WORKING PAPER 66

AN APPLICATION OF LINE-LABELING AND OTHER SCENE-ANALYSIS TECHNIQUES TO THE PROBLEM OF HIDDEN-LINE REMOVAL

Mark A. Lavin

Massachusetts Institute of Technology

Artificial Intelligence Laboratory

Vision Group

March, 1974

ABSTRACT

The problem of producing hidden-line drawings of scenes composed of opaque polyhedra is considered. The use of Huffman labeling is suggested as a method of simplifying the task and increasing its intuitive appeal. The relation between the hidden-line problem and scene recognition is considered. Finally, an extension to the hidden-line processor, allowing dynamic viewing of changing scenes, is suggested. That process can be made far more efficient through the use of Change-Driven Processing, where computations on unchanging inputs are not repeated.

Work reported herein was conducted at the Artificial Intelligence Laboratory, a Massachusetts Institute of Technology research program supported in part by the Advanced Research Projects Agency of the Department of Defense and monitored by the Office of Naval Research under Contract Number N00014-70-A-0362-0003.

0.0 BACKGROUND OF THIS WORK

The work described in this paper is the outgrowth of a minor digression in a proposed project for doctoral research.

Briefly, that project involves the recognition of visual scenes under multiple views. For example, we might be interested in looking at a scene while its component parts move, or as our viewing point moves. One goal of such a project is to minimize redundant computation through the use of Change-Driven Processing <Lavin, in progress>.

Initially, the following scenario for this project came to mind: the system would "watch" a dynamically changing scene, updating its description of the scene as "significant changes" occurred in the input image. As a simple starting point, something like a "line-drawing movie" seemed appropriate. Unfortunately, the relative sloth of current line-finding programs argued against attacking the problem with "real" scenes. However, by choosing a suitable restricted domain (such as the "Blocks World") it should be possible to produce simulated "movies" using a hidden-line processor.

This practical application was the immediate impetus for designing a hidden-line processor. As the work progressed, however, two points became apparent:

(1) Given the restricted domain, certain techniques of scene-analysis, particularly Huffman Labeling, could be used to greatly simplify the hidden-line problem; at the same time, their use

greatly enhances the intuitive appeal of the program, by introducing a "Semantic of hidden-line drawings."

(2) Techniques suggested (but not yet implemented) for a possible dynamic hidden-line processor might have ramifications for the dynamic recognition process described above. In particular, it was recognized that producing hidden-line drawings (i.e., mapping 3-D descriptions into 2-D images) is in some sense the inverse of the recognition. Certain techniques developed for the hidden-line processor might, therefore, carry over to the recognition program.

1.0 THEORY OF THE HIDDEN-LINE PROCESSOR

The top-level goal of a hidden-line processor (HLP) is to transform a description of a scene, in terms of 3-dimensional coordinates, into a set of two-dimensional coordinates for lines that would be seen from some arbitrary viewing point, under the assumption that the scene is composed of opaque objects. For example:

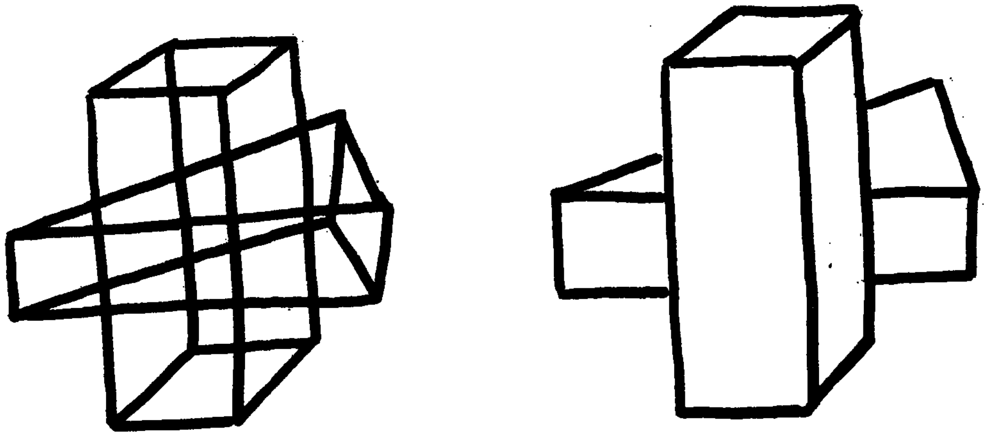

Figure 1: Hidden-Line Removal

Considerable simplification of the "Hidden Line Problem" results from imposing some constraints on the possible scene descriptions. In the current case, the scene is assumed to be composed of simple, closed, non-intersecting polyhedra, which are in turn composed of faces which are simple ("hole-less") closed polygons. Some examples of "legal" and "illegal" objects or scenes are shown in Figure 2.

The algorithm described in this paper is largely attributable to

Loutrel <Loutrel, 1970>. The significant contribution of the current work is the recognition of the duality of the hidden-line problem and scene-description problem, as discussed by Guzman, Huffman, and Waltz.

