A Characterization of Ten Hidden-Surface Algorithms

IVAN E. SUTHERLAND*, ROBERT F. SPROULL**, AND ROBERT A. SCHUMACKER*

This paper discusses the hidden-surface problem from the point of view of sorting. The various surfaces of an object to be shown in hidden-surface or hidden-line form must be sorted to find out which ones are visible at various places on the screen. Surfaces may be sorted by lateral position in the picture (XY), by depth (Z), or by other criteria. The paper shows that the order of sorting and the types of sorting used form differences among the existing hidden-surface algorithms To reduce the work of sorting, each algorithm capitalizes on some coherence property of the objects represented. "Scan-line coherence," the fact that one TV scan line of output is likely to be nearly the same as the previous TV scan line, is one commonly used kind of coherence. "Frame coherence," the fact that the entire picture does not change very much between successive frames of a motion picture can be very helpful if it is applicable.

By systematically looking for additional kinds of coherence and untried sorting orders and sorting types, the paper is able to suggest two promising new approaches to the hidden-surface problem. The first, a combination of three existing algorithms, is promising because it would capitalize on both frame and scan-line coherence The second new approach would sort in the order Y, Z, X, $\cdots$ the only sorting order for which an existing algorithm could not be found

Key words and phrases hidden-line elimination, hidden-surface elimination, sorting, coherence, computer graphics, raster-scan, perspective transformation, analysis of algorithms

CR Categories: 8 2, 5 31.

I. INTRODUCTION

While it is relatively easy to produce a perspective picture of a transparent object made up only of lines, it is rather more difficult to produce a realistic rendering of an opaque object. The opaque object is more difficult to show because one must decide not only where each part of the object will appear on the picture, but also whether to show any part at all Some parts of an opaque object will be concealed in any view of it; a computer programmed to make pictures of opaque objects must be able to decide which parts are visible in the chosen view and thus must be shown, and which parts are hidden and thus must be omitted.

The task of deciding which parts of an object should be shown and which parts should be omitted was originally known as the "Hidden-Line Problem," because it amounted to finding and eliminating--or making dashed--all of the lines in an output drawing which were hidden by other objects. Now that shaded pictures are being produced by computer, a variant of the problem, the "Hidden-Surface Problem," has become important. In a shaded picture one must include or omit entire surface areas rather than just the lines representing edges. Because the hidden-line and hidden-surface problems are very similar, we have chosen to treat them together in this paper.

* Evans and Sutherland Computer Corporation, Salt Lake City, Utah

** Stanford University, Palo Alto, California (formerly with Evans and Sutherland Computer Corporation).

CONTENTS

Shaded pictures are produced by recording the shade of gray or the color of each point in a two-dimensional array Because many shades of gray or shades of color may appear in such pictures, they are correctly called shaded pictures, but the slightly erroneous term "half-tone pictures" has also been applied Because the large array of points used by a computer to define a shaded picture is often reproduced by scanning it in a raster, much as does a TV set, these pictures are also referred to as "raster-scan" pictures. The raster-scan process contrasts with the random-scan process used by plotters and calligraphic display systems to make line drawings.

The computer programs which produce pictures of opaque objects accept as input a description of the object to be shown, and a desired viewing position and direction for a hypothetical observer. From this basic data the program then computes what such an object would look like to an observer so positioned, a process long known by architects as "rendering." Although it is easy to compute the perspective projection that is usually involved, it is much more difficult to solve the hidden-surface problem. In light of the difficulty of the hidden-surface problem it is remarkable that many people have

This work is sponsored by the Information System Program of the Office of Naval Research under NR 049-333

independently solved it, and natural that their solutions have taken many different forms. It is the purpose of this paper to survey the principal published methods and to provide as background some understanding of the mathematical operations common to all.

The study which led to this paper had a further purpose. It was our plan to compare and categorize the known algorithms in the hope that such a categorization of alternate solutions to a problem might lead to some fundamental insight into the nature of the problem itself. We took this taxonomic approach for reasons of research rather than teaching, discovering only later that a survey paper like this could also be useful.

Two underlying principles have emerged from our study. First, all of the algorithms *sort* or search through collections of surfaces, edges, or objects according to various criteria, finally discovering the one visible item and displaying it. Although the order and kind of sorting used differ, our supposition that sorting is the key to the task seems amply justified

The second underlying principle is *coherence*. The environments rendered by the hidden-surface algorithms consist of objects with more or less flat surfaces and straight edges rather than random discontinuities. This coherence of the environments being rendered limits how different the picture can be from place to place or from time to time. All of the algorithms capitalize on various forms of coherence to reduce to manageable proportions the work of sorting. The kinds of coherence most helpful to particular algorithms are easily identified; the extent to which useable coherence exists in a particular solid object seems to determine the speed with which the algorithm will render it.

II. BACKGROUND

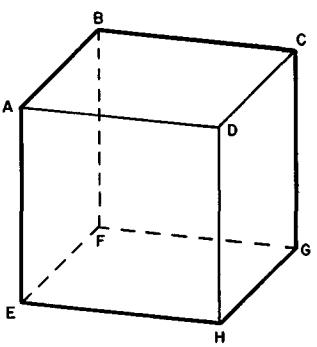
The Environment

Object Descriptions

There are, of course, as many ways to describe three-dimensional objects in a computer memory as there are programmers to assign to the task. The algorithms which we treat in this paper operate on objects which are made up only of flat faces, i.e., plane-faced objects. Convex objects can be described by giving the coefficients of the plane equation of each of their faces, but it is often simpler to use the coordinates of their vertices and the topology of the connections between vertices. Although the data required by the various algorithms may differ from this form, conversion to the required form is straightforward.

Each vertex, then, is described by giving its coordinates in three dimensions in some convenient coordinate system, the "object coordinate system" Each face is described as a polygon (presumed to be planar) by listing its vertices. Such an object description is shown in Figure 1. If the faces have more than three vertices each, the vertex positions must be related if the surfaces are actually to be planar. Each face might also be assigned a color, transparency, reflectance, texture, or other properties.

Because collections of objects are often shown together, it is convenient to build a



POLYGONS
ABCD
DCGH
EHGF
BAEF
ADHE
CBFG

VERTEX	X	Y	Z
A	-1	1	1
B	-1	1	-1
C	1	1	-1
D	1	1	1
E	-1	-1	1
F	-1	-1	-1
G	1	-1	-1
H	1	-1	1

Figure 1· Point-polygon representation of a cube

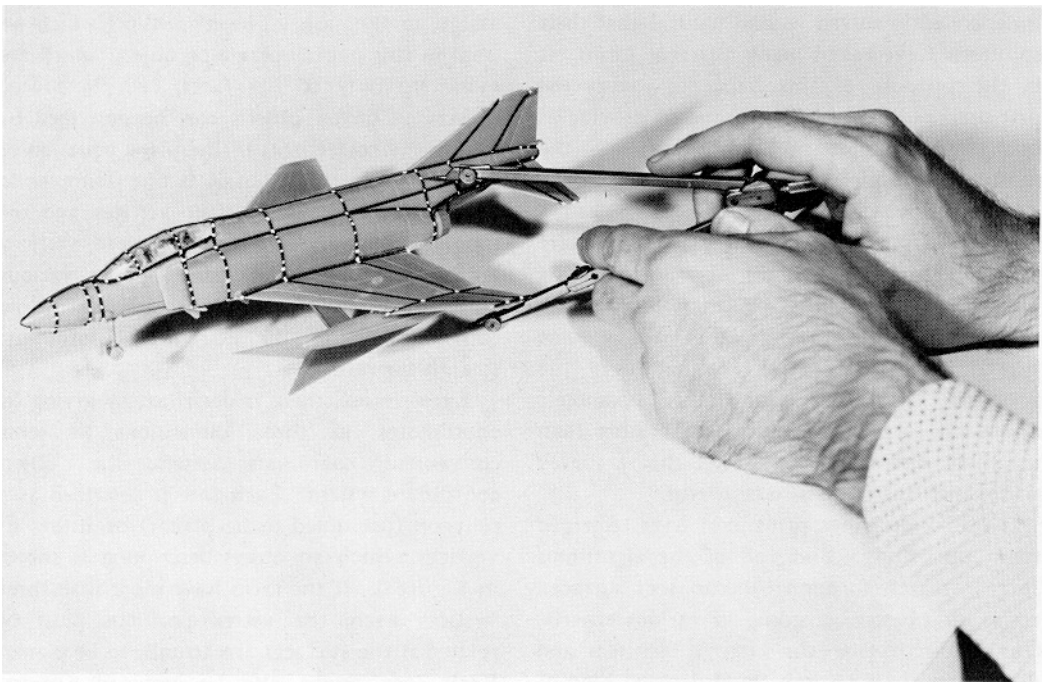


FIGURE 2A

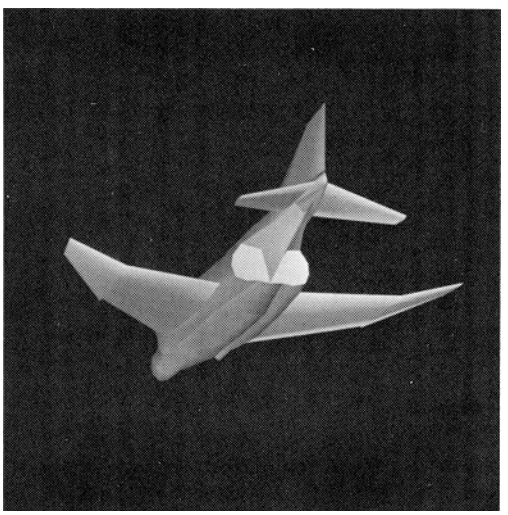


FIGURE 2B

Figure 2. The data for computer-generated pictures may be measured by hand *a*, *b*: A model F-4 and the resulting computer output. *c*, *d*, *e*, *f*: A Volkswagen and three forms of computer output.

structure of object references. A single object definition, for example a ship, might be referenced several times to make a fleet. Each reference would, of course, carry different position, size, orientation and, possibly, color and texture parameters. Because of the obvious applicability of such a presentation to simulating natural scenes, we have chosen to call the totality of objects to be shown an "environment." The environment is nothing more than a description, possibly structured, of all of the surfaces on which the hidden-surface algorithm must operate.

A structured environment requires programs

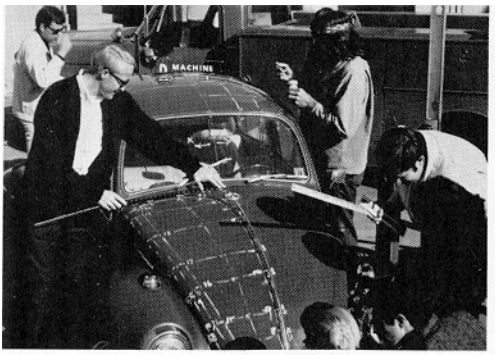


FIGURE 2c

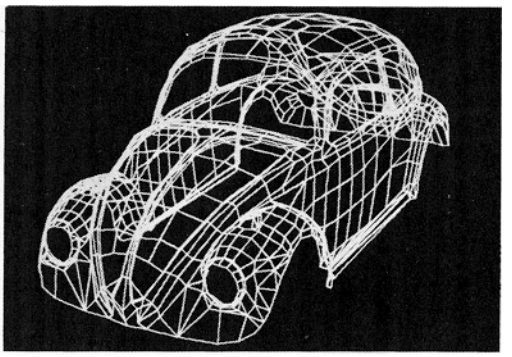


FIGURE 2D

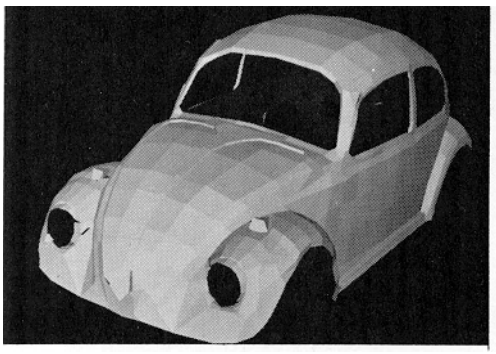


FIGURE 2E

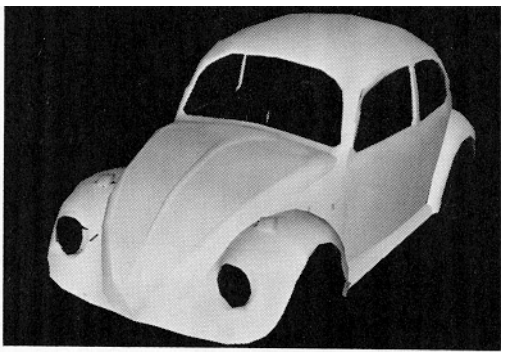


FIGURE 2F

to interpret its structure. The hidden-surface part of the computation may make use of the structure, determining that a given object is entirely hidden by another object, rather than doing the computation individually for the faces of the objects. On the other hand, the hidden-surface part of the computation may ultimately have to know the final location of a particular surface of some object in order to determine whether it hides some other surface. Thus the programs which interpret the environment must be able to compute the location of any vertex, and hence the location of any surface, as it finally appears through all of the structure of the environment. More important, we often choose to treat the environment as if it were made of only a single object, speaking of the "object coordinate system" when a more exact term would be the "environment coordinate system."

The initial generation of environments and object descriptions for use with hidden-surface algorithms can itself be a major task. For the algorithms surveyed here the object must first be approximated by a set of planar faces. Economy insists that the number of such faces be minimized, while quality of representation insists that the approximation remain faithful. Thus the first task is to choose a set of approximating faces, a task which remains an art not unlike the art of representing objects with paint on canvas. One may, of course, avoid this step if the object is already plane-faced or if some natural representation is evident.

After having chosen the set of faces with which to represent the object, one must obtain the coordinates of their vertices. This process can be done by hand, as shown in Figure 2, by digitizing in three dimensions with mechanical measuring equipment (Figure 3); by digitizing from pairs of two-dimensional drawings or photographs (Figure 4); or as a direct result of some computation

Having obtained the coordinates of the vertices one must next connect them together into faces. Omission of a face description will leave a hole in the final result. Inversion of

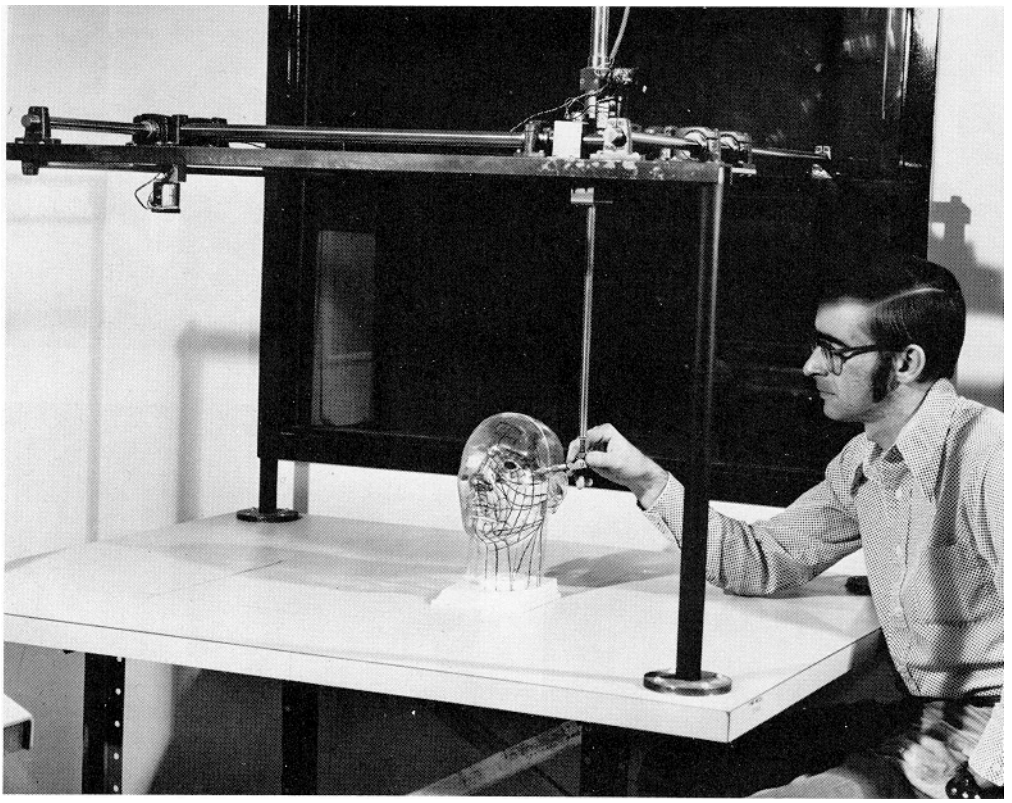


Figure 3 Direct digitization of a three-dimensional object.

the order in which two vertices are referenced, or erroneous reference will result in wild distortion of the face. Such errors, as well as any errors in vertex position, must be laboriously corrected. The process of describing a reasonably complex object, such as the automobile or the aircraft shown in Figure 2, can consume several man-days. The task is not unlike programming.

