. Here, the visible face f_2 is also in H's *leftFaces* list, so H is potentially visible. In the second step we check if any of the faces that use the connecting vertex, hides the SHE edge. Only faces that are not in the SHE's face lists are checked, so in our example it is only f_1. For each such face the following is done:

- The contour list of the face is searched to find the two incident edges that comprise the face corner at our vertex. In our example, those edges are G and E.
- With simple trigonometry we check if the face corner overlaps our SHE. In our example, corner GaE indeed overlaps H.
- The X and Y coordinates of the second vertex of the SHE (b in the example) are substituted into this face equation to get the depth of that point on the face. If this point is closer to the viewer than the second vertex, then the face hides the SHE, and therefore the SHE is invisible. If the point lies on the face's plane then the whole edge lies in that plane and we assume that it is visible. In our example we substitute b's coordinates into f_1's face equation and find that the resulting depth is less than $b{\rightarrow}z$, so H is invisible.

o Figure 4.4(a) illustrates a situation where no edge intersects the current scanline, and as a result there is no access to the edge-cluster that consists

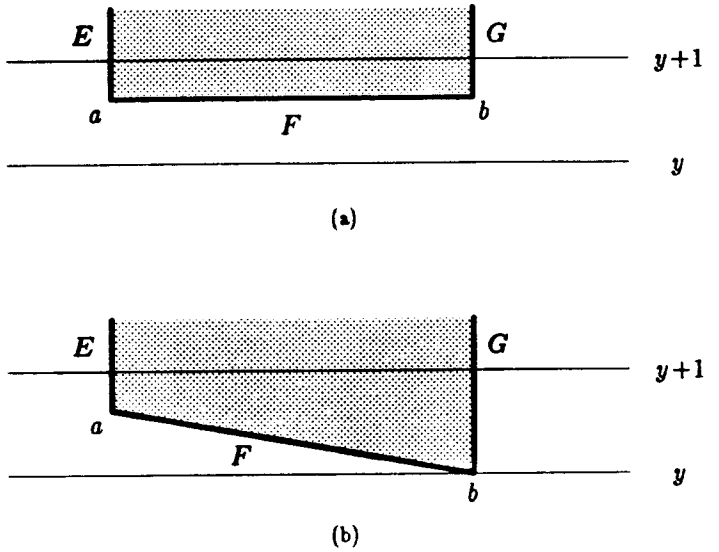


Figure 4.4.

(a) An edge-cluster with no connection to the current scanline poses a problem to edge enhancement.

(b) An equivalent construct without problems.

of edges E, F and G. Clearly this cluster is not negligible. It occurs very frequently in UNIGRAFIX scenes, and should be handled correctly. All such clusters are characterized by one or more vertices without *starting* edges, such as vertex b in this example†. The solution is to treat such vertices as if they were positioned on the current scanline. This means that all the *ending* edges of the vertex now participate in the STACK algorithm, and provide access to the cluster. In our example, treating vertex b in this way results in an equivalent situation to that described in 4.4(b); edges F and G participate in the STACK, and enable regular access to E as described in the previous paragraphs.

4.3. Borders with Intersecting Objects

When doing intersections with shaded faces only, we didn't care where exactly the intersection edge started and ended; knowing the starting and ending scanlines was enough for intersection processing.

† Notice that F is a *starting* edge of vertex a, so a is not in the above category.

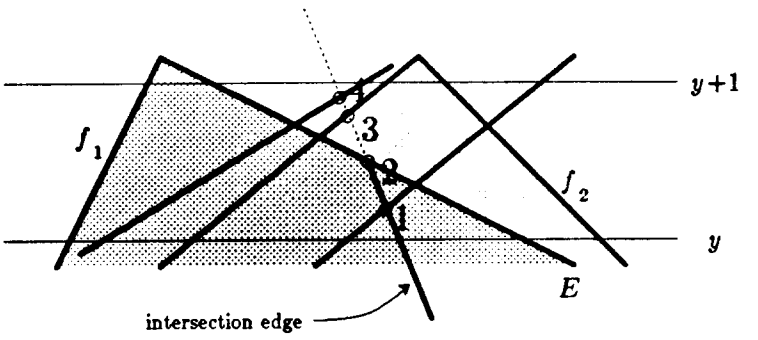


Figure 4.5. *Finding the top endpoint of an intersection edge.*
The intersection line is dotted, and becomes solid where the intersection edge starts.

Finding the bottom endpoint is relatively easy because intersection edges are terminated during the updating of the AEL. All the conditions listed in section 3.4 are checked, and when termination is detected, we find the crossing point between the intersection edge and the other edge. The edges are cut as in a normal edge crossing, and the lower part of the intersection edge is immediately removed. If there were other crossings on the intersection edge *below* the termination point, then all those sub-parts are deleted as well.

Finding the top endpoint is more difficult because it happens during execution of the STACK module, and we don't have the bubble-sort to find all the crossings for us. Instead we test the new intersection edge (call it *newIsect*) for crossings with a section of the AEL. The left and right limits of this section are chosen such that every edge that crosses the new edge is included. For each edge in that range we check if it crosses *newIsect*, and if it does, we find the crossing point and insert it into a list sorted by an ascending Y coordinate. When the list is ready, we start at the list's head (lowest crossing) and perform a cut (as in section 4.1.2) for each crossing in the list, until we get to the end of the list, or until one of the crossing edges belongs to any of the two edges that had originated the intersection. Figure 4.5 shows an example of finding the top end of the intersection edge between faces f_1 and f_2. The crossing points are represented by circles; circles 1 and 2 are crossings that were made into actual cuts, and circles 3 and 4 are crossings that occur on a higher Y coordinate than the top endpoint. That endpoint is at circle 2, and it was determined by a crossing with edge E that belongs to f_1.