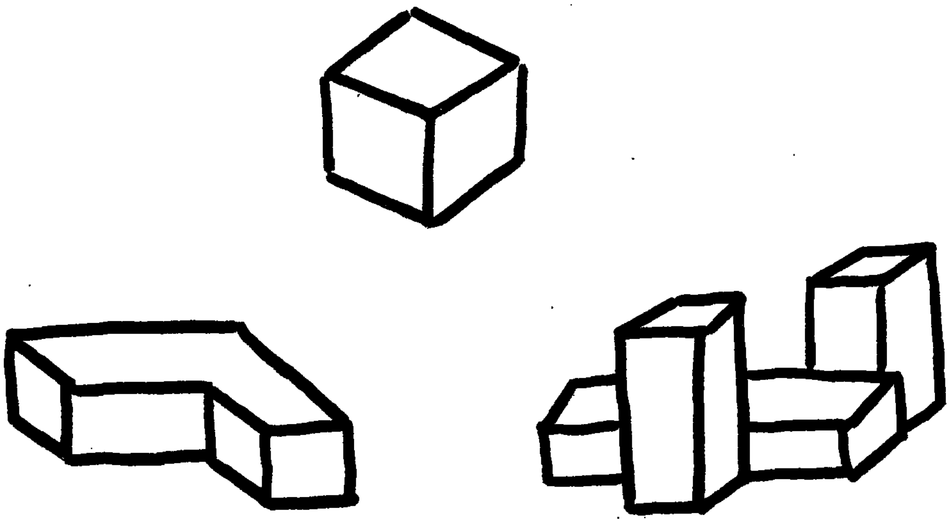

Fig 2(a): Legal Objects & Scenes

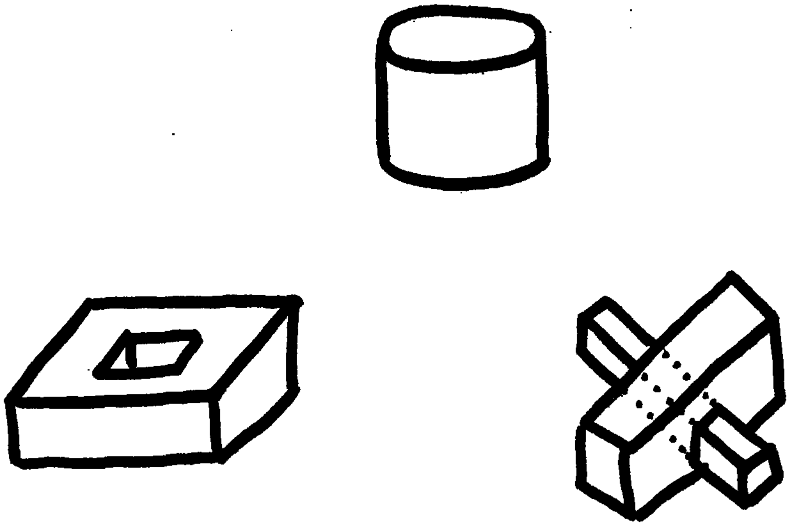

Fig 2(b): Illegal Objects & Scenes

1.1 THE NOTION OF CHANGE-DRIVEN PROCESSING

Four aspects contribute to the final nature of a hidden-line drawing:

(1) Shape of individual objects in the scene.
(2) Location of individual objects.
(3) Relative location of different objects in the scene.
(4) Location of "viewing point" and "picture plane".

The general notion of change-driven processing <Lavin, in progress> dictates that processing on invariant inputs should not be duplicated. In the current context, this means that data inferred from description of individual objects (shape and location) are not re-derived as the viewing point changes. A more ambitious application of change-driven processing to dynamic hidden-line drawings is discussecd in section 3.

1.2 OVERVIEW OF THE HIDDEN-LINE PROCESS

The first step, given a scene description in terms of 3-D vertex coordinates and surface descriptions (lists of bounding vertex names) is to produce a 3-D description of the lines in the scene. This is essentially a list of entries, one for each edge in the scene, of the form:

$$(V_1\ V_2\ S_1\ S_2\ \text{type})$$

meaning, the edge running from vertex V_1 to vertex V_2, with surface S_1 on its "left" (viewed from outside the object) and surface S_2 on its right, is of "type" type ["+" means a convex edge, "-" a concave edge, "0" a "flat" edge]. Figure 3 shows an example of an object labeled with 3-D line-types.

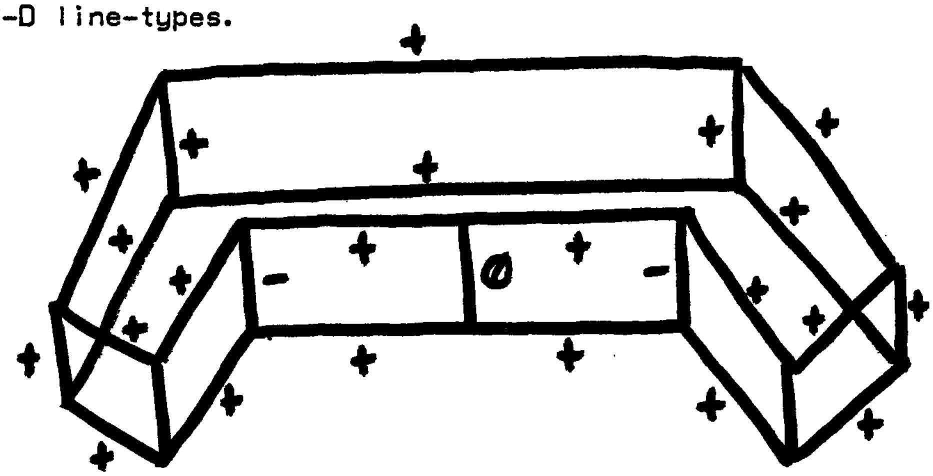

Fig 3: 3-D Line Labels

The next stage, given the particular viewing point, is to describe each "potentially visible edge" by a 2-D line description of the form:

$$(V_1\ V_2\ S_1\ S_2\ \text{type})$$

where V_1, V_2, S_1, and S_2 are as above, and a new type ">" ("obscures") is introduced. These labelings are analogous to the Huffman +, -, and > labelings (the only difference is that the "real" surface of a >-type line lies on the left, looking in the direction of the arrow). After this stage, the above drawing would be labeled as in Figure 4.

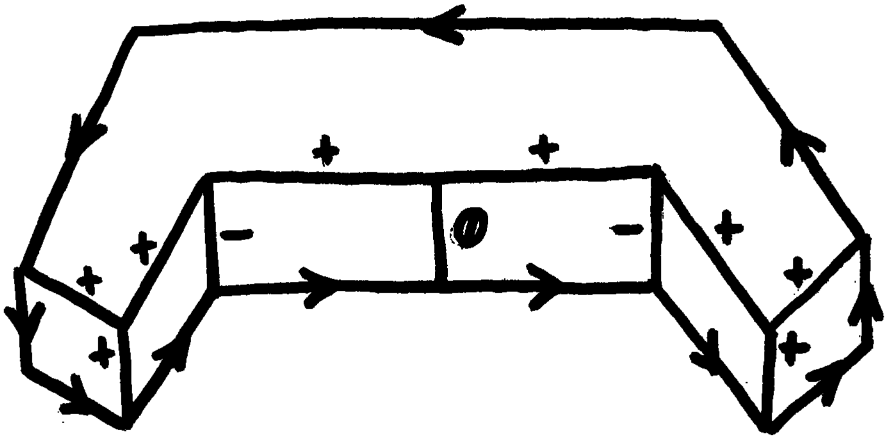

Fig 4: 2-D Line Labels

Notice that at this stage, a number of invisible edges have already disappeared (in fact, for scenes that are composed of a <u>single</u>, <u>convex</u> polyhedron, these "potentially visible edges" are exactly the final line drawing).

At this point, the effects of inter- and intra-object obscuration must be taken into account. To do this, we define (after Loutrel) for every point on a potentially visible edge, an obscuration number [OBSCUR], a non-negative integer representing the number of visible surfaces hiding that point from the viewing point. The last part of the process entails starting at some vertex, and obtaining its OBSCUR. From there, we "crawl" along all potentially visible edges (PVE's). Each time an edge crosses (in 2-D projection) a >-type line, we check whether OBSCUR changes. All segments between "real" vertices and >-crossings are noted, and those which have OBSCUR's equal to zero are non-hidden, and thus appended to the final display list. In case not all PVE's form a connected set, a new starting point is chosen and the proces repeated until all PVE's have been accounted for. The output of this stage is a list of all visible segments, each of the form:

$$((X_{i1} \ Y_{i1} \ Z_{i1}) \ (X_{i2} \ Y_{i2} \ Z_{i2}))$$

where X_{ij} and Y_{ij} are 2-D display coordinates and Z_{ij} is the "depth"--distance in front or behind the picture plane. At this point, the final drawing can be output [note that the Z coordinates might be used to modulate intensity].