One should be alert to differences in the topological properties of different environment descriptions Some hidden-surface algorithms need to know which surfaces meet at a particular edge, while others make no use of such information. Similarly, some make use of groupings of faces into objects or clusters while others simply treat faces as if they were disjoint. The difficulty of building environment models for the algorithms increases with the amount of such topological information required by the algorithm, but the algorithm may profit immensely from the quicker reference that such additional information provides.

The algorithms surveyed here capitalize on various properties of the environment. If the environment is stationary, for example, and a series of pictures is being made which simulates an observer moving through it, it is appropriate to invest a large amount of computing in preparation of the environment before starting the hidden-surface computation. One can afford to find, by exhaustive search if necessary, all penetrating surfaces, and, by dividing them at the lines of penetration, eliminate penetrations from any later consideration. If parts of the environment move with respect to each other, or if only a single picture is desired, such computations may not be worthwhile If the environment is known to be made only of convex polygons, or only of polygons smaller than a certain size, or only of a single sheet of connected polygons representing a single valued function $Z =$

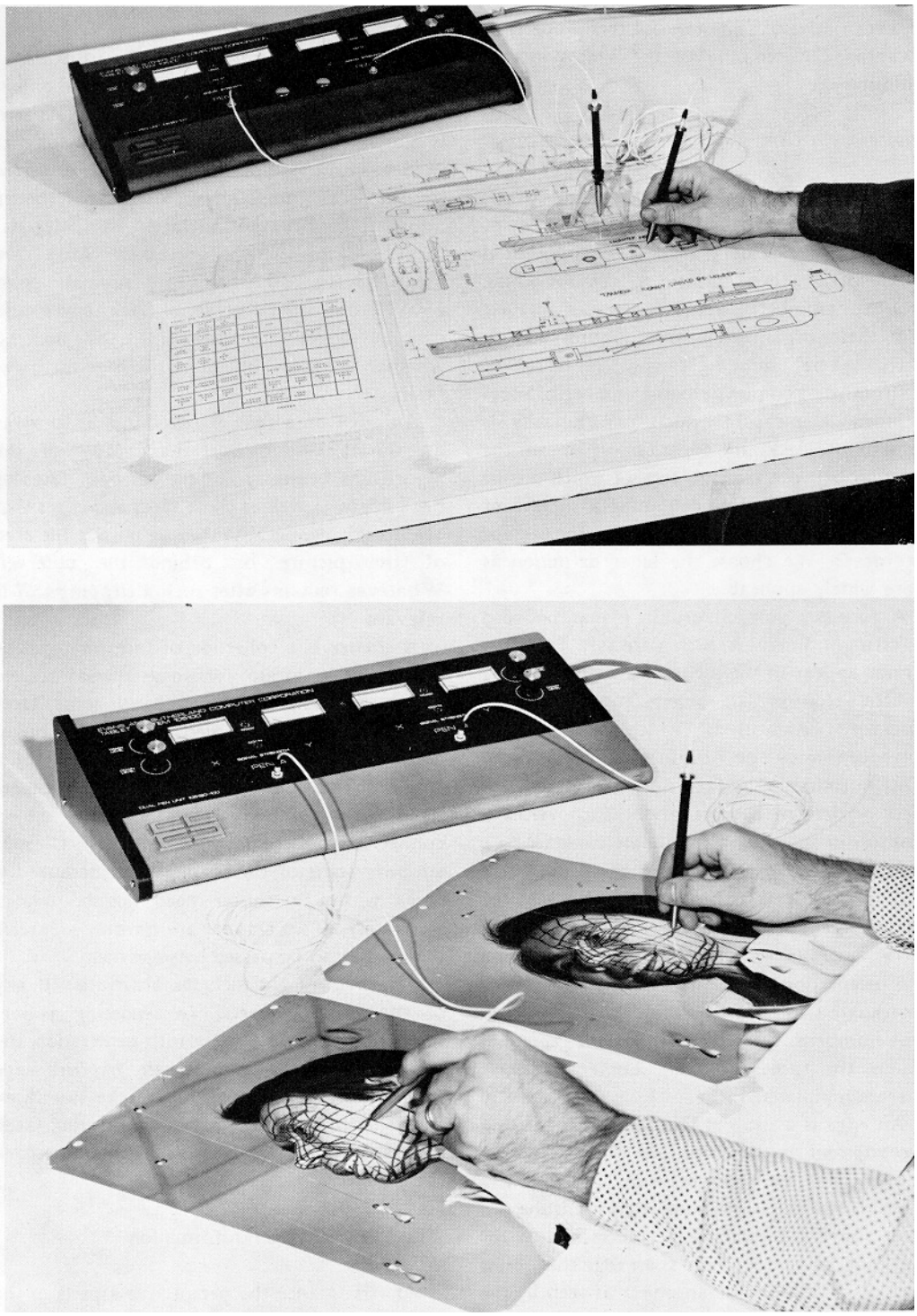


Figure 4 Digitization of an object using two views The views may be (*a*) orthographic, or (*b*) perspective

$F(X,Y)$, or has any other special property known in advance, one can use this property to advantage in simplifying the hidden-surface computation.

Environment Complexity Definitions

In order to speak quantitatively about environments, we have developed a number of statistical measures of environmental complexity. These measures say something about the number of surfaces represented, their size, their organization into groups, and so forth. The most common measure of environment complexity is the number of edges included, a measure which unfortunately is ambiguous. By an edge do we mean the junction between two surfaces, of which a cube has twelve, or the line delimiting a surface, of which a cube has twenty-four, four for each of six faces? We choose the latter definition as more widely applicable.

A *face* is a polygon, usually planar, bounded by straight lines. A *back face* is a face that cannot appear in the picture by virtue of being on the side of an object away from the observer. Algorithms which accept open polyhedra may not distinguish back faces. Most authors define faces in terms of a list of their *vertices* or corners, giving each vertex a position in space by a coordinate triple (X,Y,Z). A face usually carries a *color* or *shading rule* that is used to compute its appearance in the rendering should it be visible. A face may also carry a *plane equation* defining the location and orientation of the plane of the face. If the coordinate triples for the vertices of a face and the numbers describing its plane equation match, the face is *planar*. Some algorithms tolerate nonplanar faces

An *edge* is a straight line segment connecting two adjacent vertices of a face. This definition implies that the joint between adjacent faces contains two edges. In some algorithms the two faces share a common edge representation for the joint. A *contour edge* is an edge that forms part of the outline of an object as seen by the observer. A *back edge* is one that cannot appear in the environment being rendered because it lies on the side of an object away from the observer.

A *surface normal* is an outward-pointing vector, normal to the surface of the object. The surface normal for a face, or *face normal*, is closely related to the plane equation for that face. The surface normal at a vertex, or *vertex normal*, is sometimes used to better approximate curved surfaces by polygonal faces. Faces which are back faces are identified because their face normals point away from the observer. Back edges and contour edges are determined by noticing whether the faces that share the edge are back-facing.

An edge or a face is *relevant* if it survives an initial culling operation. Most of the algorithms begin by culling out back faces or back edges as well as those faces and edges that are not visible because they lie outside the area of the picture or behind the observer. Whatever remains after such a *clipping cull* is relevant.

A *cluster* is a collection of faces that can be treated as a group for some special reason. Often a cluster consists of all those faces belonging to a single object, but a cluster might consist of several objects or only a part of a single object. One might also define a cluster based on limited lateral extent, object connectivity, or some other property. Clusters simplify some of the sorting tasks because the faces within a cluster need not be treated separately. Two clusters are *linearly separable* if a plane can be passed between them.

An environment includes *penetration* if any of the faces intersect. In rendering a line-drawing image of a scene with penetration, the algorithm must compute an *implied edge* representing the intersection of the two faces. In a shaded rendering of two penetrating faces, a discontinuity of shade will appear to mark the penetration.

The Perspective Transformation

At first glance the perspective aspects of the hidden-surface problem seem very difficult: we must consider many "rays" leaving the

observer's eye at various angles, and compute which faces the rays intersect. Such ray computations might easily depend on complicated trigonometric relationships. It would be much easier to do a hidden-surface computation for an orthographic projection instead of a perspective projection, for in an orthographic projection all of the viewing rays are parallel, and if we choose to place the Z axis in the viewing direction, then the X and Y coordinates can be those of the screen and the Z coordinate can be that of depth.

Remarkably enough, there is a perspective projection which transforms a three-dimensional object as viewed in perspective into another three-dimensional object which looks the same when viewed orthographically, as can be seen in Figure 5. This transformation, in effect, moves a local observer off to infinity, and distorts the objects appropriately so that they still look the same to him. The transformation preserves the flatness of planes, the straightness of lines, and the ordering of objects in depth, so it is always possible to apply the three-dimensional perspective transformation before doing the hidden-surface computation, and thus do the hidden-surface work with parallel projection.

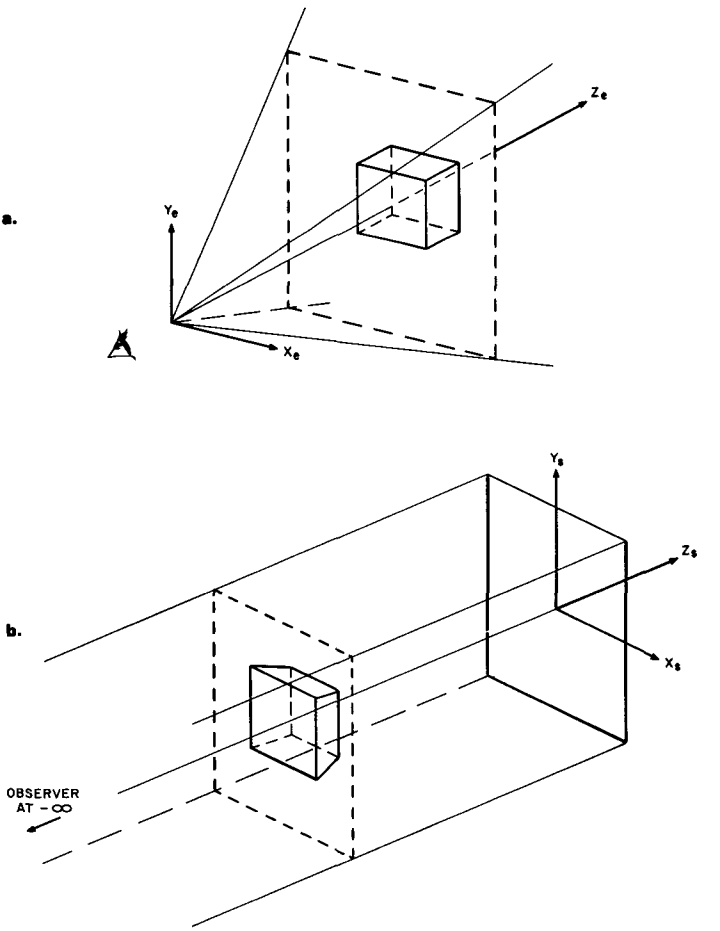


Figure 5 The perspective transformation *a* The eye coordinate system, showing a cube. *b* The screen coordinate system, showing the same cube An orthographic projection of the screen coordinate system onto the display screen produces a correct perspective image of the cube (notice that the more distant face of the cube is smaller in the screen coordinate system)

The ability to use parallel projection in the hidden-surface computation greatly simplifies many computations which would otherwise be quite difficult. Because it is such a big help, we summarize the three-dimensional perspective projection again here, although it has been known to mathematicians for at least one hundred years and has been reported several times in the recent computer literature [14, 15]. The essential idea, again, is to project X and Y into their final positions on the screen, and to adjust the values of Z so that the flatness, straightness, and depth ordering of the objects are preserved.

The complete transformation from the coordinate system of the environment, the "object coordinate system," to the coordinate system in which hidden-surface computations are done, the "screen coordinate system," takes place in two separate transformations. The first coordinate transformation expresses the location of objects relative to the observer's eye, in the "eye coordinate system," accounting for the observer's position and direction of view. The second coordinate transformation expresses the location of objects relative to the screen accounting for the effects of perspective projection. Because both transformations are in matrix form they can be applied simultaneously by using the product of their matrices.

The first transformation places information in the "eye coordinate system." We think of the eye coordinate system as a coordinate system with X to the right and Y up as the observer sees them, and Z parallel to the line of sight forward from the observer.* Multiple applications of such a transformation can be used to position various objects in different positions in the environment should one wish a structured environment.

This first transformation can be expressed easily in matrix form as:

$$(X_e\ Y_e\ Z_e\ 1) = (X_0\ Y_0\ Z_0\ 1) \begin{bmatrix} r & r & r & 0 \\ r & r & r & 0 \\ r & r & r & 0 \\ t & t & t & 1 \end{bmatrix} \quad (1)$$

where all three-component vectors have had a unity fourth component appended to them. Notice that this formulation is identical mathematically to the more familiar rotation and translation formulation:

$$(X_e\ Y_e\ Z_e) = (X_0\ Y_0\ Z_0)\ R + T \quad (2)$$

The unity fourth term in the first formulation picks out the translation terms from the bottom row of the matrix. The formulation of (1) is somewhat simpler to think about because it expresses a complete three-dimensional motion, both rotation and translation (and scaling or certain skews) as a single matrix.

The second coordinate transformation process converts the eye coordinates into the screen coordinate system. This is the "perspective transformation" The essential features of this transformation are:

1. The X and Y coordinates of the perspective view are obtained by dividing the X_e or Y_e eye coordinates by the distance from the observer forward to the object, Z_e.
2. We must compute the "perspective depth" for each point which preserves the straightness of lines, the flatness of planes and the depth ordering. This perspective depth will be used later by the hidden-surface algorithm to decide which objects are hidden by others.