Intersection edges that start and end on the same swath are a special case. Both ends of the edge are determined in a way similar to finding the top endpoint

of a normal intersection edge, and from then on this edge is treated like a SHE for edge enhancement purposes.

5

Smooth Shading and Embedded Text

This chapter describes two extensions to the hidden surface removal algorithm: smooth shading of faces using the Gouraud model, and copyfitting text onto visible faces.

5.1. Smooth Shading

Smooth shading of faces greatly enhances the realism of a picture. It can be applied to a polygonal scene, to give it a look of smooth curved faces without resorting to more complicated design methods, like spline surfaces, and to more expensive rendering tools, like ray tracers. The *Gouraud* shading model [2] was added to UGDISP with a minimal effort. This is a simple model, but more advanced models, like *Phong* shading, could be implemented in an efficient manner as well.

To achieve continuous shading across faces boundaries, shading values are computed for vertices and then interpolated twice: along the edges and along scanlines. The surface normal at a vertex is found by averaging the normals of all the faces that use the vertex. The weighting factor for each face is the angle of the face corner at that vertex. The shading value for the resulting normal is computed like that of an ordinary face normal: each ambient light source contributes its full intensity, and each directional source contributes its intensity according to the dot product between its direction and our normal.

Two more fields are added to the ACTIVE-EDGE structure to hold the shading data:

- $sCur$ is the current shading of the edge.
- $\Delta s/\Delta y$ is the increment in shading for one scanline.

The initial value for $sCur$ in a new active-edge is set to the shading value

of the *vTop* vertex of the edge, and whenever the AEL is prepared for the next scanline, it is incremented by $\Delta s/\Delta y$. This achieves the first interpolation of the shading values.

The second interpolation is done in the display module. The information to display a smooth span from face segment f_1 consists of:

- y is the current scanline.
- x_l and x_r are the left and right limits of the span.
- active edges ae_L and ae_R are the left and right boundaries of face f_1.

The *inverse shading slope* along the face segment is calculated from the bounding edges:

$$\Delta x/\Delta s = \frac{ae_R{\rightarrow}xCur - ae_L{\rightarrow}xCur}{ae_R{\rightarrow}sCur - ae_L{\rightarrow}sCur}$$

Shading values are in the range $[0,1]$. The output device has a finite number of black and white stipple patterns to represent this range, and the mapping from a shading value to the corresponding stipple index is done simply by multiplying by the largest index:

$$i = \lfloor s \cdot i_{\max} \rfloor,$$

and we thus define the *inverse index slope* to be:

$$\Delta x/\Delta i = \frac{\Delta x/\Delta s}{i_{\max}}.$$

The shading index for the left boundary of the face is:

$$i_L = ae_L{\rightarrow}sCur \cdot i_{\max}$$

Notice that i_L is not truncated to its integer part; this is because the i_L's of all spans are the key to the shading continuity across edges, and we want to keep them accurate. The indices for the beginning and the end of a certain visible span $[x_l, x_r]$ are:

$$i_l = \lfloor i_L + \frac{x_l - ae_L{\rightarrow}xCur}{\Delta x/\Delta i} \rfloor,$$

$$i_r = \lfloor i_L + \frac{x_r - ae_L{\rightarrow}xCur}{\Delta x/\Delta i} \rfloor.$$

Now starts an incremental process that sends subspans of $[x_l, x_r]$ to the device. The division to subspans is made such that each one is painted with a single stipple pattern. The limits of subspan j that corresponds to the index $i_l \le i_j \le i_r$ are:

$$x_{j_l} = ae_L{\rightarrow}xCur + (i_j - i_L)\Delta x/\Delta i,$$

$$x_{j_r} = ae_L{\rightarrow}xCur + (i_j + 1 - i_L)\Delta x/\Delta i.$$

If $i_r \leq i_j \leq i_l$ then x_{j_l} and x_{j_r} are computed with the indices $i_j + 1$ and i_j respectively.

5.1.1. Smooth Shading with Intersections

Smooth shading across an intersection edge can be be done in two ways: shade the two intersecting faces independently to enhance the intersection line, or "smooth" the intersection line by treating it like any other *seam* edge. My design decision was to choose the first way, because intersections in UNIGRAFIX are quite rare, and when they do occur, they are usually introduced on purpose and in order to see them. The second way is more suitable to a *CSG* system where intersections are more common and part of the design process.

Recall from chapter 3 that between the two intersecting faces, the face that is hiding on the left of the intersection line stays untouched, and the other face is cut into two segments. Therefore the former needs no special treatment, and its smooth shading will depend on its original bounding edges. We do however have to assign shading values to the intersection edge such that the resulting shading of the second face would be as if there was no intersection at all. This is done simply by interpolating the shading data of the original bounding edges.

Let ae_L and ae_R be the left and right bounding edges of the face segment f_2, and *Isect* be the new intersection edge. The interpolation factor for *Isect* is given by

$$\delta x/\Delta x = \frac{Isect{\rightarrow}xCur - ae_L{\rightarrow}xCur}{ae_R{\rightarrow}xCur - ae_L{\rightarrow}xCur}$$

and the shading value for *Isect* is therefore

$$Isect{\rightarrow}sCur = ae_L{\rightarrow}sCur + (ae_R{\rightarrow}sCur - ae_L{\rightarrow}sCur)\delta x/\Delta x.$$

The shading value of *Isect* on the next scanline is found in a similar manner, using the X values incremented by their respective $\Delta x/\Delta y$'s, and the shading values incremented by their respective $\Delta s/\Delta y$'s. Subtracting $Isect{\rightarrow}sCur$ from this value gives $Isect{\rightarrow}\Delta s/\Delta y$.