The stages of processing are described in more detail in subsequent sections.

1.2 DETAILED DESCRIPTION OF HLP STAGES

1.2.0 3-D Scene Description

The data relating to a particular scene all reside on the property list of some atom which effectively names the scene. As processing continues, new properties (to be described below in appropriate sections) are added to the property list. Initially, the scene descriptions consist of two properties:

(1) VERTICES-3D: A list of elements of the form:

(VNAME X Y Z)

which indicates that vertex "VNAME" (a non-negative integer, for historic reasons) has absolute 3-D coordinates X, Y, and Z.

(2) SURFACES: A list of elements of the form:

(SNAME V_1 V_2 ... V_n)

indicating that surface "SNAME" (which has the form S<n>, where n is a positive integer) has vertices named V_1, V_2, ..., V_n. Note that the order is such as to circulate counterclockwise around the surface when seen from the "outside" of the object (which is unambiguous if the object is closed); see Figure 5 for an example.

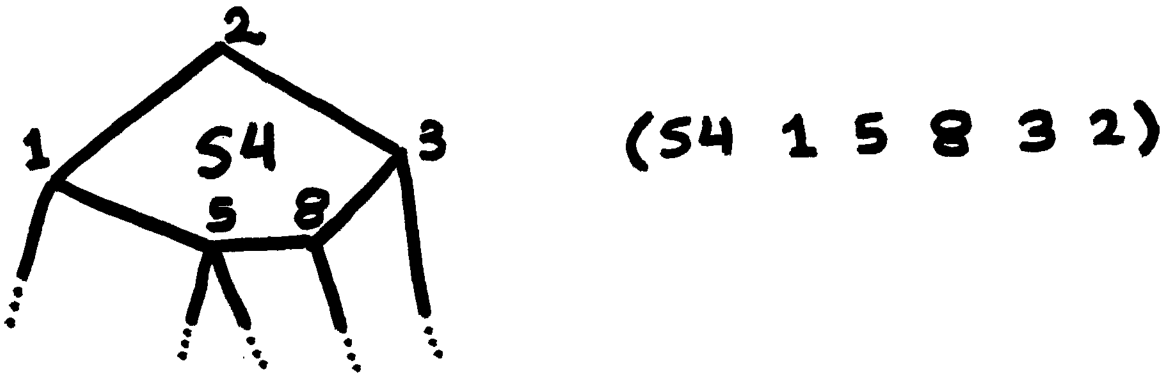

Fig. 5: Surface Naming Convention

1.2.1 SETUP-OBJ

This procedure performs view-point-independent calculations on the scene. This entails two stages:

1.2.1.1 Surface Orientation

For each surface in the scene, an outward-pointing normal vector (as shown in Figure 6.) is calculated and stored as an entry in the list associated with the ORIENTATION property.

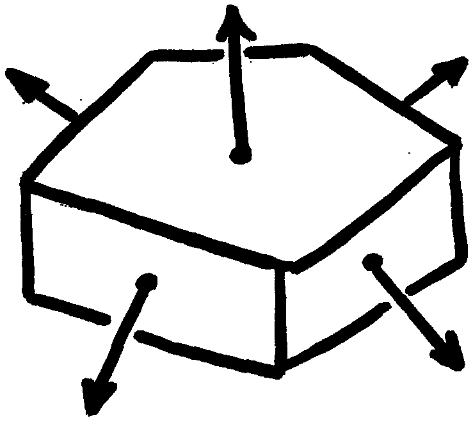

Fig. 6 : Outward-Pointing Normals

Each entry is of the form: (SNAME (X Y Z)), specifying that surface "SNAME"'s outward-pointing normal vector has components X, Y, and Z.

1.2.1.2 3-D Line-Typing

With surface orientations determined, we next assign a type [+, -, or 0] to each edge in the scene. First, all edges are collected by tracing around surfaces. Each line is then represented by a form:

$$(V_1 \quad V_2 \quad S_1 \quad S_2)$$

which states that the edge from vertex V_1 to vertex V_2 "sees from outside" surface S_1 on its left and surface S_2 on its right, as shown in Figure 7.

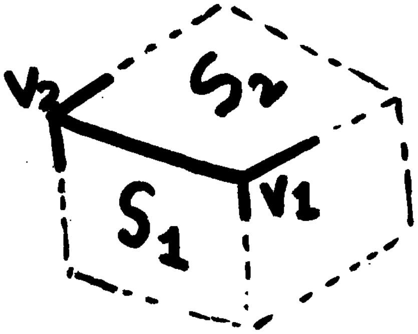

Fig. 7: Line/Surface Convention

The TYPE ["+" for convex edges, "-" for concave edges, and "0" for "flat" edges] is then calculated and an entry of the form

$$(V_1 \quad V_2 \quad S_1 \quad S_2 \quad \text{TYPE})$$

is stored under the property LINE-TYPES.

1.2.2 3D-->2D Projection

Next, the effects of selecting a particular viewing point and direction are calculated.

1.2.2.1 Projection

For each 3-D vertex specification of the form (VNAME X Y Z), a new form (VNAME X' Y' Z') is calculated (using the projection algorithm described in Appendix A). X' and Y' represent the 2-D coordinates of the point on the picture plane, and Z' its "depth" with respect to the picture plane. These entries are stored under the property "VERTICES-2D."

1.2.2.2 Visible Surfaces

Next, we determine which surfaces are "visible" (i.e., which we view from the outside). This is done by taking the dot product of the orientation vector and a vector from any vertex on the surface to the viewing point (>0 implies visible, <0 invisible). The surface <u>names</u> are divided into two bins and stored under the properties "VIS-SURF" and "INVIS-SURF."

1.2.3 Potentially Visible Edges (PVE's)

Next, we determine the 2-D line-types ["+" for convex, "-" for concave, "0" for "flat", and ">" for "obscures"] of all edges, using the following table:

3-D LINE-TYPE	Surfaces Visible	2-D Line-type
+	both	+
+	one	>
+	neither	not visible
-	both	-
-	one	not visible
-	neither	not visible
0	both	0
0	neither	not visible

A list of all resulting "potentially visible" line entries of the form (V_1 V_2 S_1 S_2 2D-type) are stored under the "VIS-LINES" property.