The perspective transformation expressed in the coordinate system of the observer's eye is:

$$(x_s\ y_s\ z_s\ w_s) = (X_e\ Y_e\ Z_e\ 1) \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1/f \\ 0 & 0 & -1 & 0 \end{bmatrix} \quad (3)$$

$$X_s = x_s/w_s \quad Y_s = y_s/w_s \quad Z_s = z_s/w_s$$

where f is related to the "focal length" of an

* The left-handed nature of this coordinate system comes about from our desire to make Z a direct measure of the range to an object and to keep X and Y in their most familiar positions. Because our decision to put the positive Z axis in front of the observer is at variance with the coordinate systems used by some of the authors, the reader must be careful in comparing our use of "minimum Z" (meaning closest to the observer) with that of authors with other coordinate systems.

imaginary optical system that might be used to generate the view.

The transformation can also be expressed in terms of a coordinate system based on the center of the screen rather than the observer's eye, a formulation that reduces to an identity transformation as the observer's eye is moved further and further from the screen, i.e., as the projection contains less and less "perspective," ultimately reducing to an orthographic projection.

$$(x_s\ y_s\ z_s\ w_s) = (X_e\ Y_e\ Z_e\ 1) \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1/d \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (4)$$

$$X_s = x_s/w_s \quad Y_s = y_s/w_s \quad Z_s = z_s/w_s$$

where d is the distance from the observer's eye to the screen.

By representing three-dimensional coordinates with four-component vectors we are free to scale arbitrarily the four components used. This notation is called "homogeneous coordinates" and is explained further in [14] and [20] The divisions which subsequently convert from the homogeneous coordinates to real coordinates, of course, remove any arbitrary scale factor that may have been used.

An orthographic projection of the screen coordinate system onto a display screen will now produce an image (see Figure 5). In addition, the "perspective depth" Z_s computed by the transformation can be used by a hidden-surface algorithm to decide which faces lie in front of others, and hence to compute a rendering with hidden surfaces removed.

The values of X_s and Y_s are directly related to the coordinates that must be given to display hardware to display the point originally represented in the object coordinate system as $(X_0\ Y_0\ Z_0)$. By convention, we shall assume that values of X_s and Y_s lying between -1 and +1 are to be mapped onto the display screen: -1 corresponds to the left (bottom) edge, and +1 to the right (top) edge. If, for example, our display required coordinates in the range 0 to 1023 for both x and y values, we compute $511.5X_s+511.5$ and $511.5Y_s+511.5$ as the coordinates to give the display hardware.

However, the perspective transformation does not *guarantee* that the values of X_s and Y_s will indeed lie within the ranges that map onto the screen surface. The two-step transformation process transforms all objects, whether they will lie off the screen or behind the observer. It is essential that objects be "clipped" so those that lie off the screen or behind the observer are eliminated from further consideration. This clipping process may be performed after the perspective transformation, but must be performed before the division, because the division destroys some essential information. The effect of the clipping operation is to insure that $-w_s \leq x_s \leq +w_s$ and $-w_s \leq y_s \leq +w_s$, and thus that the values of X_s and Y_s will lie within acceptable ranges.

The clipping operation involves simple computations on each face; its difficulty grows linearly with the number of faces. A rather simple formulation of the clipping process for planar and nonplanar faces may be found in [20].

An essential feature of the perspective transformation is that lines and planes in the object coordinate system must transform into lines and planes in the screen coordinate system. Thus, a line in screen coordinates can be generated by interpolating linearly between the endpoints in screen coordinates. The hidden-surface algorithms make extensive use of this property when comparing the depths of various edges or faces in the environment.

In summary, the screen coordinate system that we have established preserves the depth relationships of objects as seen by the observer in the object coordinate system. Furthermore, the X_s and Y_s coordinate values already reflect the perspective effect. Figure 6 illustrates the convenience of the screen coordinate system: The effective location of the observer is at $Z_s=-\infty$, thus making all rays from the eye parallel to the Z_s axis.

Geometric Computations

There are a number of geometric computations that appear in many of the

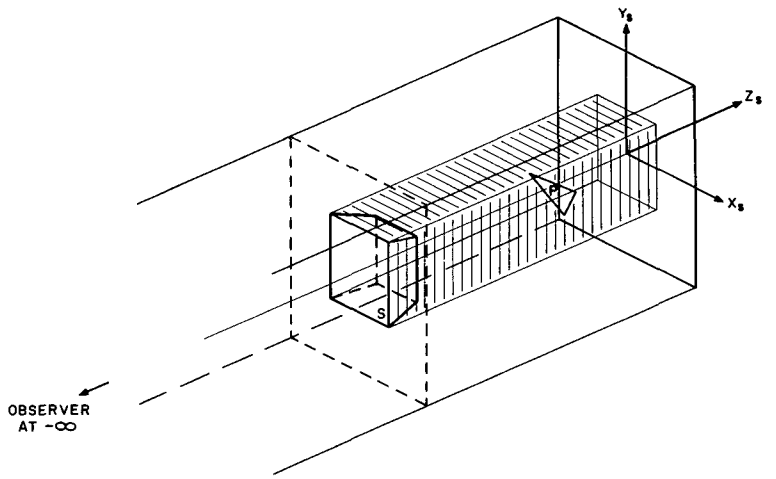


Figure 6 The hidden-surface computation in the screen coordinate system. Face *S* defines a "shadow box" (shown shaded) The box has the same cross-section as the face *S*, and extends behind *S*, away from the observer The face *P*, wholly contained inside the shadow box, is clearly hidden by face *S*

hidden-surface algorithms. The purpose of these computations is to establish relationships among polygons or edges. For example, a particular test polygon may be compared to others to decide which, if any, obscure it. Obscuring involves not only lateral computations to discover if pairs of polygons overlap on the screen, but also depth computations to discover if part or all of one polygon lies further from the observer than another. Both the lateral and depth computations can conveniently be performed in screen coordinates because the angular questions which might otherwise complicate the process have been eliminated by doing the perspective projection first

Generally the authors of hidden-surface papers fail to tell how these computations are done, leaving the reader to invent his own methods. In this section we outline several of these methods in the hope of arming our readers with a "bag of tricks" with which to attack their own programming tasks. This brief collection by no means includes all the calculations performed by the hidden-surface algorithms. More information can be found in the section that describes the algorithms, in the original papers, or in [14].

Minimax Tests

If two polygons do not overlap in X_s or Y_s, then neither can possibly obscure the other (see Figure 7a). If a fast method is available for detecting no overlap, a great many faces can quickly be proven irrelevant to the visibility of a given test face. Minimax tests provide such a quick rejection test. If the maximum X_s coordinate of a face is less than the minimum X_s coordinate of another face, the two cannot possibly overlap in X_s. A similar argument can be applied in the Y_s direction. Because these tests are equivalent to comparing the minimum bounding rectangular boxes for the two surfaces, they are sometimes called minimum box tests. There are, of course, cases which minimax tests cannot reject in which the faces nevertheless do not overlap (see Figure 7b). A minimax test in Z_s can often select the foremost of two surfaces known to overlap in X_s and Y_s.

Surrounding Polygons

A technique that can establish the relationship between a face and a point is the surrounder test (see Figure 8). We must detect

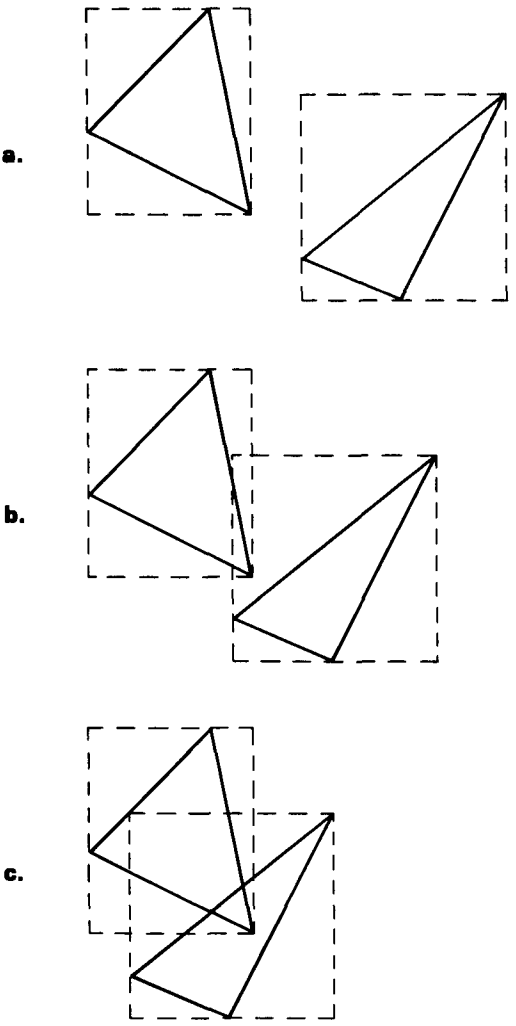


Figure 7: Minimax tests for polygon overlap *a*. Minimax boxes do not overlap, indicating that the polygons do not overlap *b* Polygons do not overlap even though the minimax boxes do *c* Polygons and boxes overlap

whether the edges of face F surround the point B in X_s-Y_s in order to determine whether F might obscure B. If the polygon does surround the point, a depth comparison of the point and the polygon will tell whether the point is visible.

There are three methods for computing surroundedness. If the polygon is known to be convex, one can substitute the point location in X_s-Y_s into the two-dimensional line equations for each of the edges, and if the signs of all such substitutions are the same, the point is "inside" every edge, and is thus surrounded. This test requires that the signs of the coefficients of the line equations be chosen correctly. If the polygon is not convex, two other methods may be used. The first draws any line from the point to infinity, and counts the number of times the line crosses the polygon boundary. If the crossing count is even, the point is outside the polygon; if odd, the point is inside To implement this method, one computes whether each edge of the polygon crosses the semi-infinite test line. The intersection computation is not difficult, but if a polygon vertex lies exactly on the semi-infinite test line, care is required to get consistent results. The second method for nonconvex polygons computes the sum of the angles subtended by each of the edges as seen in two-dimensional projection from the test point. The sum of these angles is always either 0 or a multiple of 2π. If the sum is 0, the point is outside. If the sum is 2π, the point is inside; if the sum is 4π or more, the polygon overlaps itself more than once. If the sum is -2π the polygon goes around the other way. Notice that the addition implied need only be done to 2 bits precision, and so this angle computation need not involve any complicated trigonometry [18]

Uses of Plane Equations

Figure 9 illustrates a case in which neither minimax nor surrounder tests can determine that face B obscures a part of face A. However, if we know the equation of the plane of A, for example:

$$a X_s + b Y_s + c Z_s + d = 0 \qquad (5)$$

we can calculate that B lies on the same side of the plane A as does the observer, and hence must obscure A. If the point (X_s,Y_s,Z_s) is not on the plane A, then the sign of the expression $aX_s+bY_s+cZ_s+d$ will be positive if the point lies on one side of A, and negative if it lies on the other side. When we compute the plane equation coefficients a, b, c, d, we arrange by

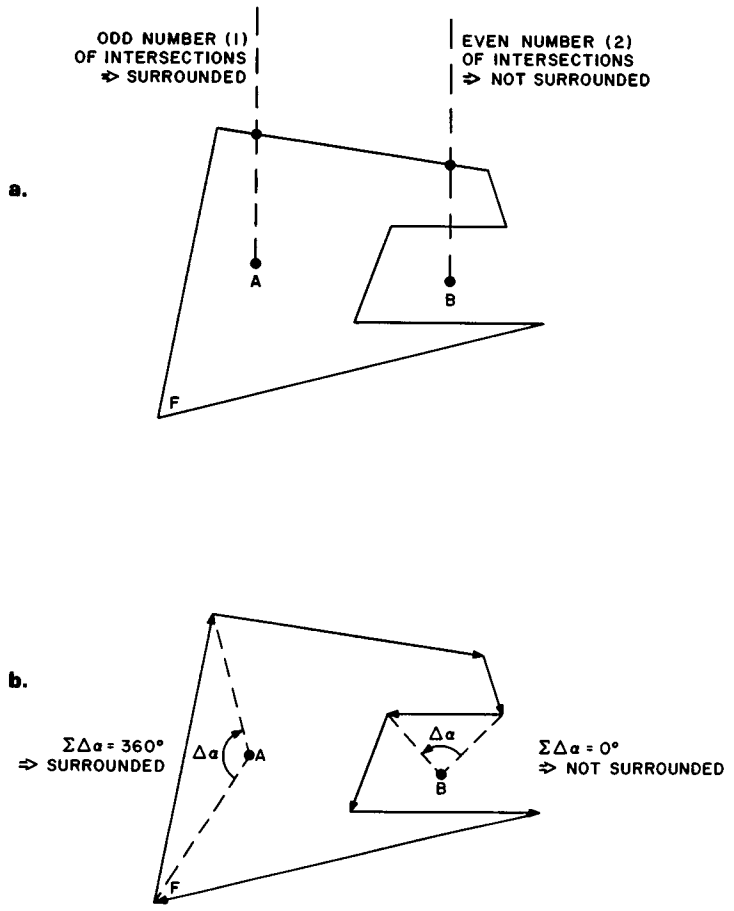


Figure 8 Surrounder tests Point A is surrounded by polygon F, point B is not a The number of intersections of the polygon and a semi-infinite line from the point is counted b The sum of the angles subtended by directed edges of the polygon determines the surrounder condition

convention that a point outside the plane (i.e. outside the object of which the plane is a face) gives a positive value when substituted into the equation of that plane.

The plane equations have other important uses in the hidden-surface algorithms. For example, the depth Z_s of a face can be calculated at a given point (X_s, Y_s). This computation is used to compare the depths of two faces, and hence to decide which one is closer to the observer. In addition, the vector of plane equation coefficients [a, b, c, d] is an expression for a homogeneous vector normal to the plane and by convention pointing outward from it. This normal can be used to identify back faces because the dot product of the normal and a vector in the viewing direction [0 0 1 0] is positive. Similarly, contour edges are identified because they separate two faces, only one of which is a back face. Face normals are also used to compute shading parameters [8, 22].

Computing the Plane Equation

Because the first three plane equation coefficients a, b and c represent a vector normal to the plane, one can find them by knowing such a normal. The fourth coefficient, d, is found by knowing a point on the plane. If three points on a polygon are known not to be collinear, then a, b and c can be computed by taking the crossproduct of the two edges between such points Because such a

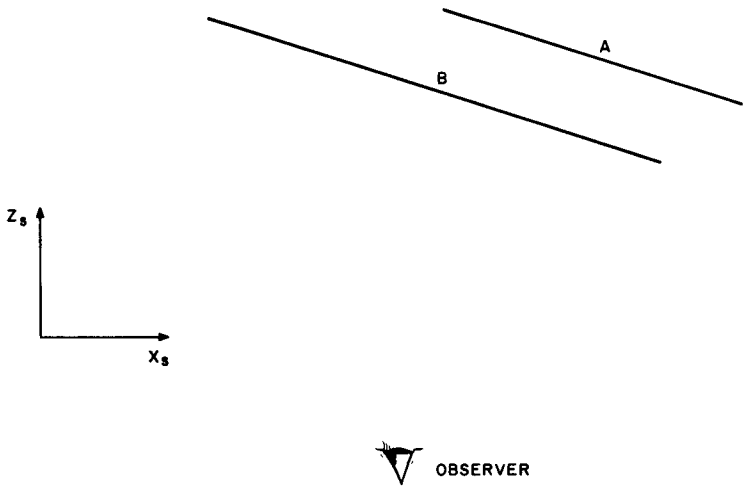


Figure 9 Use of the plane equation to compute visibility Face *B* lies closer to the observer than does face *A* This is determined by noticing that all vertices of face *B* lie on the observer side of plane *A* The plane equation for *A* is used in this calculation

computation requires detection of special cases, we prefer a method suggested by Martin Newell. For vertices $V_i = (X_i\ Y_i\ Z_i)$, the components a, b and c are found as:

$$j = (if\ i=n\ then\ 1\ else\ i+1)$$

$$a = \sum (Y_i - Y_j)(Z_i + Z_j)$$

$$b = \sum (Z_i - Z_j)(X_i + X_j)$$

$$c = \sum (X_i - X_j)(Y_i + Y_j) \qquad (6)$$

This method requires only one multiplication per coefficient per edge. If the polygon is not planar, this method will produce a plane equation related closely to the polygon, but not the best-fit plane equation.