The resulting inverse shading slope is only an approximation, because the real value depends on the shading of the bottom vertex which does not exist at the time that *Isect* is created. However, experiments have shown that it is a good approximation, and no degradation is noticed in the shading produced.

5.1.2. Edge Enhancement

Just like border enhancement for uniform shading, it is sometimes useful to enhance some of the edges in the scene to make the picture more comprehensive. The edges I chose to enhance in this mode are any of the following:

- *Contour* edges, i.e., edges with faces on one side only.
- Edges with more than two faces.
- Intersection edges.
- Edges with one face on each side, with the dihedral angle between those two faces less that some constant. This constant is user defined, and defaults to 100°. Using the default value results in smooth uninterrupted shading of curved surfaces, and enhanced edges on corners that are sharper than slightly more than a right angle.

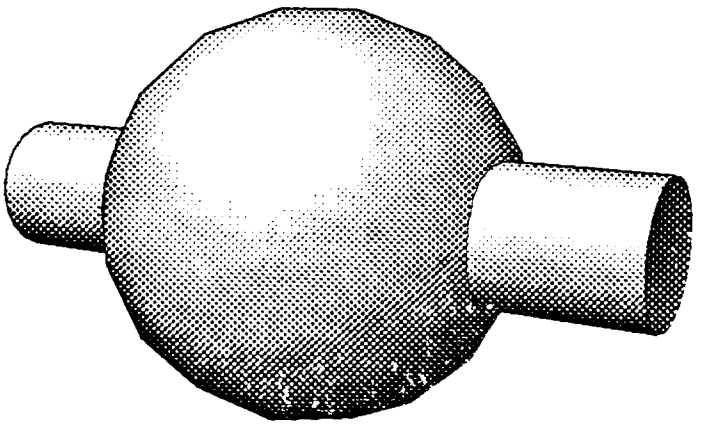


Figure 5.1. *Sphere and Cylinder.*

Figure 5.1 shows an intersection of a cylinder with a sphere. Both objects are polyhedral and smoothly shaded. The enhanced edges are the contour edges, intersection edges and the edges on the perimeter of the cylinder base which form a dihedral angle of 90° between the base and the envelope.

5.2. Embedded Text

Combining Graphics with text is common, and most text formatting systems provide some graphics capabilities. For example, the *troff* [6] family of text formatting programs is capable of drawing straight lines, curved lines and filled polygons. Pictures can be created with the *Gremlin* [8] illustrator, and converted by the *grn* [12] pre-processor to troff commands. More advanced systems like POSTSCRIPT [7] have more powerful graphics and even allow the inclusion of an arbitrary raster file into a document.

UGDISP features a similar idea but from the other direction, namely adding text to an arbitrary picture, such that the picture elements control the shape of the text paragraphs. In other words, *copyfitting* the text into specified control polygons. The STACK algorithm provides the shape of the paragraph, and the text formatting is done by TEX [4], with its unique **\parshape** command.

A *text* statement was added to the UNIGRAFIX language, specifying the input text file, the face into which the text is to be copyfitted (target face), and an optional magnification factor for the text:
t *text-id* (*text-file face-name* [*magnification*]);
The text file is a TEX source. This means that it can contain simple text, created by someone who is completely unfamiliar with text processing, or it can contain sophisticated TEX commands using its full power†.

During the display of the picture, the display module tracks the visible part of the target face, and keeps the shape of the resulting paragraph in a list of *text-lines*. Each such text-line contains the vertical position of the line (i.e., the corresponding scanline), the line indentation (i.e., the horizontal position of the beginning of the line) and the line length. There is one text-line entry for each underlying line of text. Therefore text can cover only the first span of the control face on any line, and no multiple column formatting is available. When the picture is done, the list of text-lines is translated to the **\parshape** command format, and TEX is called to format the text into the designated paragraph. The resulting *dvi* (device independent) file is merged into the raster picture and displayed. Device drivers for text currently exist only for the Sun workstation and Varian/Versatec plotters. In figure 5.2 the target face is the CRT screen. It was given a maximum illumination such that the background for the text will be white, and pushed

† A few restrictions apply though, due to the external control on the paragraph shape. See the section on **\parshape** in [4].