Let us explore the above table in relation to Huffman labeling: Suppose we acknowledge the existence of 3 labelings for an edge in a 2-D scene: +, -, and >. What kind of transformations are possible for the labeling of a particular line as the viewing point shifts? Note that the type will change only when the viewing point passes through the plane of one of the surfaces bounding the edge. In the current context, this means that one of the surfaces goes from VISIBLE to INVISIBLE or vice-versa. Then the following label transformations are all that are possible:

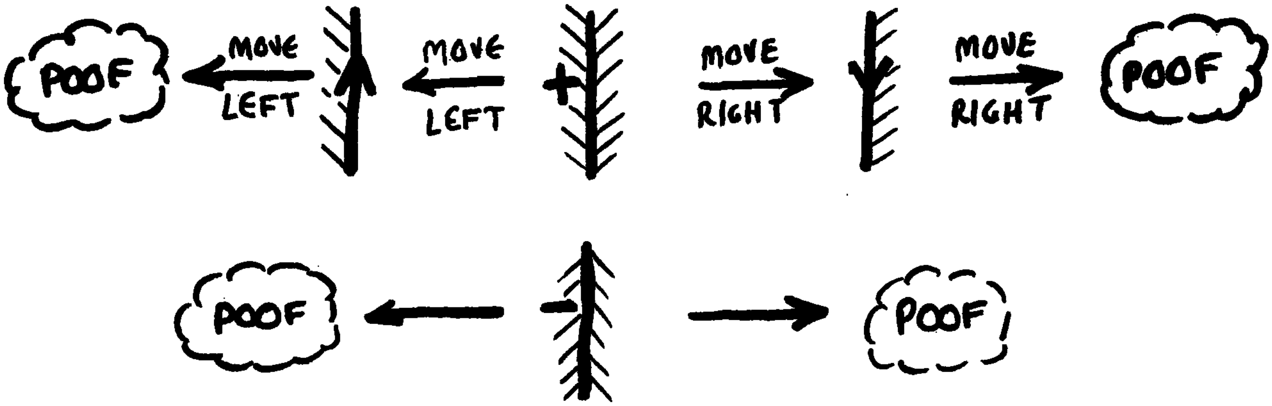

Fig 8: Label Transformations

denotes that the line has "disappeared" (is no longer potentially visible).

Potential visibility is a <u>local</u> property of edges and their bounding surfaces. It is a necessary <u>but</u> <u>not</u> <u>sufficient</u> condition for "ultimate visibility" (presence of all or part of the edge in the hidden-line drawing). In particular, note that potential visibility is a function of the shape and location of single objects, that is, the relative positions of multiple objects have no effect on it.

1.2.4 Testing for Non-local Obscuration

At this point, the "VIS-LINES" property is a list of all potentially visible lines. Now, we must account for the effects of self-obscuration (Figure 9a) and inter-object obscuration (Figure 9b).

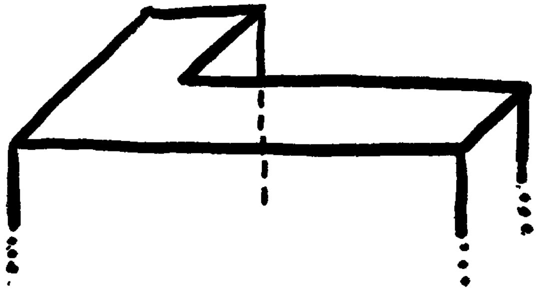

Fig. 9(a): Self-Obscuration

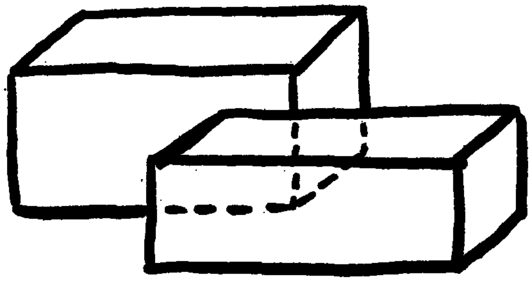

Fig. 9(b): Inter-Object Obscuration

The algorithm which does this can be described at several levels (increasing in obscurity [ha, ha] and effective computability):

LEVEL 0: Display all segments of all PVE's which are visible (a rather gratuitous starting point).

LEVEL 1: Display all segments of all PVE's which do not lie "behind" visible surfaces [note that we've already cut down the amount of processing by considering obscuration from visible surfaces only].

LEVEL 2: Starting at some vertex, crawl along an edge, noting each time the edge enters into or emerges from the obscuring "shadow" of a visible surface. Record all segments which were traversed with no surfaces obscuring them.

At this point, let me re-introduce the notion of obscuration number (OBSCUR) due to Loutrel. For any point on a Potentially Visible Edge, OBSCUR is the number of visible surfaces lying between that point and the viewing point. Further, let me introduce DELTA-OBSCUR, indicating the change in OBSCUR which occurs as we move along a line and cross the boundary of a visible surface. DELTA-OBSCUR is 0 if the line lies in front of the surface, -1 if it lies behind and we are "emerging from the surface's shadow, and +1 if it lies behind and we are "entering the surface's shadow."

<u>LEVEL 3</u> : Start at a vertex and compute OBSCUR. Now, crawl along an edge, noting each DELTA-OBSCUR and updating OBSCUR by adding DELTA-OBSCUR to it. Record all segments traversed when OBSCUR = 0 as being visible.

Two more interesting facts can be used to reduce still further the amount of processing:

(1) Only >-type lines can result in a non-zero DELTA-OBSCUR. For +, -, and 0 edges, the surfaces on both sides of the edge are visible; thus, crossing behind such edges does not change the obscuration number.

(2) OBSCUR's are "conserved;" that is, given the OBSCUR of vertex i, we can calculate the OBSCUR of vertex j by adding to OBSCUR(i) all DELTA-OBSCUR's observed in crawling along edge i,j. In some sense, OBSCUR is propogated through the network of potentially visible edges in a fashion like constraints are propogated through a hidden-line drawing in Waltz' analysis process.

LEVEL 4: Consider the following "flow-chart":

(1) Compile a list of all PVE's that haven't been traversed. Select from these the closest vertex and compute OBSCUR. Place this at the head of an "open vertex list" (OVL). If no PVE's remain, we're done!!!!

(2) Select a vertex from the head of the OVL (deleting it therefrom). Call this the open vertex. If the OVL is empty [which would result from non-connected sets of PVE's], go back to step 1.

(3) Select a line containing the open vertex from the list of remaining PVE's, deleting it therefrom (if there are no such lines, go back to step 2). Crawl along that line, noting all DELTA-OBSCUR's (see Appendix B for the method) and visible segments (where OBSCUR = 0). Add the end-point of the line and the incrementally computed OBSCUR to the end of the OVL. Repeat step 3.