If the equation of a plane is known in object coordinates, it can be transformed into screen coordinates by transformations very similar to those used to transform points. The homogeneous coordinate notation is very convenient in this regard because both points and planes are represented with four components.

Edge Intersections

The algorithms that compute *hidden-line* renderings are more concerned with edge intersections than with face relationships. Figure 10 illustrates a typical case· a face is defined by four directed edges (the direction is by convention clockwise when viewed from outside an object), we are testing the edge *AB* to see whether it is hidden by this object. Since *AB* and *CD* intersect at *I*, we can have one of three cases: *AB* may be nearer the observer than the plane including *CD* (Figure 10a), and hence be completely visible; *I* can mark the disappearance of the edge from *A* to *B* (Figure 10b); or *I* can mark the appearance of the edge (Figure 10c) The first case is identified by calculating the depth of the face containing *CD*, by substituting the X_s, Y_s coordinates of *I* into the plane equation and comparing the result to the depth at *I* of the line from *A* to *B*. If the depth of the line is less than the depth of the face, the line cannot be hidden by the face. Otherwise, if the directed edge *CD* subtends a positive clockwise angle about *A*, the edge appears, otherwise it disappears. Appel [1] calls this the "vorticity" of the edge *CD* with respect to the point *A*.

Segment Comparisons

The algorithms that generate renderings for raster-scan displays such as television monitors often use a class of techniques called "segment comparisons" to solve the hidden-surface

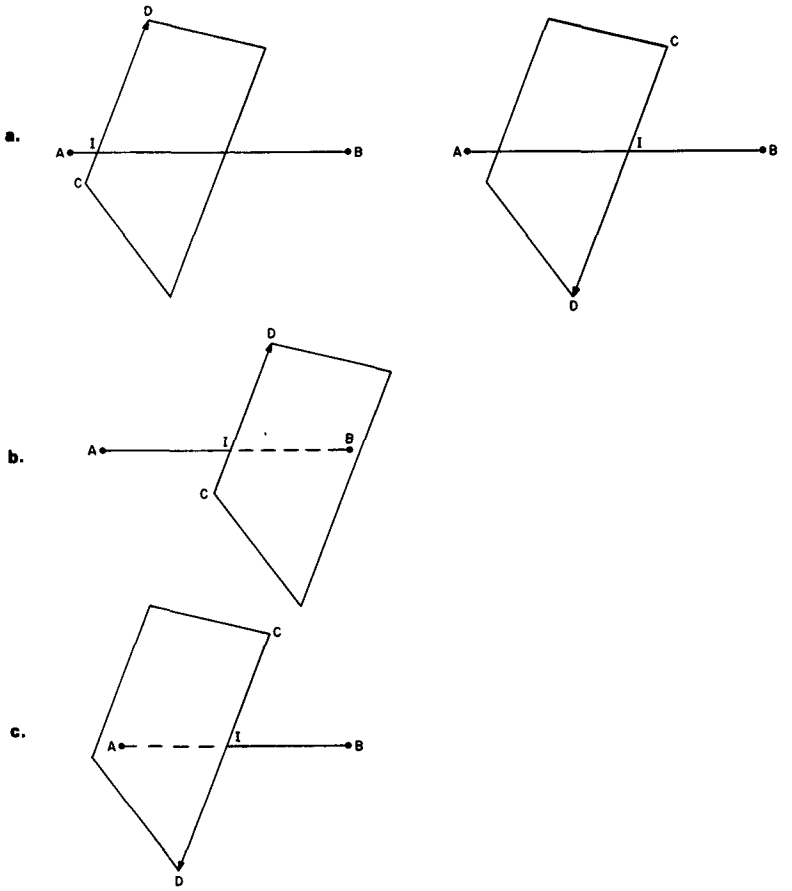


Figure 10 Computing the visibility of a line. The line *AB* intersects the edge *CD* at the point *I*. *a* Edge *CD* is farther from the eye than *AB*. *b*. *CD* marks the disappearance of *AB*. *c*. *CD* marks the appearance of *AB*.

problem. The raster displays scan from top to bottom, left to right, in a fixed pattern. The algorithms are designed to generate the computed image line by line so that it can be displayed in the same order the results are generated Computing the correct image for one scan line is considerably simpler than considering the whole image at once: the plane of the scan line defines "segments" where it intersects faces in the environment (see Figure 11). The hidden-surface problem then becomes a problem of deciding which segments are visible in which parts of the scan line.

The segment comparisons are all performed in the X_s-Z_s plane (see Figure 12). The dotted lines divide the scan line into "spans." Within each span, segment visibility can be determined by comparing the depths of the segments that lie within the span. Depths of segments are computed from the plane equation. The segment with minimum depth is visible throughout the span.

The strategy used to select spans is one of the distinctive features of the various algorithms. The method we have shown in Figure 12a uses each segment endpoint to start a new span. Figure 12b shows a better division of the scan line into spans.

Sorting

Many of the hidden-surface algorithms that we discuss make extensive use of various forms of sorting operations. This section mentions

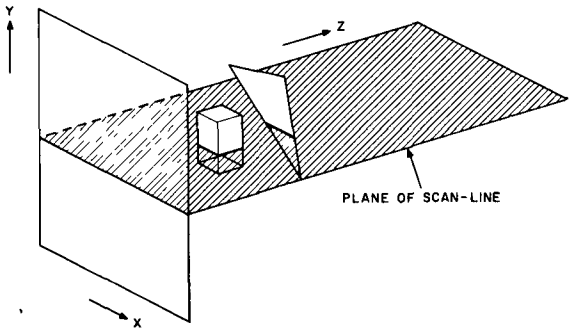


Figure 11 Segments are determined by the intersection of faces and scan lines. The depth relationships among segments in X_s-Z_s plane are used to compute visibility

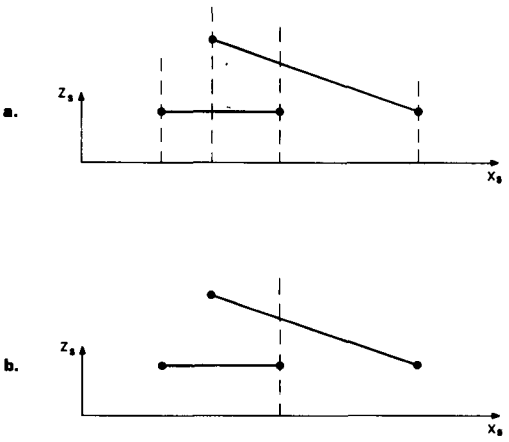


Figure 12 Comparison of segments in the X_s-Z_s plane. *a.* Dotted lines divide the scan line into "spans." Within each span, the closest segment is visible *b.* A span-selection strategy that examines fewer spans to solve the hidden-surface problem for the scan line

briefly the sorting techniques that are relevant to the further discussion; for a complete description and analysis of sorting and searching methods, the reader is referred to Knuth's excellent book [9].

When used in hidden-surface algorithms, the sorting and searching techniques operate on records whose "keys" are often geometric quantities. For example, we might want to sort all polygons; the key for each polygon is the minimum value of the Z_s coordinate of its vertices. Or we may wish to sort edges, using as key the minimum value of the Y_s coordinates of the two endpoints.

Sorting is an operation that orders a set of records according to a selected key. The time required to perform the sort depends on the number of records to be processed (N), the algorithm used to perform the sort, and various statistical properties of the initial ordering of the records. We shall be concerned primarily with two initial orderings: random, and nearly in sort. The following table summarizes several sorting algorithms:

Name	*Knuth page*	*Time (random)*	*Time (nearly in sort)*
Bubble sort	106	N^2	N
Shell sort	84	$N^{3/2}$	N
Quick sort	114	$N \log N$	N^2
Tree sort	422,451	$N \log N$	N
Radix sort	170	N	N

One variation of the radix sort, which we shall call a *bucket sort*, chooses the radix of the sort to be the range of all possible keys. Thus, if we are sorting elements on a 10-bit key field, we may simply allocate 2^{10} buckets: an element with key i is placed into bucket i. In order to fetch an element from the sorted output of a bucket sort, we must scan the buckets until a nonempty bucket is found. Since this search operation can be quite expensive if most of the buckets are empty, one should include the cost of a "priority search" operation following a bucket sort.

The precise properties of sorting techniques may be of tremendous importance. Some require more storage space than others; some lend themselves to fast hardware implementations more easily than others. Although these properties impact the performance of the hidden-surface algorithms, we shall not consider them here.

A *search* operation is used to identify exactly one element in a set of records. For example, we might consider the set of all polygons and wish to search for the polygon whose furthest vertex is nearest the viewpoint. If we are searching for a record with a minimum or maximum value of some key, and if the records are already sorted on that key, then the search may be extremely fast. If, on the other hand, we are searching for a record with a particular key (or a particular property), and the set of records is unordered on that key, we must examine all records. In general, the cost of searching algorithms depends on the structure of the set of records being searched and on the nature of the key:

Name	*Structure of records*	*Knuth page*	*Time*
Sequential	unordered	393	N
Sequential	ordered table	393	$N/2$
Binary search	ordered table	406	$\log N$
Binary tree	ordered tree	22	$\log N$

In some cases, a set of records that is ordered on a key related to the key we are searching for can cut the searching time. Suppose polygons are ordered by depth of the vertex closest to the observer. Then the search for the polygon whose furthest vertex is nearest the observer need consider only the first few polygons--as soon as we encounter a polygon whose closest vertex is farther from the observer than the farthest vertex already located, the search may be terminated.

A *cull* is a particular form of search: we wish to extract from a set of records all those that have a given property, e.g., extract from a list of edges those edges that intersect a given edge on the screen. If the records are sorted in some way that is relevant to the property, we may be able to avoid examining all of the records in the set.

A *merge* adds a new record to some existing set of records and preserves any (sorted) structure of the original set of records. Thus, merging 23 into the ordered list (1 34 56) should yield a list (1 23 34 56). Merging is often a component of sorting algorithms. The performance of merging steps is related to the structure of the records and to the number of records already in that structure:

Name	*Structure*	*Knuth page*	*Time*
List merge	List	159	$N/2$
Tree insertion	Binary tree	422	$\log N$

Coherence

Throughout this paper we use the term coherence to describe the extent to which the environment or the picture of it is locally constant. Just as laser light exhibits a characteristic coherence length, a distance which the light must travel before it is no longer possible to predict its phase accurately, so environments and pictures exhibit a coherence distance, a distance over which one must travel before he can no longer predict with accuracy what he will find. An environment is coherent not only because it consists of flat faces but also because those faces relate to each other to form objects.

The coherence of a set of data can vastly increase the speed with which it can be sorted. If an initially sorted deck of cards is lightly shuffled, for example, the coherence remaining in the deck can be of great use in resorting it. Similarly the relatively slow changes that take place in the appearance of a picture from one place to the next can be of great help in reducing the number of sorting operations that must be applied.

We will later distinguish several types of coherence to see what properties of the objects being rendered enable the rendering algorithms to save work. For example, area coherence describes the fact that many pictures have areas in which the shade does not change very much. Frame-to-frame coherence describes the fact that in a sequence of movie frames the successive frames are likely to be closely related.

III. TAXONOMY OF THE ALGORITHMS

Taxonomy: orderly classification of plants and animals according to their presumed natural relationships.

The objective of the research from which this paper grew was to categorize, compare, and

contrast the methods used by ten authors for solving the hidden-surface problem and thus to learn something fundamental about the problem. Such a categorization, or taxonomy, is easily expressed as a tree whose nodes represent different classes of algorithms and whose various branches represent distinctions between those classes. The particular categorization we have chosen is shown in Figure 13.

The root node of the tree divides the algorithms into three classes: those that compute a solution to the hidden-surface problem in "object-space"; those that perform calculations in "image-space"; and those that work partly in each, the "list-priority" algorithms. By calculations in object space, we mean that computations are performed to arbitrary precision, usually the precision available in the computer executing the algorithm. The aim of the solution is to compute "exactly" what the image should be; it will be correct even if enlarged many times. The image-space solutions are performed with less resolution, usually the resolution of the display screen that will ultimately present an image of the solution. The goal of these algorithms is simply to calculate an intensity for each of the 250,000 or 1,000,000 resolvable dots on the display screen.

In other words, the object-space algorithms ask whether each potentially-visible item in the environment is visible; the image-space algorithms ask what is visible within a raster dot on the screen This difference in attitude produces a corresponding difference in performance: the cost of the object-space algorithms grows as a function of the complexity of the environment, but the cost of the image-space algorithms is limited because the number of screen dots remains constant, independent of the environment complexity.

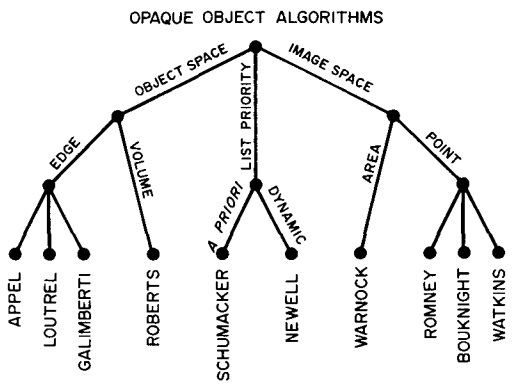


Figure 13. The ten algorithms arranged in a tree.

Coincidentally, the root node of the tree also divides the algorithms into those that are *hidden-line* algorithms and those that are *hidden-surface* algorithms. This distinction is incidental to the object-space/image-space distinction; one can imagine an object-space hidden-surface algorithm, and we know of at least two image-space hidden-line algorithms, both derivatives of Warnock [22].

Object-Space Algorithms

Among the object-space algorithms, we can identify a further division. Although all of these algorithms test relevant edges to determine what parts of the edges are visible, the invisibility criteria are quite different. In the Roberts algorithm, an edge may be obscured by the volume of an object that lies between the edge and the viewpoint. The algorithm thus capitalizes on the spatial coherence of objects: it tests edges against object volumes.

The algorithms of Appel, Loutrel, and Galimberti and Montanari, however, test edges against edges. They observe that the visibility of an edge is coherent, particularly at the vertices that terminate the edge Thus, if the visibility of one edge is calculated, it can be used to save calculations on other edges that share a vertex with the first edge. In this way, most of the visibility decisions become incremental calculations.

L. G. Roberts (1963) [15]

Roberts devised the first known solution to the hidden-line problem. His algorithm tests each relevant edge to see if it is obstructed by the volume occupied by some object in the environment This test is implemented by writing a parametric equation for a line from a

point on the edge to the viewpoint:

$$P = (1-\alpha)E_1 + \alpha E_2 + \beta[0\ \ 0\ -1\ \ 0]$$
$$0 < \alpha < 1\ ;\ 0 < \beta \qquad (7)$$

The first two terms are simply the parametric equation of a point on the edge E_1E_2 in the perspective coordinate system; the third is a vector pointing toward the viewpoint in the perspective space, (0,0,-∞).