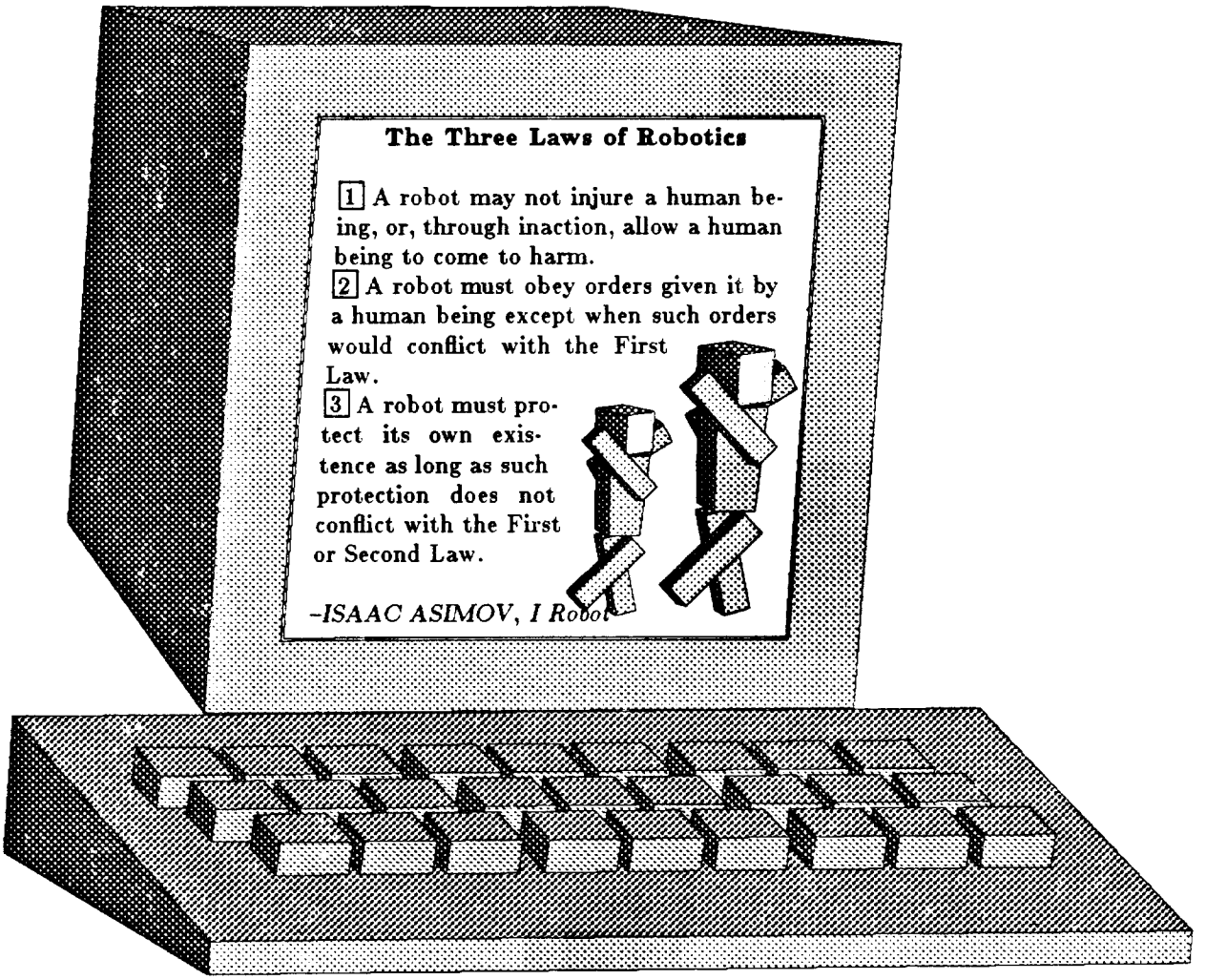


Figure 5.2. *March of the Robots.*
Design of CRT, keyboard and tobor - courtesy of H.B. Siegel.

slightly back such that the robots will hide it and the text will *wrap-around* them.

6

Evaluation and Conclusions

6.1. Run-Time Statistics

6.1.1. Comparisons with Other Renderers

The performance of the three UNIGRAFIX renderers was compared on several runs involving various scene files, display options, and output resolutions; the test scenes are shown in figure 6.1, and the results of three tests are presented in table 6.1.

- The input in test *mesh1* is a 3-dimensional mesh of 288 interlocked rings. Each ring is rectangular with a squared cross section. The file contains 4608 vertices, 6912 edges and 2880 faces. The output medium is a Versatec 36" wide-bodied plotter with resolution of 200 dots/inch, so a full size raster contains about 50 million pixels.
- The input in test *mesh2* is the same as before, but this time the plot is scaled down to an 8 inches square such that the raster size is 3 million pixels.
- The input in test *rubik-9* is a Rubik's cube with 9 cubes on each side, so there are 5832 vertices, 8748 edges and 4374 faces. The output medium is a small scale Versatec plot as in test *mesh2*.
- Each test includes three plots with different display options:

  -sf is *show faces* mode; faces are shaded without border enhancement.

  -se is *show edges* mode; only enhanced borders (i.e., edges) are displayed.

  -sa is *show all* mode; faces are shaded and borders enhanced.
- The table entries are cpu times in seconds. They reflect only the run-time of the hidden surface algorithm and do not include reading or data structure construction times. All tests were run on a VAX-750 with seven Megabytes of memory.

Renderer	*mesh1*			*mesh2*			*rubik-9*		
	-sf	-ho	-sa	-sf	-ho	-sa	-sf	-ho	-sa
UGSHOW	476	516	581	155	167	175	146	150	159
UGPLOT	541	329	557	335	304	365	304	290	331
UGDISP	1536	1927	2049	446	800	817	420	629	639