At this point, we have a list of all "really visible segments" in the form:

$$((X_1 \quad Y_1 \quad Z_1) \quad (X_2 \quad Y_2 \quad Z_2))$$

which are the 2-D coordinates of the endpoints. We can now pass this list to a suitable display routine.

2. BUGS IN THE CURRENT SYSTEM

At this point, I examine some embarrassing inadequacies in the present system described above (let he who hath not resistance cast the first Rheostat).

2.1 The Accidental Alignment Problem

Profound hassles arise when a vertex of a PVE lies on another PVE (in 2-D projection). [The reader is advised to read Appendix B, on the calculation of DELTA-OBSCUR, before proceding.]

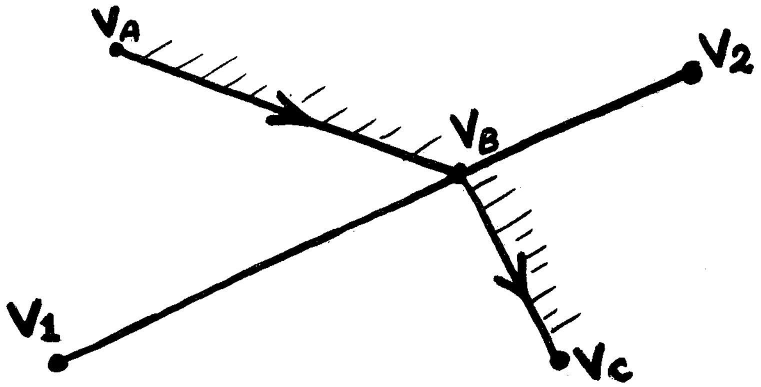

Fig. 10: A Pseudo-Psi Vertex

Suppose we have the situation shown above. Further, suppose we are crawling along from V_1 to V_2. At point V_B, we record two line-crossings: V_1-->V_2 crosses V_A-->V_B and V_B-->V_C. Thus, we record a DELTA-OBSCUR of +2 rather that the appropriate +1. Some patch, such as checking for duplications like this (which constitute pseudo-PSI vertices) could alleviate the problem. Note that the situation shown in Figure 11 is right since the resulting DELTA-OBSCUR is correctly 0.

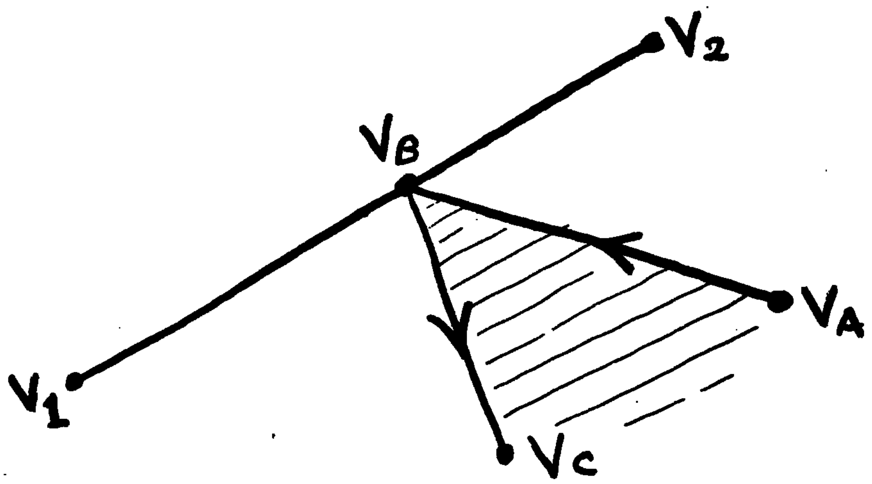

Fig. 11: Pseudo-K Vertex

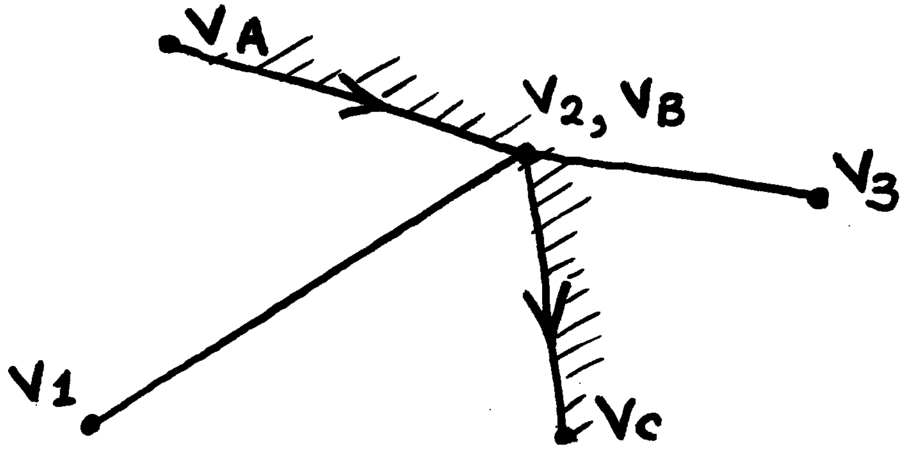

Fig. 12: Pseudo-X Vertex

Vastly more profound lossage occurs when we are following an edge which a line-crossing at one of its endpoints (yes, the inverse of the above problem). In that case (see Figure 12), the conservation of OBSCUR may not apply, and we must recalculate an appropriate OBSCUR for each edge radiating from the terminal vertex. Note the that accidental alignment problem is particularly aggravated by the petty predilection of foolish robots to stack blocks in neat piles. Perhaps the answer lies not in "correcting" the HLP, but improving the inherent creativity of the robots ("Foolish consistency is the Hob-Gobblin of little minds"--Ralph Waldo Emerson).

Accidental alignment arises from several sources: First, non-contacting vertices and edges may align because of a particular choice of viewing point. In schemes like Huffman labeling, such coincidences are precluded by demanding that scenes be viewed in "general position." A second case occurs when vertices and edges (or vertices and vertices) actually touch in the 3-D scene. In this case, a labeling scheme with "crack" line-types must be introduced (which is also beyond the scope of the original Huffman labeling). Thus, the problem of accidental alignment in the hidden-line problem has a real precedent in the scene-labeling process.

2.2 Limitations on the Hidden Line Processor

In this section, I consider the result of the constraints on the type of scenes allowed by the current Hidden Line Processor, and suggest some possible fixes.

2.2.1 The Hole-less Surface Restriction

This restriction would rule out the closed polyhedron shown in Figure 13.