This point, P, lies inside a convex object, j, if P is on the "inside" of all planes that comprise the object. This condition corresponds to:

$$P \cdot F_{ij} < 0 \quad \textit{for all } i \qquad (8)$$

where F_{ij} is the plane equation of the ith face of object j. If, for a given object j, values of α and β can be found that satisfy (8), the point on the edge corresponding to α is hidden by the object.

This edge/object test may discover that: 1) the edge is entirely hidden by the object; 2) no portion of the edge is obscured by the object; 3) one part of the edge is not obscured; or 4) two portions of the edge are not obscured. Any unobscured portions are then tested against the remaining objects.

The algorithm uses a variety of techniques to solve (8) for minimum and maximum values of α. Roberts made very effective use of the minimax test, eliminating whole collections of objects from comparison against other objects when their bounding boxes did not overlap. If the simple minimax tests fail, linear programming techniques are required to solve (8).

The algorithm severely restricts the environment: the volume test requires that objects be convex. Although concave objects can be represented by a collection of convex ones, computing a useful decomposition is a difficult task. The computation required by the Roberts algorithm grows roughly as the square of the number of objects in the scene: each edge of a body must be tested against every object in the scene.

Edge-Intersection Algorithms

A. Appel (1967) [1]
P. P. Loutrel (1967) [10, 11]
R. Galimberti and U. Montanari (1969) [5]

The algorithm of Appel exemplifies a class of hidden-line algorithms that compute line drawings. Appel defines the *quantitative invisibility* of a point as the number of relevant faces that lie between the point and the viewpoint. The solution to the hidden-line problem requires computing the quantitative invisibility of every point on each relevant edge.

Appel's algorithm uses "edge coherence" to limit the computing requirement: the quantitative invisibility of a relevant edge can change only where the projection of that edge into the picture plane crosses the projection of some contour edge. At such an intersection, the quantitative invisibility increases or decreases by 1. After all contour edges have been considered, the relevant edge has been divided by the intersections into a number of segments. If the quantitative invisibility of the initial vertex of the edge is known, the visibility of each segment can be calculated by summing the quantitative invisibility changes (see Figure 14)

The quantitative invisibility of the initial vertex is calculated by an exhaustive search of all relevant faces to count how many faces hide the vertex. The hiding condition has two parts: 1) the line of sight to a vertex intersects the plane of the face between the viewpoint and the vertex, and 2) the point of intersection lies "inside" the polygonal face.

The calculations of the initial and incremental invisibilities determine the quantitative invisibility of the final vertex of the edge, which can be used to determine the initial invisibility of other edges emanating from that vertex. This, too, is a form of coherence, for it often saves the exhaustive search to determine the quantitative invisibility of an initial vertex. In general, the exhaustive search will be performed only once for each cluster. For these algorithms we define a "cluster" as a collection of faces and edges that

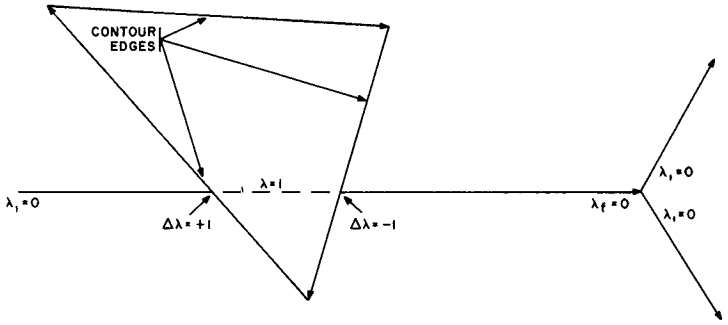


Figure 14 Computation of the quantitative invisibility (λ) of a relevant edge The invisibility of the initial vertex, λ_i, is computed The quantitative invisibility on the edge changes only where images of contour edges intersect the image of the relevant edge and the contour edges are closer to the viewpoint than the relevant edge The invisibility of the final vertex, λ_f, is used as the initial invisibility of other edges emanating from the vertex.

requires only one exhaustive search to compute the quantitative invisibility of a starting point from which invisibility may be promoted along a network of edges.

A correction must be applied to the quantitative invisibility of a vertex to determine the quantitative invisibility of the starting point of an edge beginning at the vertex (see Figure 15). The complication arises because faces that intersect the vertex may hide some relevant edge emanating from the vertex. This "invisibility correction" requires testing the edge against only those faces that intersect the vertex.

The implementation of these ideas varies among the three edge-intersection algorithms we studied. Appel, who was the first to propose this scheme, defined the terms *quantitative invisibility*, *contour edge*, and *material edge* (equivalent to our definition of relevant edge). His method of intersecting relevant and contour edges in object coordinates is noteworthy: a contour edge will change the visibility of a relevant edge if it pierces the triangle formed by the viewpoint and the vertices of the relevant edge as shown in Figure 16. The test which determines that the piercing point lies within the triangle is called *vorticity*. it measures the direction (clockwise or counter-clockwise) of the three directed edges of the triangle relative to the piercing point. If all directions are the same, the piercing point lies within the triangle and hence changes the quantitative invisibility of the relevant edge; if not, the contour edge has no effect on the visibility of the relevant edge.* If an intersection is located, the change in the quantitative invisibility is either +1 or -1; the sign is determined from the sign of the cross product of the contour edge vector and the relevant edge vector. This is a very quick test that depends on all faces being drawn in a consistent direction (e.g., clockwise) as viewed from outside the object.

Loutrel's approach is very similar to that of Appel: his term "order of invisibility" is equivalent to quantitative invisibility; his term "boundary edge" is equivalent to contour edge. Loutrel computes intersections of edges by projecting edges onto the picture plane and computing intersections in a two-dimensional space. If an intersection is found, the depths of the two edges are compared at that spot to decide which edge hides the other.*

The approach of Galimberti and Montanari is similar to Appel's and Loutrel's, but rather than computing the number of faces hiding a point, they compute the *set* of faces hiding a point, which they call the *nature* of the point. The methods of determining the nature of an initial vertex, the invisibility correction, and

* These calculations would be greatly simplified if performed in the perspective coordinate system. None of the three algorithms, however, did so.

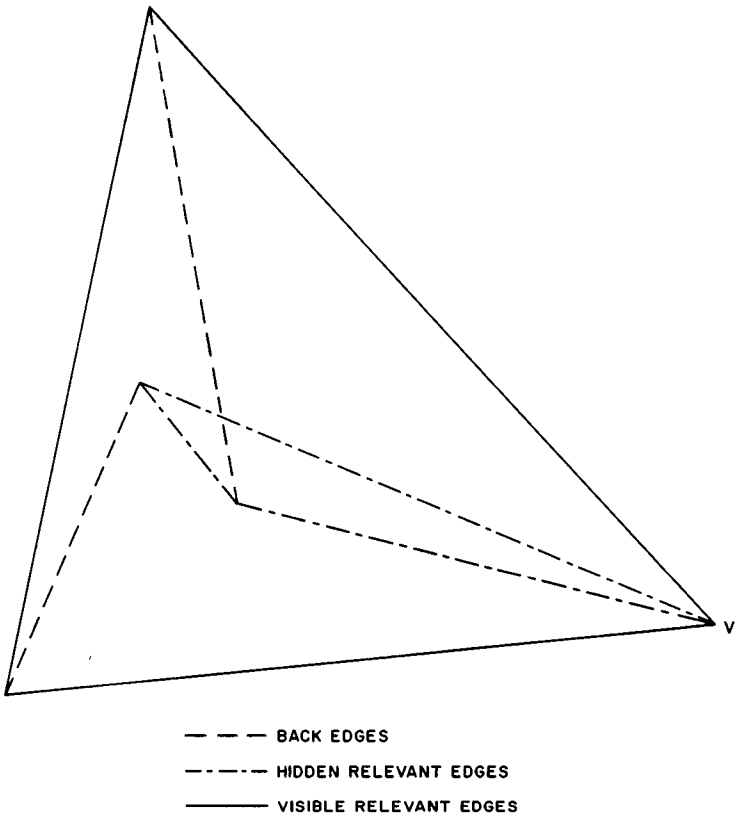


Figure 15 Invisibility correction Two of the four edges emanating from vertex V are hidden by a face of the polyhedron, even though vertex V is not obscured.

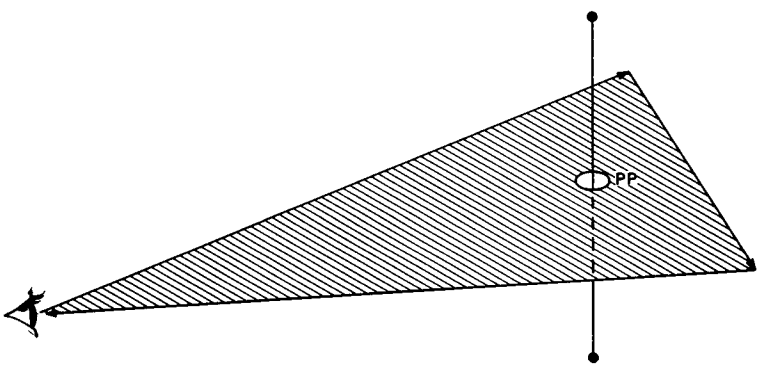


Figure 16 Appel's test for intersection of two edges is performed by testing one edge against the triangle formed by the eye point and the other edge. If the test edge pierces this triangle, as shown, it not only crosses the other edge in the viewing plane, but is known to obscure it.

the incremental invisibility changes are similar to those of Loutrel, with the added complexity of computing a set of obscuring faces. Each edge has an associated set of, at most, two relevant faces that the edge separates. For example, if the nature set at a point of an edge is { ... α ... }, and the edge image crosses another edge whose face set is {α,β}, i.e., the edge

between faces α and β, the nature of the moving point changes to { ... β ... }. Whenever the set is empty, the point is visible.

Each of these three algorithms locates all intersections along a relevant edge, and then, although none of the papers mentions it, must sort the intersections in order of their occurrence along the edge in order to establish the quantitative invisibility of all points on the edge. However, in certain instances, the sort can be avoided by noticing that the quantitative invisibility of the initial vertex is so high that there are not enough intersections to make any portion of the edge visible. In this case, the quantitative invisibility of the final vertex can be computed quickly as the sum of the quantitative invisibility at the initial vertex and all the incremental changes The set notion of Galimberti and Montanari does not seem to be amenable to omitting the sort within an edge.

A number of unpleasant singularities can occur which require careful attention to compute the invisibility correctly (Figure 17). Galimberti and Montanari have encountered these problems and report ad hoc solutions in their paper; the other authors make little mention of them.[**] Since all visibility calculations are incremental, it is important that these cases be handled carefully, for errors will propagate to other portions of the picture.

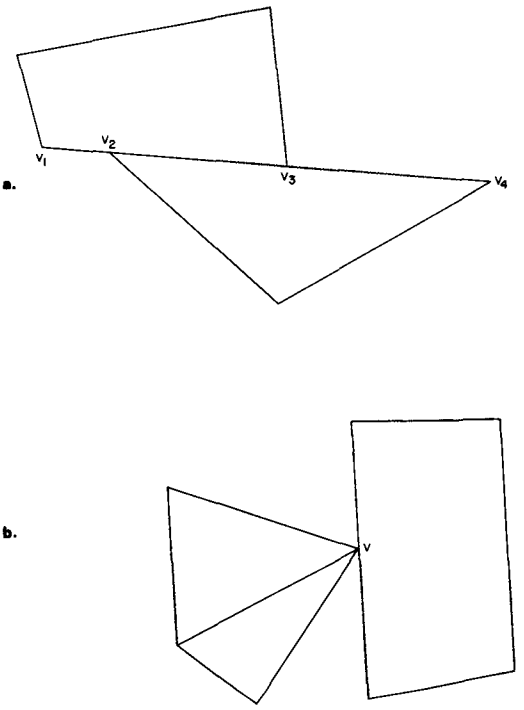


Figure 17. Singularities *a*. The images of edges V_1V_3 and V_2V_4 overlap *b*. The image of vertex V lies on the image of an edge

Image-Space and List-Priority

The image-space and list-priority algorithms are designed to create images for a fixed-resolution display, often a television monitor. Although the specific aims of the various algorithms are not identical, the group has been motivated by desires for real-time speed and for realism in the images. These algorithms are now used to generate quite spectacular shaded pictures in color; they have been used to produce a number of quality movies.

Historically, the efforts in this field are due to two groups. The first work, reported mostly in the writing of Schumacker and his collaborators, was begun at General Electric in 1965 Their goal was to develop a high-quality image-presentation system for use in visual flight simulation. Their work culminated in the delivery and later enhancement of a system for NASA's Manned Spacecraft Center. This system generates images of a spacecraft environment, typically involving another craft and a background landscape, and presents the image to the pilot on a television monitor This was the first real-time solution to the hidden-surface problem and has been operational since 1968.

The other major group to develop image-space algorithms was started in 1967 at the University of Utah by D. C Evans. Evans understood clearly from the start the importance of the limited resolution of image-space, and the need for incremental computation during TV scanning. The Utah efforts produced a series of interesting algorithms, and have resulted in a real-time algorithm by G S. Watkins that is now

** Loutrel commented to us, "They all had to be solved in the program and that was no picnic"

commercially available in hardware from the Evans and Sutherland Computer Corporation.

The most recent algorithm, that of Newell et al of the Computer Aided Design Centre, Cambridge, England, resembles most closely that of Schumacker; these two are therefore classed together in the tree. It appears that the Newell group constructed their algorithm without any knowledge of the details of Schumacker's work. The reasons for classifying these two algorithms together is, as we shall see below, based on a technical similarity rather than any detectable historical influence.

The distinction between image-space and list-priority algorithms concerns the way in which the ultimate visibility of a surface is computed In the image-space algorithms, the visibility test is postponed until last, and comes about as a computation of the depth of the various surfaces that would be penetrated by a viewing ray at a particular point in the image. Thus, these algorithms can capitalize on the lateral separation of the image to reduce the number of depth computations required. The list-priority algorithms of Schumacker and Newell, on the other hand, precompute in object-space a visibility ordering or "priority" for all surfaces before generating the picture in image-space. The priority of a surface can be expressed as a linear-ordering of the surfaces such that if ever two surfaces need be compared for visibility, the one with the higher priority is the visible one.

The list-priority algorithms are placed between the object-space algorithms and the image-space algorithms because they function partly in each space. The algorithms have object-space character because the depth overlap calculations are performed with high precision. Their image-space character comes about only because of the finite resolution of the output medium available. Were an output device available which could paint a sequence of polygons to arbitrary resolution, leaving visible at any place only the last painted polygon, then these algorithms could be considered object-space algorithms. Because such a device is not available, the output step of these algorithms takes on much of the character of the image-space algorithms.

List-Priority Algorithms

The difference between the Schumacker and the Newell algorithms concerns the manner in which the list-priority is computed. The Schumacker algorithm performs most of the priority calculations "off-line," occasionally with human intervention. Schumacker's priority list is primarily a property of the environment and does not depend very much on the location of the viewpoint.

Although the investment in computing the priority list from the environment description is quite high, virtually the same list can be used to generate many, many frames. This approach is particularly convenient for flight simulation, where the environment rarely changes, although the viewpoint changes quite frequently. The Schumacker algorithm takes advantage of the environments usually employed in flight simulation to limit the topology of the environment: only environments with convex faces and linearly separable clusters are allowed.

The Newell algorithm, on the other hand, computes a priority list from the environment description before processing each frame. This approach very conveniently accommodates changing environments. In addition, Newell's priority computation makes no restrictions on the topological complexity of the environment.