Table 6.1

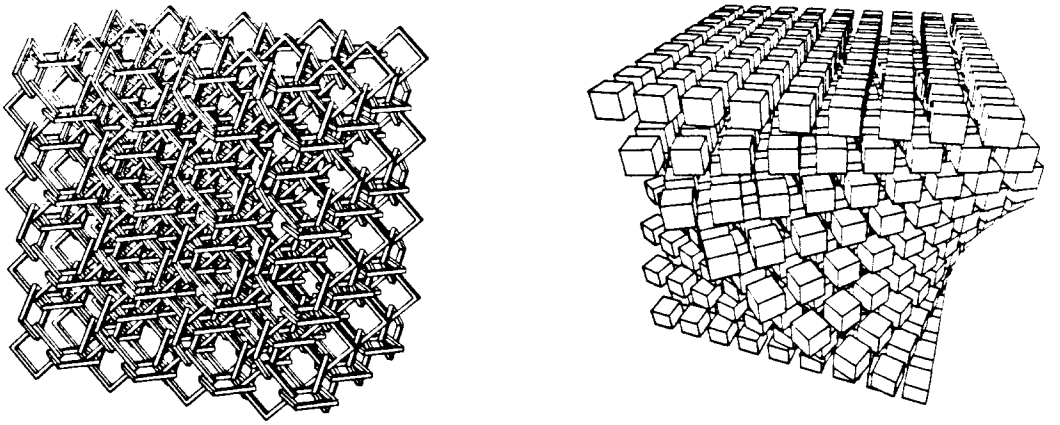


Figure 6.1. *Mesh and rubik-9 scenes for the performance comparisons tests.*

Let's first check the behaviour of each renderer with the various display options and resolutions. The big win of UGPLOT is with showing edges alone; it is fast and almost independent of the output resolution because processing is controlled by vertices and edge-crossings, not by scanlines. Showing shaded faces, however, is performed per scanline; it increases execution times and shows a clear dependency on output resolution in both *-sf* and *-sa* modes. It should be noted though, that this penalty for showing shaded faces is greatly reduced when the output device is capable of displaying filled polygons.

The two other renderers process each scanline, and consequently execution times increase strongly with output resolution. UGDISP experiences a major decrease in performance when border enhancement is done. This is caused by calculation of crossing points and edge-cutting that do not occur otherwise. The difference between *-se* and *-sa* is minor, and caused by sending the shaded spans to the output device; the rest is exactly the same.

UGSHOW demonstrates a very stable behaviour in all display modes. It is not as careful as UGDISP is with almost horizontal edges; therefore enhancing edges does not affect execution time so much, but the difference in output quality is significant: UGSHOW does not handle correctly cases like that in figure 4.2(d).

UGPLOT and UGSHOW are significantly faster than UGDISP in most cases. The difference in run-time decreases as more complicated scenes are displayed on lower resolutions, but still UGDISP is slower. A very important feature, presented in the table below, is the memory requirements of each algorithm. The numbers represent the size of the text and heap segments in Kbytes; they do not include the size of the raster files created by the renderers.

Renderer	*mesh1*	*rubik-9*
	-sa	-sf
UGSHOW	3650	9305
UGPLOT	3526	3220
UGDISP	2975	2628

Table 6.2

UGDISP uses the least amount of memory and is thus capable of displaying bigger and more complicated scenes than the others. The large memory usage of UGSHOW on *rubik-9* is due to the high degree of face overlaps in this scene; in most parts of the picture the depth of its face stacks is six to nine faces, and hundreds of those stacks are kept in parallel.

6.1.2. Evaluation of Intersection Detection

We would like to know what is the performance penalty that is incurred in UGDISP for detection and display of intersecting objects, and then how does

UGDISP compare with pre-processing of the scene with the intersection remover UGISECT and displaying it with the fast UGPLOT.

The three UGDISP columns of table 6.3 address the first question; they all give run-times in seconds of UGDISP with the *-sa* display option.

The first column presents the run-time on a scene with a number of similar objects. The objects are *disjointed*, i.e., do not intersect.

The second column presents the run-time on the same scene, but this time intersection detection is turned on. No intersections were found of course, so no intersection edge was incorporated into the data structure.

The third column presents the run-time on a scene with the same objects, but this time they are positioned in the same place, such that they intersect. Intersection detection is turned on, and all detected intersections are made into edges and added to the data structure.

To make the third case comparable with the first two, an orthogonal view was used, and the translation of the objects to the point where they all intersect is done along the eye direction. Therefore all face overlaps are exactly the same, and the AEL is identical in all three cases.

The next three columns give the run-time of UGISECT on the scene with the intersecting objects, the run-time of UGPLOT on the pre-processed scene with the *-sa* display option, and the total of both. Unlike the previous table, all figures include reading time, so real execution times can be compared to answer our second question. Reading time is significant since the pre-processed file produced by UGISECT is much larger than the original file.

- The input in test *knots1* includes two objects. Each one consists of eight knots in a diamond lattice structure. When they are placed together one is slightly rotated with respect to the other, such that many intersection lines are visible. The test was carried on a VAX-750 and the output medium was a small scale Versatec plot.
- Test *knots2* used the same input as above, but was carried on a Sun 3/160 workstation. The pictures were displayed on a window of 510×560 pixels.
- In test *pipes* there are four objects. Each one is a pipe with inner and outer tubes. When they are placed together, they all intersect at their centers. This test was also performed on the Sun workstation.

the test scenes are shown in figure 6.2.