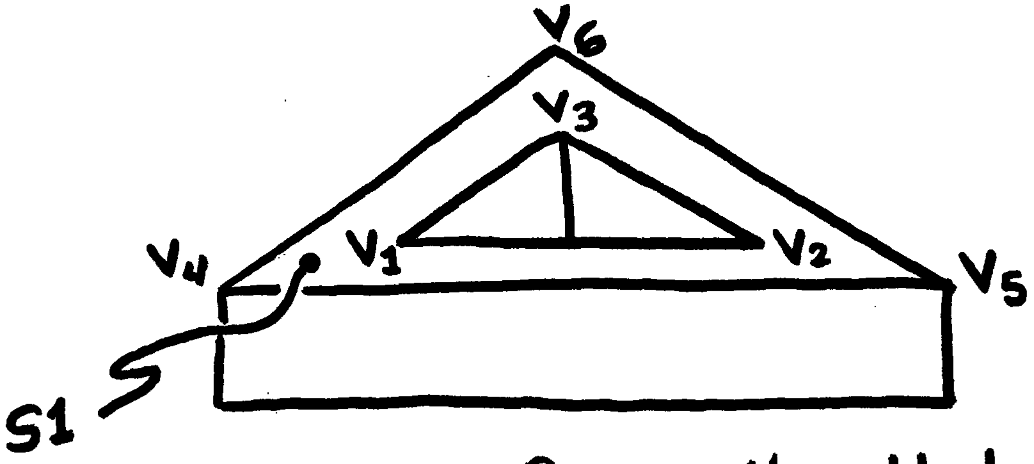

Fig. 13: A Surface with a Hole

The problem is that S_1 is effectively bounded by two polygons, V_4-->V_5-->V_6 and V_1-->V_3-->V_2. I believe that this could be fixed rather easily by allowing surface vertex lists to be segmented in the following format:

(SNAME (VLIST$_1$) (VLIST$_2$) . . . (VLIST$_n$))

where VLIST$_1$ is a list of vertices bounding the outside of the surface and VLIST$_i$, i>1, is a list of vertices bounding an interior hole (note that in these lists, an entry (...V_i, V_j ...) is appropriate iff the "stuff" of the surface is on the left of the line V_i-->V_j when viewing the surface from "outside").

This would result in the production of extra LINE-TYPE and VIS-LINE entries in a manner consistent with the present system. The only other difference would be in the calculation of OBSCUR, where we would have to check whether a point hidden by the outside edge of a simple surface nonetheless peeks through a hole in that surface.

2.2.2 The Closed-Surface Restriction:

The requirement that all objects in the scene be closed polyhedra rules out the following object:

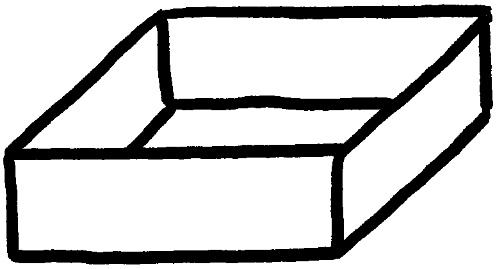

Fig. 14: A Non-closed Object

Since much of the economy of the current system is predicated on the notion of visible vs. invisible surfaces (normal vector pointing toward or away from the viewing point), accomodation of this class would require profound restructuring of the entire system. Of course, the scene-analysis programs of Guzman, Huffman, and Waltz can't handle this situation, either. Note that a sleazy solution to the above scene (provided that surfaces with holes are allowed) would be:

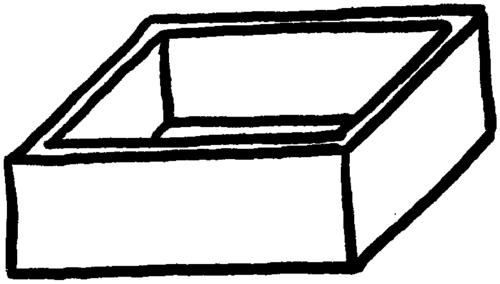

Fig. 15: Sleazy Solution

3. EXTENSIONS: THE DYNAMIC HIDDEN LINE PROCESSOR

The problem to be considered here is: given a scene description, we wish to produce a series of hidden-line drawings which would result from movement of the viewing point (or the scene or any of its component objects) along some specified trajectory; in short, a hidden-line "movie." Given the existence of a HLP, the brute-force solution is obvious: construct a series of hidden-line drqwings *ab initio* for each successive frame [!]. This is clearly abhorrent to the notion of change-driven processing, and in this section I consider some ideas for an alternative solution.

3.1 A Fundamental Conjecture

Consider two hidden-line views of the same scene from slightly different viewing points:

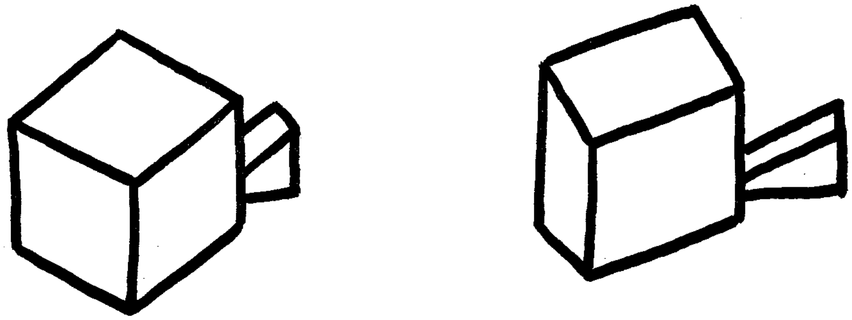

Fig. 16: Two Views of a Scene

I will state that these two views are topologically equivalent but not geometrically equivalent. Geometric equivalence (GE) implies complete identity of the 2-D display lists. Topological equivalence (TE) implies an isomorphic relation such that all points and segments connecting them in one drawing are present in the other. However, the 2-D coordinates of related elements may not be equal. 2-D images related by translation, rotation or scaling are thus TE. The following conjecture is at the heart of the proposed dynamic hidden line processor (DHLP):

> As the viewing point of a scene changes, the resulting hidden-line views always change geometrically, but, with high probability, are topologically equivalent.

An efficient DHLP, one in which the topology of a scene can effectively be "decoupled" from its geometry (as described below), will exploit the ramifications of this conjecture.