Another important difference between the algorithms concerns *clustering*. Schumacker observed that the computation of priority need not compare every face in the environment with every other face to determine the order of faces in the priority list Rather, the environment is divided into clusters. Within a cluster, each face is compared with every other face in the cluster to compute a *face priority*. If the image consists of only one cluster, then the priority computations are complete. If, however, the environment contains several clusters, the algorithm computes the relative priorities of the clusters, the *cluster priority*. Cluster priority thus relates entire objects: if object *A* is nearer to the viewpoint than object *B*, clearly all faces of object *A* take priority over all faces of object *B*. This observation fails when objects *A* and *B* penetrate or

intertwine in a complex fashion, i.e., when they are not linearly separable.

The remarkable thing about these clusters is that within a cluster the priority of faces can be determined independent of the viewpoint. Thus the priority within a cluster can be determined once for all time and need not be recomputed as the viewpoint changes. Priority within a cluster can be independent of viewpoint because a different set of back faces will be removed from the priority list for each viewpoint, and those which remain will assume the proper priority order (experiment with Figure 18).

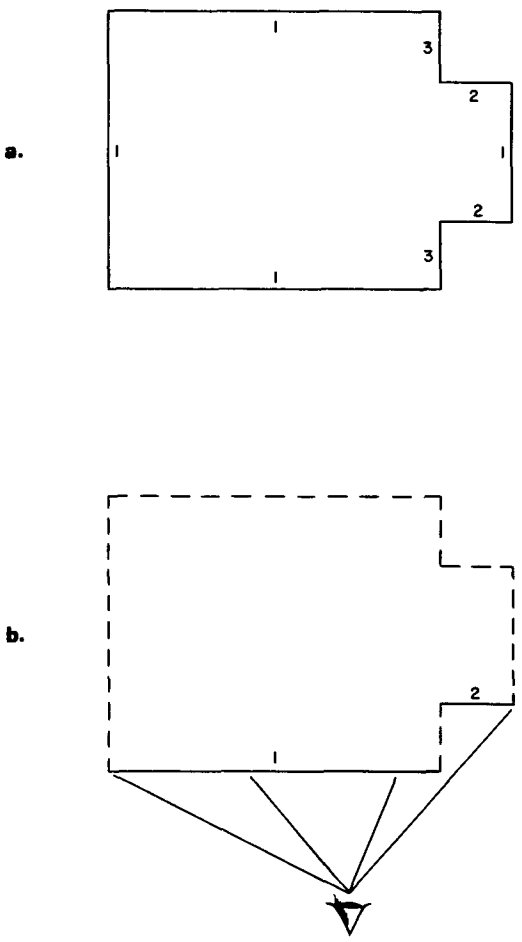


Figure 18 Face priority. *a* Top view of an object with face priority numbers assigned (the lowest number corresponds to the highest priority) *b* The same object with a specific viewpoint located The dashed lines show back faces. Face 1 takes priority over face 2.

The priority index computed by Schumacker can be viewed as a number with integer and fractional parts. *cluster•face.* The face-priority calculation is a property of the topology of the cluster and does not depend in any way on the location of the viewpoint. The cluster-priority computation, on the other hand, is determined by isolating clusters with separating planes defined as part of the data base. As the viewpoint moves, cluster priority depends on the location of the viewpoint relative to the separating planes

Calculating cluster and face priorities independently drastically reduces the amount of computation. In other words, the Schumacker algorithm capitalizes on cluster coherence. The Newell algorithm puts fewer restrictions on the environment by not taking advantage of this coherence.

R. A. Schumacker, B. Brand, M. Gilliland, W. Sharp (1969) [6, 7, 17, 24]

The major contributions of the Schumacker work, as we have already mentioned, are the priority computations based largely on topological properties of the environment, thus utilizing frame-to-frame coherence, and the cluster coherence techniques. In this section, we shall describe in more detail the face-priority computation within a cluster, the cluster-priority computations, and the actual generation of the image in real-time.

The notion that face priority within a cluster can be computed independent of the viewpoint is extremely important. Consider the top view of an object, as shown in Figure 18. If, for any viewpoint, we eliminate the back faces (relative to that viewpoint), the numbers assigned to each face in the figure are the priority numbers. A *cluster* is a collection of faces that can be assigned a fixed set of priority numbers which, after back edges are removed, provide correct priority from any viewpoint.

The computation of face priority requires computing whether face *A* can, from any viewpoint, hide face *B*. If so, face *A* has priority over face *B*. These computations are performed for all faces of a cluster, and a

priority graph is constructed. If there are any circuits in the graph (e.g., face *A* has priority over face *B*, and face *B* has priority over face *A*), the faces cannot be assigned priorities that will produce a correct image. In this case, the cluster will have to be split manually into smaller clusters

The calculation of priority of several clusters is demonstrated by Figure 19. If the viewpoint lies in region *C*, cluster 3 should have priority over clusters 1 and 2. Of these last two, cluster 2 should have priority over cluster 1. These observations can be formalized by deciding where the viewpoint lies with respect to planes that separate the clusters. The tree in the figure shows how the relation of the viewpoint to the two separating planes

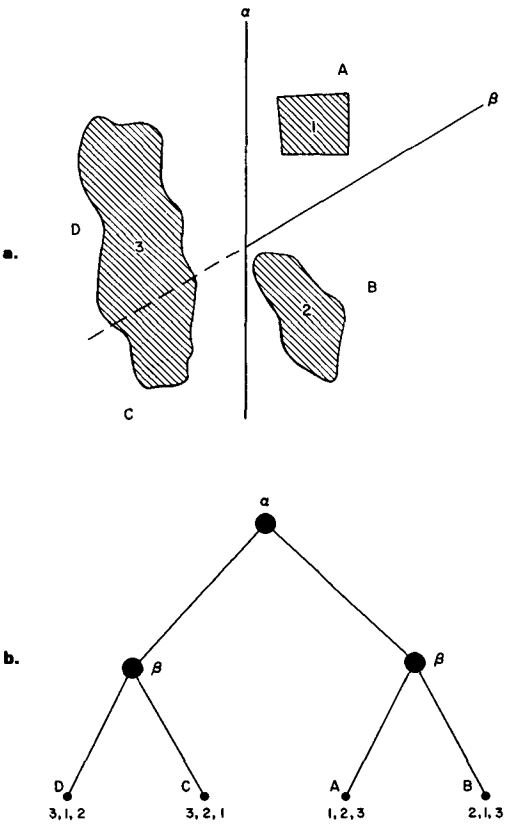


Figure 19 Cluster priority *a* Three clusters (1,2,3) are separated by two planes (α,β) The viewpoint may be located in one of four areas (*A*,*B*,*C*,*D*) *b* A tree structure for finding the cluster priority from the viewpoint location At nodes labeled with planes, we take a branch depending on which side of the plane the viewpoint lies. The result is to sort the clusters into priority order.

produces one of four possible orderings of the three clusters. This concept can be extended to arbitrarily large collections of clusters, provided that separating planes can always be found (i.e., the environment is linearly separable).

Schumacker will tolerate motion of clusters in the environment provided they remain linearly separable. The cluster priority, recomputed for every frame, correctly accounts for the changing depth relationships of the clusters. Of course, if either a cluster or the viewpoint moves, the *X* and *Y* perspective coordinates of the edges and vertices of the faces must be recomputed for every frame.

The generation of the video-image signal is accomplished with a large amount of special-purpose hardware. For each frame, the hardware performs the following operations: 1) the cluster priorities are computed by comparing the viewpoint location to the separating planes in the environment; 2) a list of faces is constructed, in priority order, excluding those that are back faces for the present viewpoint location; and 3) the perspective coordinates of each edge in the environment are computed, giving edge equations in the viewing plane of the form $X_s = A + BY_s$.

For every scan line, the following computations are performed: 1) the X_s intercepts of all edges are updated by adding the incremental value, *B*, to each intercept; and 2) the priority-ordered list of faces is processed to find the X_s intercepts of the faces on this scan line. This procedure assumes that all faces must be convex polygons, and can therefore result in, at most, one segment per face per scan line. Each edge of a face is either a left edge or a right edge; the procedure for a face computes the maximum (rightmost) of the *X* intercepts of its left edges (*L*), and the minimum (leftmost) of the *X* intercepts of its right edges (*R*). If $L < R$, then the face intersects this scan line, and may be visible (see Figure 20).

For every picture element a priority determination is accomplished with special hardware. At the start of the scan line the *L* and *R* intercept values for each face that intersects the scan line are loaded into two

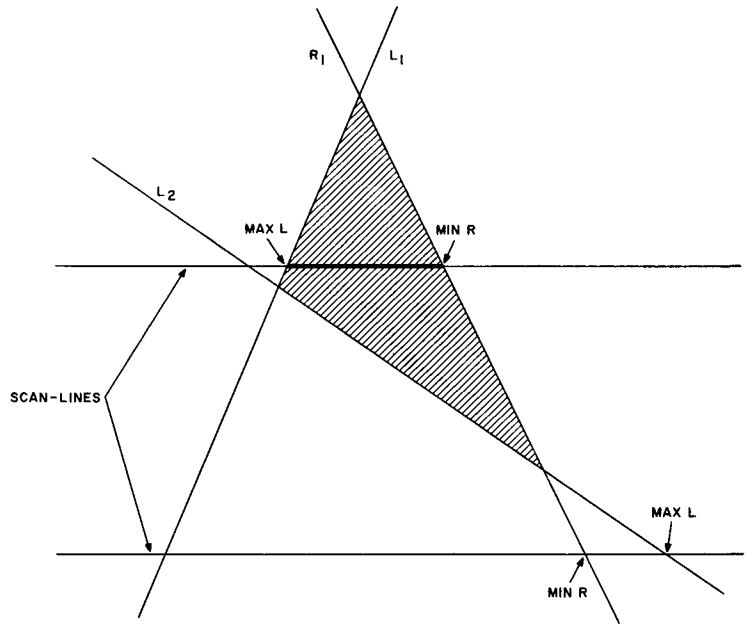


Figure 20 Calcuation of X intercepts of a face on a scan line The triangular face is described by three infinite edges (L_1,L_2,R_1) The maximum of the X intercepts of the left edges and the minimum of the X intercepts of the right edges are calculated. If $max\ L < min\ R$, then the face intersects the scan line in a segment (dark line on the top scan line) Otherwise, the face does not intersect the scan line (bottom scan line)

down-counters. There must be an individual pair of such counters for every face that intersects the scan line Furthermore, the counters are arranged in order: the first intersecting face to be found, the one with the highest priority, is recorded in the first pair of counters, the second in the second, etc.

As the scanning spot progresses across the scan line, the down-counters are decremented for each raster element encountered. Any counters with $L \leq 0$ and $R > 0$ represent faces that might be visible at that raster position. A simple combinatorial logic network decides which counter pair with this property has the highest priority (ie, the lowest-numbered counter pair with this property).

These last two operations, the X sort to determine which faces intersect a particular raster element, and the priority search to determine the visible face are potentially very costly operations However, the hardware realization of these operations is so extremely simple and fast that the scheme becomes feasible. Emulating this process in software would be quite costly.

M. E. Newell, R. G. Newell, T. L. Sancha (1972) [13]

The principal contributions of the Newell algorithm are the development of a priority computer and the concept of "overwriting" faces to achieve the effect of transparency. In the discussion above, we have taken the view that the priority list is used to determine the face that is visible where a number of faces surround a given element of the picture raster. In other words, the priority index is used to announce the visible surface at any spot. Newell views the list in quite a different way: if we write the images of successively higher priority faces successively onto a picture buffer, the picture buffer will have a correct hidden-surface view after we have processed the entire list. Faces of higher priority will overwrite faces of lower priority.

Newell achieved the transparency effect by permitting transparent faces to only partially overwrite the underlying face. If the intensity of the transparent face is greater than that of the underlying face, it becomes the new

intensity directly. Otherwise, the two intensities are combined according to a linear rule to produce the new intensity.

In fact, Newell's algorithm does not store every picture element in the picture buffer as in a video buffer. Instead it stores segments of scan lines, allocating a bucket for each scan line. Each bucket contains a list of the visible segments, which Newell called "beads," for that scan line. As each face from the priority list is painted, its segments are merged into these lists, replacing (i.e., overwriting) any conflicting segments.

The heart of Newell's algorithm is the priority computer This procedure sorts an arbitrary collection of faces into a priority order, not necessarily unique The first step in the procedure sorts all faces by the depth of the furthest vertex of each face. If faces do not overlap in depth at all, this sort successfully establishes the priority order (see Figure 21).

The remainder of the procedure, the "Newell special sort," tests whether the depth-sorted list is indeed in priority order, and if not, fixes the list appropriately. First, we examine the last face on the list, P. If the *closest* vertex of P is deeper than the farthest vertex of Q, the next-to-last face on the list, then P cannot possibly obscure Q. Furthermore, since the furthest vertices of other members of the list are closer still, P cannot obscure any other member of the list, and P may safely be written onto the video buffer.

If, as is more likely the case, P and Q overlap in depth because the closest vertex of P is closer than the furthest vertex of Q, we must use some other test to determine that P cannot possibly obscure Q. In fact, we must use such other tests to compare P with the set of faces $\{Q\}$ at the end of the list which overlap P in depth. If we are successful in proving that P cannot obscure any member of $\{Q\}$, then P may be written onto the video buffer. Notice that the set $\{Q\}$ is not the entire collection of polygons, but only those up to the first one whose furthest vertex is closer to the observer than the closest vertex of P.

If in testing P against the set $\{Q\}$ we discover that P can indeed obscure some polygon, P cannot be written onto the output buffer. Instead we consider as a candidate for next output the offending polygon by placing it at the end of the list and making it serve the role of P. Such a case is illustrated in Figure 22. A marker must be placed on such a displaced polygon so that should we be unsuccessful in proving that it cannot obscure any other, we will not enter a non-terminating loop, but rather conclude that a pair of unorderable polygons exists.

If unorderable polygons exist, that is if Q cannot be written before P and P cannot be written before Q, the priority computer must divide either face P or face Q to eliminate the

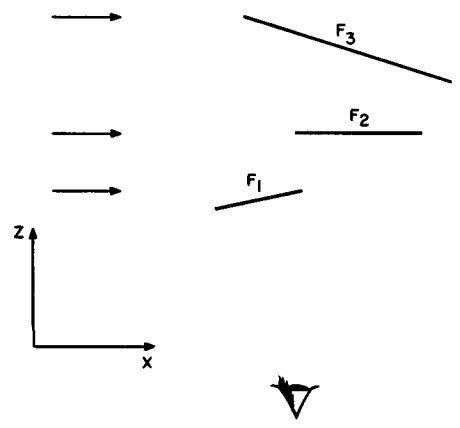


Figure 21· Z sort to determine priority order. If the faces are sorted by furthest vertex from the viewpoint (arrows), the order F_1, F_2, F_3 is produced, which is the correct priority order for these faces

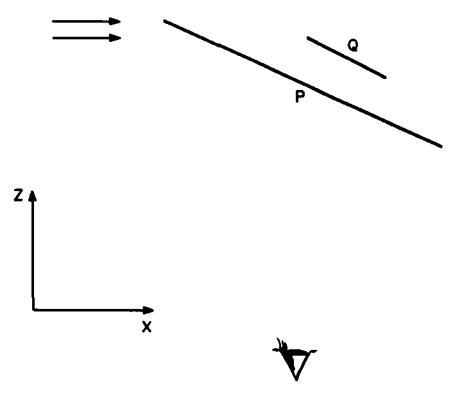


Figure 22 The Z sort fails to place faces Q and P into the correct order (Q,P) However, the Newell special sort will interchange the order and discover that the order (P,Q) is acceptable

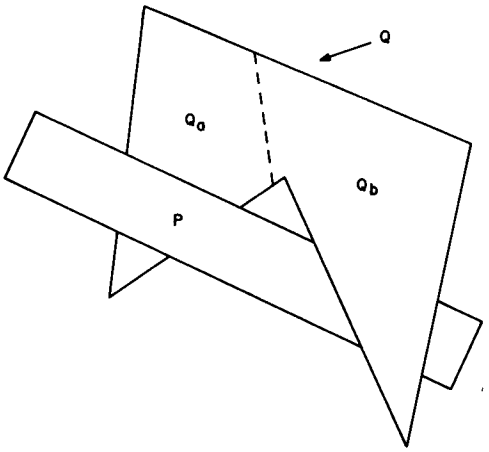


Figure 23. Cyclic overlap. faces P and Q cannot be placed in priority order because they conflict. However, if Q is divided into two faces Q_a and Q_b by the plane of P, then the order (Q_b, P, Q_a) is acceptable.

conflict. This conflict is often called *cyclic overlap* In the example of Figure 23, face Q has been divided by the plane of face P into two faces Q_a and Q_b. These two faces are placed in the priority list; the priority computer will then determine that the order Q_b, P, Q_a is the correct priority order.