Test	UGDISP			UGISECT	UGPLOT	UGISECT and UGPLOT
	disjoint -sa	disjoint -sa -in	intersect -sa -in		-sa	
knots1	332	600	603	3284	226	3510
knots2	88	127	135	1216	121	1337
pipes	50	84	86	479	72	551

Table 6.3

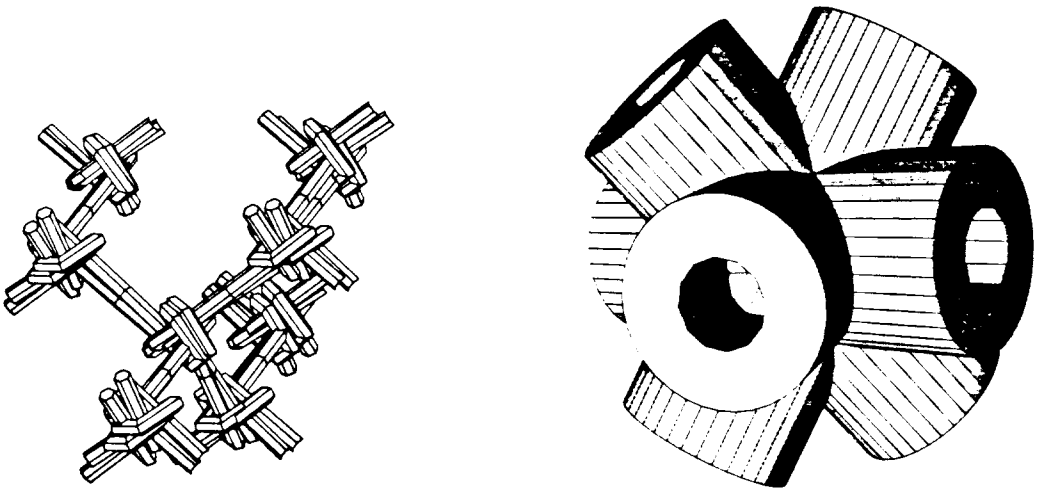


Figure 6.2. *Knots and pipes scenes for the intersection comparisons tests.*

From the table we learn that checking intersections in UGDISP increases execution times significantly. A number of additional tests indicate that the increase is usually in the range of 50–100 percent, depending on scene complexity. It is interesting to note from the second and third columns, that the number of actual intersections in the scene does not affect execution times. The reason for this is that the extra work of adding a detected intersection to the AEL is

compensated by the fact that the *Potential Penetrators Lists* of the two faces that intersected are now shorter and take less time to be traversed.

Comparing a single run of UGDISP with a single run of UGPLOT after pre-processing by UGISECT, clearly favors UGDISP. In the *knots* test UGISECT found and processed 3058 intersections, while UGDISP detected only the necessary 136 visible intersections from the given view direction. In the *pipes* test the figures were 2176 for UGISECT, and 49 for UGDISP. Generally, on a scene with N faces, UGISECT will run in $O(N^2)$ time, and produce $O(N^2)$ face intersections, while UGDISP will run in a linear time, and detect $O(N)$ intersection. This last observation is empirical, and can be explained as follows: If the N faces are big and have a high degree of overlap (as in *rubik-9*), then many of them will be hidden behind the front faces, and never become *current* visible faces. If on the other hand, the faces are small and sparsely scattered in the scene (as in *mesh*), then there are less potential penetrators for any visible face, and consequently there is less work and less visible intersections.

6.2. Conclusions

UGDISP was developed over a period of one year as a Master's project. The basics of the STACK algorithm are taken from Hamlin & Gear; intersection detection, border enhancement and text copyfitting are new algorithms, and smooth shading follows the Gouraud model.

The front and back ends of UNIGRAFIX, i.e., reading input files and driving output devices, are very similar to those of the previous renderers, and most of the code was used directly. This part of the program is implemented in 11,000 line of *C* code (7,000 input and 4,000 output). The core program is implemented in an an additional 9,000 lines of *C*. The most difficult task in this project was the border enhancement, with and without intersection detection. It still is far from perfection, and I doubt if the problem can be completely solved in the framework of a fast scanline renderer. Other modules, like the smooth shading and embedded text were straight forward and took just a few days to implement.

New options provided by UGDISP are intersection detection, fast smooth shading (a version of UGPLOT provides smooth shading through anti-aliasing, but it runs very slowly), and embedded text. UGDISP also features robustness and quick recovery from erroneous input scenes. An option that was not described in this report even allows reasonable display of very warped (non-planar) faces.

The major drawback of UGDISP, as realized from table 6.1, is its speed.

Further investigation of this issue and optimization of critical routines are called for; my prediction is that it may reduce the run-time by as much as half. As it is now, UGDISP is the best choice for an interactive design session on intermediate resolution devices, for scenes involving intersecting objects, coinciding edges and faces, and unexpected scene illegalities that always occur in the first steps of such designs. It is the previewing tool for Jessie, the interactive editor for UNIGRAFIX [15], and for UGI, the UNIGRAFIX environment shell [1]. When the design is finished, and intersections are removed with UGISECT, UGPLOT becomes the better choice for a fast plot.

References

1. Nachshon Gal, "The UGI Shell for UNIGRAFIX", To appear as Tech. report UCB/CSD , U.C. Berkeley (January 1986).

2. Henri Gouraud, "Continuous Shading of Curved Surfaces", *IEEE Transactions on Computers*, Vol. c-20, No. 6, June 1971, pp. 623–629.

3. G. Hamlin and C.W. Gear, "Raster-Scan Hidden Surface Algorithm Techniques", *Computer Graphics* (Proc. Siggraph 77), Vol. 11, No. 2, pp. 206–213.

4. Donald E. Knuth, *The TEXbook*, Addison-Wesley Publishing Company (1984).

5. Lucia Longhi, "Interpolating Patches between Cubic Boundaries", To appear as Tech. report UCB/CSD , U.C. Berkeley (January 1986).

6. J. F. Ossanna, "NROFF/TROFF User's Manual", Bell Laboratories Computing Science Technical Report 54, 1976.

7. "PostScript™ Language Manual", Adobe Systems Incorporated,

8. Barry Roitblat, "Development of the Gremlin Picture Editor", M.S. Report, Computer Science Division, U.C. Berkeley, (December 1981).