3.2 "Logical" Display Lists

Suppose we have the following two "scene fragments":

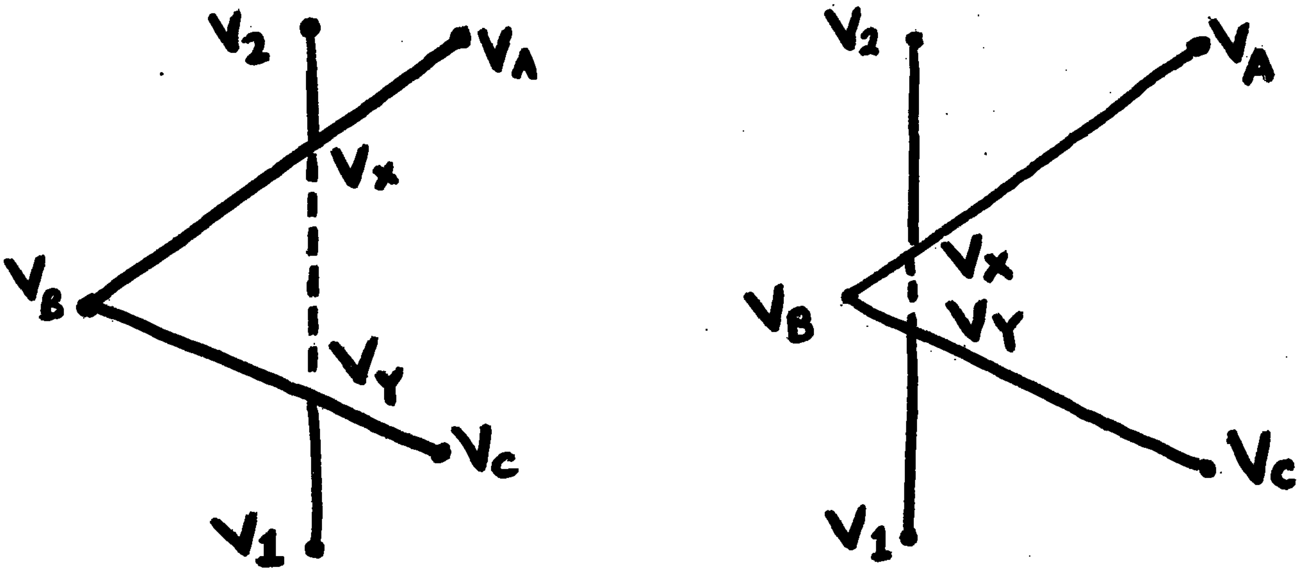

Fig 17: Two Scene Fragments

The two fragments are not geometrically equivalent, but they are topologically equivalent. Both can be described by the following "logical display list" (LDL):

VERTICES = (V_1 V_2 V_A V_B (V_X V_1 V_2 V_A V_B) (V_Y V_1 V_2 V_B V_C))

SEGMENTS = ((V_A V_B) (V_B V_C) (V_2 V_X) (V_Y V_1))

Note the additional specification of the "virtual vertices" V_X and V_Y in terms of the endpoints of the lines which intersect to form them. When it comes time to actually display this fragment, we merely "instantiate the geometry;" that is, for real vertices, substitute their 2-D projections, and for virtual vertices, the calculated 2-D locations. Thus, we have effectively "decoupled the topology from the geometry."

3.3 Changes and Demons

In general, the original logical display list [LDL] must be built up "from scratch" using the logic of the present Hidden Line Processor [for an interesting alternative, see section 3.6]. The nub of the argument is that the rather expensive step of redefining the LDL need only be executed infrequently, when the topology of the 2-D scene changes. The "geometric instantiation" should be a relatively low-cost operation.

The crux of the problem is: when do we update the logical display lists? The answer "when the 2-D scene changes" will lead to nothing but circularity and remorse. Ultimately, what we'd like is some set of "demons" embedded in the DHLP which watch for certain kinds of changes; when these occur, recalculation of the LDL is executed. Although this is highly tentative, let's consider two examples of such demons:

(1) <u>Surface Demon</u>: Associated with each surface in the scene is a "surface demon" which "interrupts" when that surface changes from visible to invisible or vice-versa. The possible ramifications of such an interrupt are as follows: Visibility changes-->potential visibility of all bounding edges changes-->positive deletion or "tentative addition" of these edges (and sub-segments) from or to the LDL. The demon itself is implemented by checking for change of sign of the dot product of the surface's normal vector and a vector from one of its vertices to the viewing point (quite a simple computation).

(2) <u>2-D Vertex Demon</u>: This demon, associated with every vertex, interrupts each time that that vertex crosses from one side to the other of a line-segment in the 2-D projection. This is a method of handling changes in inter- and intra-object obscuration. It could be implemented by checking the cross product of the edge-vector and a vector from one of the edge's end-points to the vertex in question. Generally, the effect of such an occurrence is to add or delete virtual vertices, affecting all line segments containing them in the LDL.

3.4 Singularities and Demon Priming

As noted above, singularities due to accidental alignment in the current HLP lead to great lossage. In the proposed DHLP, this bug could be feature-ified by exploiting such singularities as "priming mechanisms" for the demons. Consider the proposed vertex demon. Clearly, any movement which results in a vertex aligning with some in a 2-D scene will immediately be followed by movement of that vertex across that edge, thus causing an interrupt. Since these singularities can be caught rather easily (perhaps even by the existing interrupt hardware like "divide fault"), the complexity of the demons may be reduced.

3.5 Localization of Changes

A second tenet of change-driven processing states that if possible, when inputs to a process change, compute only the difference in the output. This is predicated on the assumption that the mapping performed by the process from input to output is to some extent decomposable. In terms of the current proposal for a Dynamic Hidden Line Processor, this has the following ramification: When a demon interrupts, it should be capable of specifying not only a potential locus of change in the LDL, but also some sort of "fence" past which the changes cannot propogate. In the optimal case, this would mean that interrupts would (1) occur rather infrequently, and (2) have only limited ramifications which are thus easily calculated.

3.6 Appearances, Disapperances, and Initialization

There is some question as to what extent the above suggestions for a Dynamic Hidden Line Processor are based on an assumption that the scene changes smoothly and continuously. It may be the case that more radical interrupts are necessary when a discontinuous change, such as the appearance or disappearance of an object, occurs. On the other hand, the "fence" idea suggested above may come into play to limit the extent to which such an occurence affects the scene.

Berthold Horn has suggested an interesting consequence of being able to handle appearance and disappearence. As mentioned above, it would seem necessary to begin a dynamic hidden-line drawing with a relatively "brute force" pass with an Hidden Line Processor. Perhaps the initial Logical Display List could also be built by actuallly "constructing" (in a Robotic sense) the scene. For example, the component objects could "appear" in the distance, and then be moved into their appropriate locations in the scene. As they are moved, the Change-Driven discipline could be used to update the scene, resulting finally in the appropriate initial configuration.

4. CONCLUDING REMARK

The purpose of this paper is to show the relation between the hidden-line problem and a technique used in scene analysis--line-labeling. As such, i make no pretentions about the relative merit of the current program for practical applications. The reader is advised to consult the excellent survey by Sutherland, et. al. (<Sutherland, 1974>), for a comparison of various hidden-line programs. Among these is the program by Loutrel, which bears strong resemblance (and, perhaps, performance) to the current work.