The test, "does P obscure Q?" is applied many times and must be made quite efficient. If the answer to the question is "no," then P may be written onto the frame buffer before Q. This condition exists if any of a sequence of increasingly more discriminating tests is true:

1. Test for Z overlap; implied in the selection of the face Q from the Z sort list.
2. The extreme coordinate values in X of the two faces do not overlap (minimax test in X).
3. The extreme coordinate values in Y of the two faces do not overlap (minimax test in Y).
4. All vertices of P lie deeper than the plane of Q
5. All vertices of Q lie closer to the viewpoint than the plane of P.
6. The faces P and Q do not overlap on the screen.

These conditions are tested in the order given here, because they become increasingly difficult to compute. The final test, the overlap condition, is particularly troublesome.

The priority computer used by Newell is capable of finding a priority-ordering of faces for any environment. This is accomplished by dividing conflicting faces until the conflicts are resolved. It should be noted, however, that this division may be computationally expensive. A suitable method is described in [20].

Depth-Priority Algorithms

The depth-priority algorithms divide neatly into two different categories: those that sample areas of the screen (Warnock), and those that sample infinitesimal points on the screen (scan-line algorithms). We shall call these two approaches *area-sampling*, and *point-sampling*.

The aim of the area-sampling approach is to compute an appropriate intensity for every area of the screen. If much of the screen is homogeneous, such as sky or background intensity, the area-sampling approach need only perform one computation for each such homogeneous area. In other words, the algorithm capitalizes on area coherence.

The point-sampling scan-line algorithms are all designed to compute answers to the hidden-surface problem in a form and order suitable for a raster-scan display such as a television monitor. These algorithms compute the intersection of the plane of a scan line and each face in the environment; the line segments resulting from these intersections are called *segments* (see Figure 11). As we shall see, the scan-line algorithms capitalize on the coherence properties of segments: the relations among segments change only slightly from one scan line to the next

The creation of segments simplifies the hidden-surface problem to an analogous problem on segments in two dimensions: segments are measured by X and Z coordinates only. The reduction of the problem from three to two dimensions makes many common computations, such as those that test segments for overlap or depth, simpler than the corresponding tests in three dimensions used in the area algorithms.

This reduction has one serious drawback: the intensity calculated for a raster element cannot be an *average* intensity corresponding to all visible items that fall within the square raster element Instead, the intensity of the entire element is based on computation at one discrete point As a result, objects can "disappear" between scan lines or between raster elements (see Figure 24). Even though the lateral extent of these objects is below the lateral resolution of the screen, it is important that illumination of these objects be included when calculating intensities at surrounding raster elements Similarly, raster elements near edges of large objects must have intensities that represent the average intensity within the raster element, if this average is not computed, ugly "sawtooth" patterns are displayed at object boundaries.

J. E. Warnock (1968) [22]

The Warnock algorithm hypothesizes that sample areas on the screen, called *windows*, can be declared to be homogeneous, and hence can be displayed after a simple shade calculation. The hypothesis is considered correct if 1) no faces fall within the sample window at all; or 2) one face completely covers the window and is nearer the viewpoint than every other face that falls in the window. If the hypothesis cannot be proven true or the proof appears too difficult, the sample window is divided into four smaller sample windows, and each of these is examined analogously. When the size of the sample windows decreases to the size of the raster element, the subdivision process is terminated (see Figure 25). (Actually, we can subdivide until the test window is 1/2 or 1/4 the raster size, and thus compute an average intensity for that raster element)

The procedure for testing a sample window is a cull. A set of faces is compared to the window to see whether the face· 1) surrounds the window; 2) intersects the window, or 3) is completely disjoint from the window (see Figure 26) The complexity of this operation depends on the relation between the window and the face· if the window and the face are laterally disjoint in either X or Y, the cull operation can quickly determine that the face is disjoint from the window. Otherwise, more

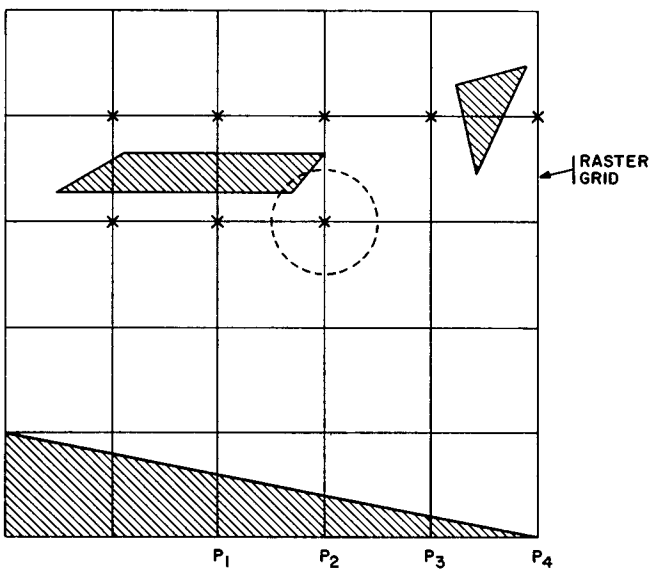


Figure 24 Incorrect shading intensities result unless locations of objects within a raster element are measured The two small objects, which may be brightly illuminated, should contribute to the intensity at the points marked with x's Similarly, the points P_1, P_2, etc should have decreasing intensities because the object does not fill the region underlying the raster dot

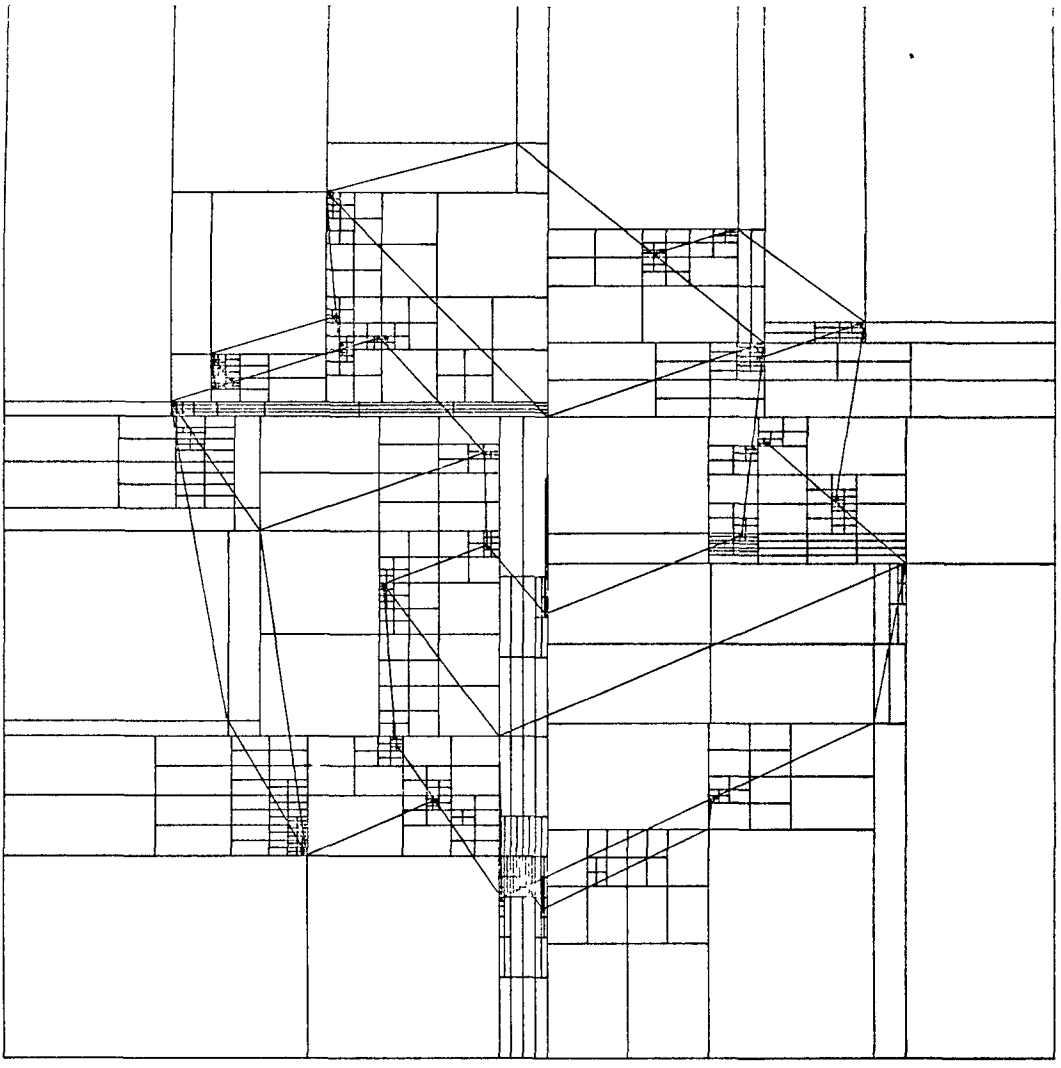


Figure 25 Subdivision by Warnock's algorithm. The object contains three intersecting bricks. In this example subdivision occurs at a vertex if possible.

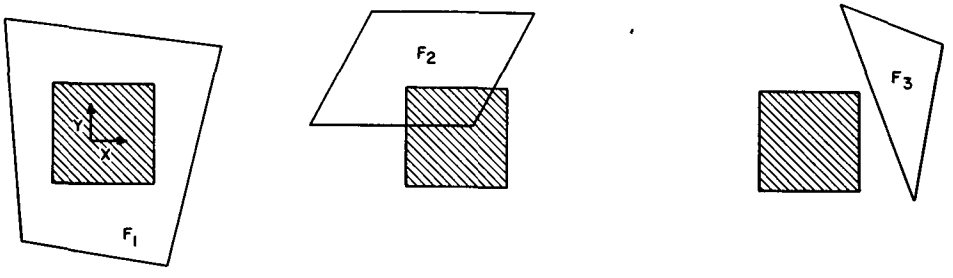


Figure 26 The relationship between a face and a sample window. F_1 surrounds the window, F_2 intersects the window, and F_3 is disjoint from the window. Note that these properties depend only on X-Y relationships, not on Z.

expensive calculations are required [see 4, 18, and 22]

An important concept of the Warnock algorithm is that the hypothesis test for a sample window need not test *all* faces in the environment. If a hypothesis test fails, the four sub-windows to be examined need only be tested against intersectors of the original window; faces disjoint from the large window will certainly be disjoint from the four small windows, and faces which surrounded the original window will surround its descendant windows. The algorithm saves "ancestral information" with each face to avoid needless computation: the surrounder and disjoint properties can both be passed down to sub-windows.

The cull operation is turned into a legitimate sort, the "Warnock Special," by the subdivision operation. In fact, the algorithm bears a striking resemblance to Quicksort: the faces are culled into two groups· those that are disjoint from this window, and those that are relevant to this window. The relevant faces are then passed down to sub-windows, where the faces are again culled according to a new criterion, aspect to the smaller window, and so forth. The process terminates when a window is proven to be homogeneous, just as Quicksort terminates when the lists contain indistinguishable elements. The Warnock cull and subdivision thus become a radix 4 quicksort.

Once the surrounders and intersectors for each sample window have been found, the algorithm must decide whether either of the two homogeneous cases exists Clearly if no intersectors or surrounders are found, the entire window is empty. If surrounders are found, the algorithm searches for the *critical surrounder*, the one that is nearer the eye than all others. This search requires computing the depth of the surrounders at the four corners of the window. These values are compared to determine the closest surrounder. Then, the depths of the critical surrounder are compared against the depths at the corners of the window of the planes of all the intersectors, suitably extended if necessary. If the critical surrounder is closer than all such planes, the window is homogeneous.

If there is no unique critical surrounder, then two surrounder faces must penetrate somewhere within the window. In this case, the Warnock algorithm does not find the window homogeneous; subdivision of the window eventually results in the correct display

There are many variations on the Warnock algorithm: the windows need not be square; we can subdivide the windows at specific points, such as vertex locations, rather than at the center point; windows do not need to be rectangular, the decision procedure can be enhanced to check for a number of simple cases: 1) only one face intersects a window, in which case all portions of the face that intersect the window are visible; and 2) exactly one intersector intersects the window in front of the critical surrounder, in which case the shade for the window can be fairly easily computed, etc

One difficulty with the Warnock algorithm is that its output cannot conveniently be passed to a raster-scan device like a television. The decisions about windows are reached in a seemingly random order, rather than in a top-to-bottom left-to-right order. Cohen has devised a scheme for driving a raster display from window computations, but it involves a massive sort of the windows by Y and X coordinates [4].

Scan-line Algorithms

C. Wylie, G. W. Romney, D. C. Evans, A. C. Erdahl (1967) [16, 25]
W. J. Bouknight (1969) [2, 3]
G. S. Watkins (1970) [23]

The three point-sampling scan-line algorithms are remarkably similar; we shall describe the general philosophy used by all three, and then describe the differences among them, and the particular strong points of each approach All make use of the notions of segment and span defined at the end of the Geometric Computations section above.

All three algorithms perform a Y sort, then an X sort, and finally a Z depth search to establish the visible face The purpose of the

Y sort is to limit the attention of the algorithm, on each scan line, to only those edges or faces that intersect the scan line. Thus the edges or faces are first sorted by Y. As processing for each scan line begins, the Y-sorted structure is examined to find any new edges that enter on this scan line they are added to those already entered. Any edges that terminate on this scan line are discarded.

This feature of the algorithms already takes advantage of one kind of scan-line coherence: the edges that intersect one scan line are very likely to intersect the next scan line It is therefore quite sensible to keep a list of "active" edges and merely make incremental changes to this list as new edges enter or as old edges terminate.

Next, the algorithms examine the reduced list of edges in order to compute which faces are visible in which portions of the scan line. This process involves dividing the scan line into smaller sections, called sample spans, within which the same face is visible. Here, the algorithms capitalize on another form of coherence: point-to-point coherence along the scan line.

The processing of each sample span requires comparing the faces that fall within the span to determine which one is closest. The exact details of this comparison depend on the method of selecting sample spans For example, if sample spans go from edge crossing to edge crossing, then the comparison is quite straightforward· we merely compare the depths of the faces at the limits of the sample span. The procedure must be altered slightly if penetrating faces are allowed [2]

This process is illustrated in Figure 27a. At each X coordinate indicated with a caret, we compute the nearest face; that face is visible at least until the next edge in X order. In the illustration, five sample spans are required to process the scan line. The next edge in X order may quite probably be invisible, however, and so it may be possible to save computation by using more "aggressive" selection of a longer sample span as shown in Figure 27b.