9. Mark Segal, "Partitioning Polyhedral Objects into Non-Intersecting Parts", To appear as Tech. report UCB/CSD , U.C. Berkeley (January 1986).

10. Carlo H. Séquin and P.S. Strauss, "UNIGRAFIX", *IEEE 1983 Proceedings of the 20th Design Automation Conference*, pp. 374–381.

11. Carlo H. Séquin, "Creative Geometric Modeling with UNIGRAFIX", Tech. Report UCB/CSD 83/162, U.C. Berkeley (December 1983).

12. David Slattengren, "Combining Text and Graphics: DITROFF and GRN, a Configuration and Implementation," M.S. Report, Computer Science Division, U.C. Berkeley, (February 1984).

13. Carlo H. Séquin, and Paul Wensley, "Visible Feature Return at Object Resolution", *IEEE CG&A*, Vol. 5, No. 5, pp. 37–50, (May 1985).

14. Carlo H. Séquin, "More Creative Geometric Modeling with UNIGRAFIX", To appear as Tech. report UCB/CSD , U.C. Berkeley (January 1986).

15. H. B. Siegel, "Jessie, An Interactive Editor for UNIGRAFIX", To appear as Tech. report UCB/CSD, U.C. Berkeley (January 1986).

16. Carlo H. Séquin, Mark Segal, and Paul Wensley, "UNIGRAFIX 2.0 User's Manual and Tutorial", Tech. Report UCB/CSD 83/161, U.C. Berkeley (December 1983).

17. I. E. Sutherland, R. F. Sproull, and R. A. Schumacker, "A Characterization of Ten Hidden-Surface Algorithms", *Computing Surveys*, Vol. 6, No. 1, pp. 1–55, (March 1974).

18. Paul Wensley, "Hidden Feature Elimination and Visible Polygon Return in UNIGRAFIX 2", Tech. Report UCB/CSD 84/172, U.C. Berkeley (May 1984).

Appendix A

The UNIGRAFIX Language

Command	*Keyword*	*Arguments*	
vertex:	**v**	*ID*	$x\ y\ z$;
wire:	**w**	[*ID*]	$(v_1\ v_2 \ldots v_n)$ (...) [*colorID*];
face:	**f**	[*ID*]	$(v_1\ v_2 \ldots v_n)$ (...) [*colorID*] [*illum*] [< $a\ b\ c\ d$ >];
definition:	**def**	*defID* ;	
			non-def-commands
	end;		
instance:	**i**	[*ID*]	(*defID* [*colorID*] [*xforms*]);
array:	**a**	[*ID*]	(*defID* [*colorID*] [*pre-xforms*]) *size in-xforms* [(*post-xforms*)];
text:	**t**	[*ID*]	(*text-file faceID* [*magnification*]);
light:	**l**	[*ID*]	*intensity* [$x\ y\ z$];
color:	**c**	*colorID*	*intensity* [*hue* [*saturation*]];
include:	**include**	*file-name* ;	
comment:	{	[*anything* {*nesting is OK*} *but unmatched* { *or* }] }	

Appendix B

The UGDISP manual

NAME

ugdisp - render a UNIGRAFIX scene on a screen or plotter

SYNOPSIS

ugdisp [options] [< scene]

DESCRIPTION

Ugdisp can render a scene on many possible output devices. Scene description is read from standard input, unless the **–fi** option was specified. The display is controlled by the following groups of options:

Viewing Geometry

–ep *x y z*

Eye point for perspective view from this point.

–ed *x y z*

Eye direction for parallel projection.

–vc *x y z*

View center for a perspective view; i.e., the display is centered at that point in the scene.

–va *angle*

View angle for a perspective view; must be between 0 and 180, exclusively. It defines the maximum angle of a square-based viewing pyramid, anchored at the eye point. The default view angle is 90 degrees.

–vr *angle*

View rotation. By default the y-axis points up; the displayed scene is rotated CCW around the viewing direction by *angle* degrees.

Ugdisp centers the scene and scales it to the maximal size that would still fit in the rectangle of the screen or plot. Specifying view center or view angle for a perspective view overrides this auto scaling, and the picture may occupy only part of the screen or plot. If no eye direction or eye point is specified, the default view is **–ed** 0 0 -1, i.e., an orthogonal projection from the negative z-axis.
Clipping is *not* performed with hidden feature removal, so the user should be careful when specifying a perspective view not to position the eye point too close (or inside) the scene. This will hopefully be remedied in the future.

Display Modes

-hn Hide nothing, make no visibility checks (Default).

-hb Hide back-faces, i.e., faces with face normal pointing away from eye.

-ho Hide overlaps; remove all features hidden by overlaps. Implies **-hb** .

-ab Add back-faces. Overrides any specific or implied **-hb** .

-se Show edges and wires only (Default).

-sf Show only faces without edges or wires. Implies **-ho** and **-hb** .

-sa Show faces and edges. Implies **-ho** and **-hb** .

-fw *x y z d1 d2*
Fade against white background in the interval *d1* - *d2*.

-fb *x y z d1 d2*
Fade against black background in the interval *d1* - *d2*.
x, y and *z* specify the eye point; *d1* and *d2* are distances from this eye point.

-sg Show smoothly shaded faces (with Gouraud shading). Implies **-sf** .