APPENDIX A: THE PERSPECTIVE PROJECTION ALGORITHM

A critical step in the hidden-line processing is the mapping of 3-D coordinates specifying scene elements into their corresponding 2-D "picture coordinates." To do this, the user must specify three entities:

(1) V_{eye} (X Y Z): The 3-D coordinates of the "eye."

(2) V_{gaze} (X' Y' Z'): The 3-D coordinates of the origin of the picture plane [i.e., the point V_{gaze} is mapped into 2-D coordinates (0.0 0.0 0.0)].

(3) SCALE: An arbitrary scalar magnification factor for the 2-D image.

The basic strategy is illustrated as follows:

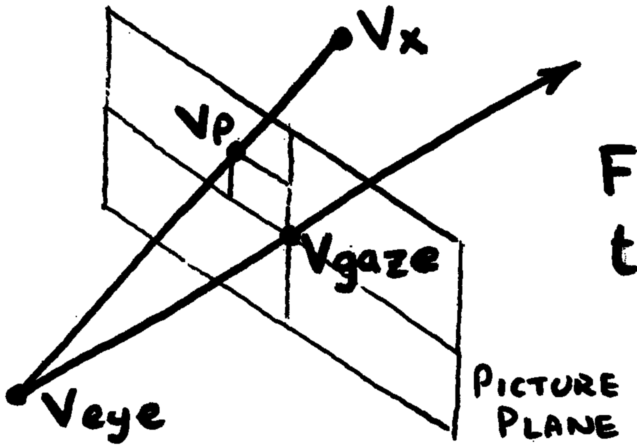

Fig. 18: 3-D to 2-D Transform

V_x is the point to be mapped into 2-D. V_p is the point of intersection of the picture plane (with origin at V_{gaze}) and the ray from V_{eye} to V_x. Thus, V_p represents the 2-D "image" of V_x. The 2-D coordinates in the picture plane (multiplied by SCALE) are the X and Y values produced by the mapping. An additional Z coordinate, corresponding to the "depth" of V_x with respect to the picture plane, can be found by projecting the ray from V_{eye} to V_x on the ray from V_{eye} to V_{gaze}.

Note that there is some ambiguity: the picture plane's origin is specified, but it could rotate around the ray from V_{eye} to V_{gaze}. To resolve this, we make an assumption that the viewer's "eyes" are "horizontal," that is, the X-axis of the picture plane is parallel to the X,Y plane in 3-D.

The projection algorithm produces a perspective transformation, which may result in an undesirable degree of fore-shortening. To avoid this, the V_{eye} point may be moved back "far" from the scene, and SCALE increased to compensate for size change.

APPENDIX B: EDGE-CRAWLING AND CALCULATION OF DELTA-OBSCUR

In this section, I consider how the DELTA-OBSCUR factor (change in obscuration number) is calculated as we move along a potentially visible edge [PVE] in the scene. As mentioned above we need only consider the cases where the edge in question crosses "behind" a >-type edge.

As we crawl along a PVE, we test for a possible crrossing with every >-type edge (some economies could result by partitioning the scene into "buckets", although I haven't attempted this). Suppose we are crawling along a PVE from V_1 to V_2, and testing for a possible DELTA-OBSCUR due to >-type PVE from V_A to V_B. Two tests are involved:

(1) 2-D Intersection: Does the projection in the picture plane of the edge from V_1 to V_2 (call it E_{12}) intersect the projection of the edge from V_A to V_B (call it E_{AB})? If it doesn't, we don't have to consider this case further.

(2) Relative Depth: If they do cross, does E_{AB} lie in front of E $_{12}$ at the point of intersection? [Note how the inclusion of a Z coordinate in the perspective transformation facilitates this test.] If E_{AB} lies in front, then E_{12} is moving into or emerging from the "shadow" of a surface bounded by E_{AB}.

If a crossing has been detected, we must decide whether we are entering the shadow (DELTA-OBSCUR = +1) or leaving it (DELTA-OBSCUR = -1). The method of doing this is suggested by the following picture:

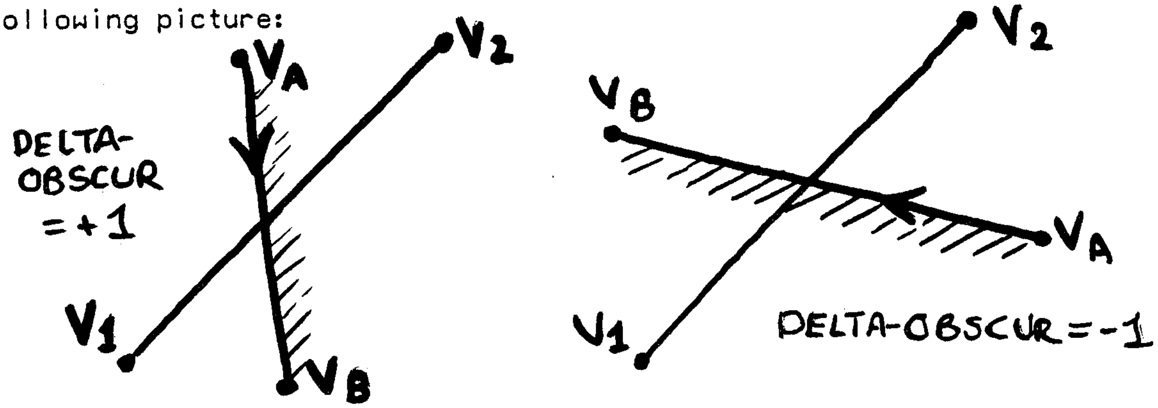

Fig. 19: Calculating DELTA-OBSCUR

Note that the test is easily performed by taking the cross-product of E_{AB} and E_{12} (the resulting sign determines the sign of DELTA-OBSCUR).

Note that the process described above may seem relatively arduous. In fact, several shortcuts can be applied. These generally applying stronger "sufficiency" tests to check for possible >-crossing. For example, we might check to see whether both V_1□ and V_2 lie in front of V_A and V_B (if so, the test need proceed no futher). As mentioned above, considerably more savings could be realized if we could partition the edges into disjoint buckets (perhaps on the basis of projected X and Y coordinates) so that checking for DELTA-OBSCUR would only involve checking for crossings of >-type lines in a given bucket.

BIBLIOGRAPHY

Loutrel, Phillipe P., A Solution to the Hidden-Line Problem for Computer-Drawn Polyhedra, IEEE Transactions on Computers, Vol. C-19, No. 3, March 1970, pp. 205-213.

Sutherland, I. E., and R. F. Sproul and R. A. Schumacker, A Characterization of Ten Hidden-Surface Algorithms, Computing Surveys, Vol. 6, No. 1, March 1974, pp. 1-55.