If the **-sa** option is combined with gouraud shading then only a subset of the edges is displayed; those edges are wires, contour edges (edges with faces on one side only), intersection edges, and edges with one face on each side and with a dihedral angle that is less then the value of some specified *corner angle*. This corner angle defaults to 100 degrees, and can be changed with the following option:

-ca *angle*
The corner angle is set to *angle* degrees. The default is 100 degrees.

-st Show text. The first text statement in the input scene is executed.

-sc Show coordinate axes.

-in Detect and correctly display intersecting faces.

-w Perform extra checks to display warped (non-planar) faces.

Labeling

-lv Label vertices. The vertex identifier is printed next to the vertex position. If **-hb** was specified (or implied) then vertices that belong to back-faces are not labeled.

-lf Label faces. The face identifier is printed on the center of gravity of the face. Faces are not labeled if hidden features are removed.

-lw Label wires. The wire identifier is printed on the center of gravity of the wire.

-le Label edges with their length.

-la Label all (vertices, faces, wires, edges).

Identifiers starting with the character '#' are not printed. If an identifier contains a '#' then only the suffix following the last '#' is printed. Some devices do not support labeling.

Output Devices

-dv Output device is a Benson Varian plotter.

-dw Versatec 36" wide-bodied plotter.

-dm Imagen printer.

-da AED 512 color display (set GRTERM to appropriate /dev/tty??).

-dx Vectrix color display (set GRTERM to appropriate /dev/tty??).

-dr IRIS graphics terminal (set GRTERM to appropriate /dev/tty??).

-di Ikonas frame buffer. A raster file called "rast.iv" is created, and can be displayed with the *iv* program.

Output is also sent to a variety of display terminals that usually serve as the user's console or tty. If the environment parameter TERM is set to the terminal's name then no device option is necessary. Otherwise (or when *ugdisp* is used from *ugi* with a permanent device option different from the console) the terminal type should be specified:

–dt Tektronix 4115 (TERM = 4115).

–dT Tektronix 4691 plotter.

–dk Tektronix 4010 (TERM = 4010).

–dK Tektronix 4107 (TERM = 4107).

–dh Hp 2648a (TERM = hp2648a).

–dS Sun microsystems workstation (Default, TERM = sun). If hidden feature removal is performed the picture is sent to the screen in 10 or more equal chunks. If only edges and wires are plotted then the picture is sent to the screen only when it is complete.

–ds Sun microsystems workstation. If hidden feature removal is performed the picture is sent to the screen scanline by scanline (it is slower than with the default **–dS** and useful only for debugging purposes). If only edges and wires are plotted then the picture is sent to the screen line by line; this is useful to preview complicated scenes before doing the hidden feature removal.

–df *frame-file*

Sun microsystems workstation. *Frame-file* is a name of a raster file that will contain the picture. The raster dimensions are 256 × 256 and it is meant to be an input for the *framedemo* program. The *uganimate* program allows easy creation of simple animations with framedemo.

If no device option is specified, and the TERM parameter is not set to any of the above terminals then a *dumb* terminal is assumed; the output is in a crude form of ascii characters.
The output to the plotters and to some of the terminals can further be controlled by setting its size:

–sx *number*

x-size of the plot is adjusted to fit into *number* inches. The default is the width of the display device.

–sy *number*

y-size of the plot is adjusted to fit into *number* inches. The default is the height of the display device. On the Varian and Versatec plotters the default is 8" and 36" respectively; specified y-size can be up to twice the default.

Files

–fi *input-file*
Read input scene from file *input-file*.

–fc *command-file*
Read options from file *command-file*.

–cm *colormap-file*
Read color map description from file *colormap-file*.

–fr *raster-file*
Put raster file in file *raster-file*. Raster files are named by default "/usr/tmp/ug??????". This is useful if there is not enough space in /usr/tmp on your machine.

–kf Keep raster file. By default raster files are deleted after plotting; with this option the raster file name is printed on standard error and the file is not deleted. This is useful if you want to plot several copies of the same scene.

–kt Keep the temporary files that are created during the processing of a text statement. Those files are "/usr/tmp/ug?????.tex", "ug?????.log" and "ug?????.dvi". The default is to delete them.

–np No plot. The raster file is not sent to the plotter.

File-names can be expressed with all the *csh* conventions except globbing, i.e., start with "~user/..." or "~/...", contain environment parameters like "$WORKDIR", contain "$$" etc. If the file is not found in the current directory, and the **–fi** or **–cm** options are used, then *ugdisp* tries to read "~ug/lib/*input-file*".
The **–fr** , **–kf** and **–np** options apply only to the Varian and Versatec plots.

EXAMPLE

```
ugdisp -ep -1 2 -10 -va 30 -sa -in -dw -sy 8 < scenefile
```

FILES

~ug/bin/ugdisp
~ug/src/ug2/ugdisp
/usr/tmp/ug??????

SEE ALSO

ugi(UG), ugisect(UG), ugshow(UG), ugplot(UG), uganimate(UG)

DIAGNOSTICS

Ugdisp prints to standard error the elapsed user and system times (in seconds) after each step in the processing. This printout is suppressed on some devices where is would interfere with the picture, and when *ugdisp* is called from *ugi*.
If the **-v** option is specified, *ugdisp* will print more detailed statistics: number of intersections in **-in** mode, number of warped faces in **-w** mode, and the cpu times of the hidden feature removal module.

BUGS

Does not clip to the viewing pyramid in perspective view, so behavior is unpredicted if the eye-point is too close or inside the scene.
Horizontal edges may be excessive or missing in **-ho** and **-sa** modes.

AUTHOR

Nachshon Gal