To summarize, the scan-line algorithms have four basic steps: 1) edges are sorted by Y so that only those edges intersecting the current scan line need be examined; 2) on each scan line, appropriate sample spans are determined (this usually involves sorting the edges on the scan line by X coordinate); 3) within each sample span, we must cull out the segments which fall in the span and therefore must be examined, and 4) the segments that fall within a span are searched to find which one is visible. These four operations are called Y sort, X sort, span cull, and Z depth search respectively.

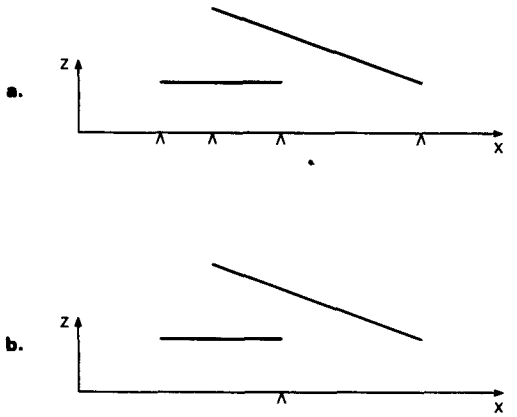


Figure 27· Sample span selection. *a*. Each edge crossing starts a new span *b* Aggressive sample spans The caret divides the scan line into two manageable spans

The algorithm developed by Romney et al was the first to use these main features. The Y sort is a bucket sort; triangular faces are sorted by the Y coordinate of their uppermost vertex. On each scan line, the corresponding Y bucket is used to update a "Y occupied table" that lists all the faces that intersect the scan line. Then the X intercepts of the faces that intersect the scan line are sorted with an X bucket sort. The X buckets are scanned from left to right; an "X occupied table" is kept that records which faces are potentially visible under the current raster element. Whenever a face enters or leaves the X occupied table, Romney's algorithm recomputes the depths of all faces in the occupied table to establish which one is closest. That decision persists until the next change to the X occupied table. Thus the sample spans are determined by edge crossings.

Where penetration is not allowed, Romney

made a very important observation about the *depth* coherence of faces: if exactly the same faces are present on one scan line as in the previous scan line and if the *X* order of their edge crossings is exactly the same, then one need not repeat any of the depth computations. The same faces will be visible as were previously, although their extent in *X* may be different.

Romney failed, however, to capitalize on the coherence of *X* intercepts of edge crossings from one scan line to the next. Both the Bouknight and Watkins algorithms use this coherence by keeping a linear list of edges or segments called the "*X* sort list." When a new scan line is encountered, the *Y* sorted edges that enter on the scan line are merged into the *X* sort list; any edges in the *X* sort list that exit on the new scan line are deleted. Then the list is sorted in *X* with a bubble sort, and because very few edges cross each other from one scan line to the next, the bubble sort is extremely rapid.

Bouknight used this *X* sort list in a fashion analogous to the Romney procedure: in Bouknight's algorithm, edge crossings define the limits of sample spans. As each new span is entered, a new depth computation is performed to decide which face is visible within the span. Bouknight's algorithm marks faces with a "visible" bit whenever the face falls within the current span; the bit is turned off when the sample span moves to the right of an edge of the face. This is precisely analogous to the "*X* occupied table" concept of Romney.

Watkins, however, generated spans more aggressively. In his algorithm the left end of the sample span is fixed and the right end "floats." Initially, the right end coincides with the right end of the scan line. As new segments are extracted from the *X* sort list, the right edge of the sample span may be moved to the left until the situation represented within the sample span is simple enough to compute directly which segment is visible. Watkins' algorithm solves the situation of Figure 27b by placing the right edge of the first sample span at the caret.

The Watkins algorithm also uses a very economical form of *Z* search, a logarithmic search. Rather than solving exactly the plane equations of all faces that lie within the sample span in order to determine their depth, it computes only as much information about the depth as is required to decide which face is visible. Very often, no depth interpolations are needed: the closest edge of one segment is often deeper than the deepest edge of the other segment. This condition does not exist in the example of Figure 28. However, if segment *A* is divided at its midpoint, its left half can be seen to obscure segment *B* for precisely this reason. The midpoint division scheme might ultimately compute the depth of segment *A* at the points indicated by carets. However, the process can often be terminated quickly, as the example of Figure 28 demonstrates.

IV. OBSERVATIONS

Our avowed intent in this activity was to study systematically the existing hidden-surface algorithms to discover what principles they share in common and what distinctions exist between them. We had hoped that such a study might highlight approaches to the hidden-surface problem that not only would be novel, but also would be more efficient than heretofore possible. In order to find such algorithms we need to examine our

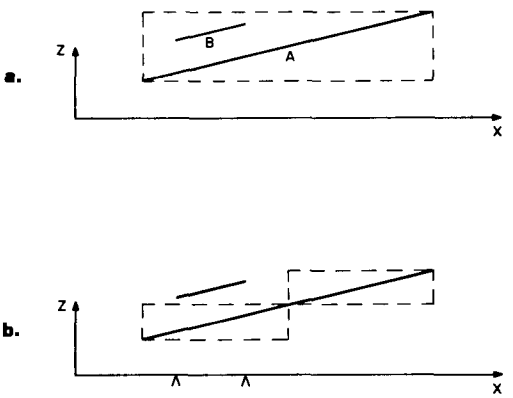


Figure 28 Logarithmic *Z* search. *a* Segment *B* is not farther from the viewpoint than the farthest part of *A*. *b* After one midpoint division, segment *B* can be declared invisible because it lies farther from the viewpoint than the corresponding part of *A*. We have no need to compute accurately the depth of segment *A* at the endpoints of *B* (carets)

categorization tree carefully for missing nodes and for other combinations of the basic operations found in the various algorithms

Use of Coherence

Each of the algorithms was designed to use some form of coherence as the basis for efficiently computing the rendering. In some cases, the use of coherence permits special performance gains in sorting operations; in others, coherence allows incremental calculations to replace more costly direct computations Roberts chose to use object coherence, because he noticed that each object can divide an edge into at most two pieces. Appel, Galimberti and Montanari, and Loutrel all chose to use edge coherence, progressing outward along the network of edges from some starting point in order to promote the known visibility of one vertex along edges to other vertices. Schumacker and Newell both made use of depth coherence to precompute an order of priority for the faces, and Schumacker's algorithm also makes use of cluster coherence to reduce the per-frame computing cost by making some additional investment in environment preparation.

Finally, the four remaining algorithms make use of lateral coherence to reduce the number of surfaces under consideration at any position on the screen by eliminating from consideration those that are laterally displaced. Warnock used a kind of lateral coherence that is symmetric in the X and Y screen directions, whereas the other three, Watkins, Romney et al, and Bouknight, made specific separation between the X and Y processes in order to capitalize on particularly favorable sorting techniques, bucket and bubble sorting.

Let us enumerate the various forms of coherence we have uncovered in the algorithms we surveyed.

Frame coherence· The picture does not change very much from frame to frame.

Object coherence Individual bodies are confined to local volumes which may not conflict Use of clusters is a form of object coherence

Face coherence The faces are generally small compared to the size of the screen and may therefore not conflict. Moreover, penetration of faces is a relatively rare occurrence which may reasonably be allowed to introduce extra cost into the process

Edge coherence: The visibility of an edge changes only where it crosses another contour edge

Implied-edge coherence. If face penetration is detected, the location of the entire implied edge can be extrapolated from two penetration calculations This avoids repeatedly calculating the penetration on each scan line

Scan-line coherence: The set of segments treated on one scan line and their X intercepts are closely related to those of the previous scan line

Area coherence A particular element of the output picture and its neighbors on all sides are likely to be influenced by the same face.

Depth coherence: The different surfaces at a given screen location are generally well separated in depth relative to the depth range of each.

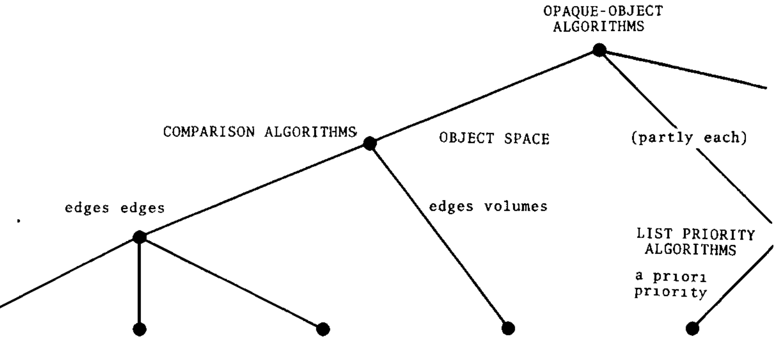
If a new form of coherence were to be discovered, or if a class of environments were to exhibit a particular predominant coherence, the coherence might well be the basis for an entirely new approach.

Environment Restriction Codes

An unrestricted environment is one in which clusters need not be closed polyhedra, faces need not be planar nor convex polygons, penetrating faces are allowed, and faces may be positioned arbitrarily with respect to the observer.

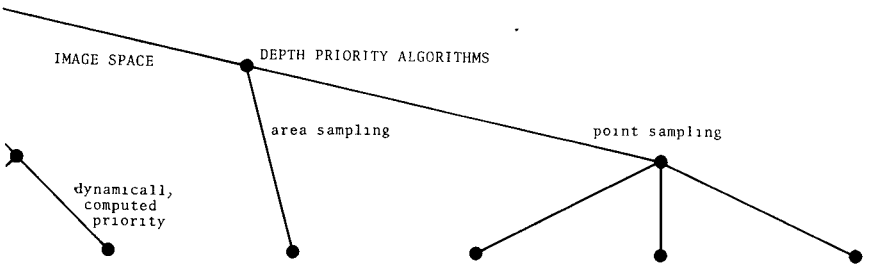
CC All clusters must be closed convex polyhedra.
CF All faces must be convex polygons
TR All faces must be triangular
LS All clusters must be linearly separable.
NP No penetrating faces are allowed
TP The algorithm needs topological information about the environment, ie, faces are classed as adjacent, or clustered in some way
PF Only planar faces are allowed
OF No faces may lie outside the field of view.
BE No faces may lie behind the observer

Figure 29 Characterization of ten opaque-object algorithms *a* Environment restriction codes.



	APPEL 1967	GALIMBERTI, et al 1969	LOUTREL 1967	ROBERTS 1963	SCHUMACKER, et al 1969
RESTRICTIONS	TP,NP	TP,NP	TP,NP	TP, CC, CF, NP	CF, NP, LS (TP)
COHERENCE	Promote visibility of a vertex to all edges at vertex	Promote visibility of a vertex to all edges at vertex	Promote visibility of a vertex to all edges at vertex		Frame coherence in depth No X coherence used
SORTING (1) What, what property (2) Method (3) Type (4) Result structure (5) Number per frame, number of objects (merge) Number of new entries per frame, length of list (search) Number of searches, length of list	Back Edge Cull 1) Edges separating back-facing planes 2) Dot product with normals & topology 3) Cull 4) List of edges, E_s 5) 1, E_t	Back Edge Cull 1) Edges separating back-facing planes 2) Dot product with normals & topology 3) Cull 4) List of edges, E_s 5) 1, E_t	Back Edge Cull 1) Edges separating back-facing planes 2) Dot product with normals & topology 3) Cull 4) List of edges, E_s 5) 1, E_t	Back Edge Cull 1) Edges separating back-facing planes 2) Dot product with normals & topology 3) Cull 4) List of edges, E_s 5) 1, E_t	Intra-Cluster Priority 1) Faces - visibility 2) Dot product with normals 3) Exhaustive search 4) Ordered table 5) 0, (off-line)
	Contour Edge Cull 1) Edges separating front & back faces 2) Dot product with normals & topology 3) Cull 4) List, E_c 5) 1, E_t	(Omitted)	(Omitted)	Clipping Cull 1) Intersect edge with visible volume 2) 3) Cull 4) E_s 5) 1, E_s	Inter-Cluster Priority 1) Clusters 2) Dot product with separating planes 3) Prefix scan binary tree 4) ordered table 5) 1, C_t
	Initial Visibility 1) Ray to vertex against all faces 2) Depth, Surroundedness 3) Exhaustive search 4) Quantitative visibility of vertex 5) # Objects, F_r	Initial Visibility 1) Ray to vertex against all faces 2) Depth, surroundedness 3) Exhaustive search 4) Quantitative visibility of vertex 5) #objects, F_r	Initial Visibility 1) Ray to vertex against all faces 2) Betweenness, surroundedness 3) Exhaustive search 4) Quantitative visibility of vertex 5) #objects, F_r	Edge/Volume Test 1) Edges, visibility relative to volumes 2) Linear Programming 3) Mini-max sort 4) Answer 5) E_s+ split edges, #objects	Back-Face Cull 1) Faces 2) Dot product with face normal 3) Cull 4) Smaller ordered table 5) 1, F_t
	Edge Intersection 1) Intersect one E_s with all E_c 2) Penetration with sweep triangle 3) Cull (unordered) 4) Intersection list 5) E_s, E_c	Edge Intersection 1) Intersect one E_s with all E_c 2) Intersect in picture plane, depth 3) Cull (unordered) 4) Intersection list 5) E_s, E_s - 1	Edge Intersection 1) Intersect one E_s with all E_c 2) Intersect in picture plane, depth 3) Cull (unordered) 4) Intersection list 5) E_s, E_s - 1		Y Cull 1) Faces by Y extent 2) Mini-max on X intercepts 3) Cull (unordered) 4) X intercepts of relevant segments 5) n, E_s
	Sort Along Edge 1) Intersections on edge, ordering 2) Comparison 3) Bubble 4) Answer 5) E_s, X_v/E_s (Omit if well hidden)	Sort Along Edge 1) Intersections on edge, ordering 2) 3) 4) Answer 5) E_s, X_v/E_s (must be done)	Sort Along Edge 1) Intersections on edge, ordering 2) 3) 4) Answer 5) E_s, X_v/E_s (Omit if well hidden		X Sort 1) Segments 2) Counters 3) Hardware 4) Segments at this X 5) nm, S_ℓ
					Priority Search 1) Segments, priority 2) Logic network 3) Logic network 4) visible segment 5) nm, S_ℓ

Figure 29. Characterization of ten opaque-object algorithms *b*. Comparison of the algorithms.



NEWELL, et al 1972	WARNOCK 1968	WATKINS 1970	ROMNEY, et al 1967	BOUKNIGHT 1969
None	(TR) None	None	TR,CF,NP	
None used	Area coherence	Scanline X coherence	Scanline Depth Coherence	Scanline X Coherence
Z Sort 1) Faces, max Z 2) Comparison of max points 3) n logm 4) Ordered table 5) 1, F_r	Z Sort (Opt) 1) Faces, max Z 2) Comparison of max points 3) n log m 4) Ordered table 5) 1, F_r	Y Sort 1) Edges, min Y 2) Comparison 3) Bucket 4) Table of Lists 5) 1, E_r	Y Sort 1) Polygons, Y endpoints 2) Comparison 3) 2 bucket 4) Table of lists 5) 1, F_r	Y Sort 1) Edges, Min Y 2) Comparison 3) Bucket 4) Table of lists 5) 1, E_r
Newell Special 1) Faces, pairwise visibility 2) Depth, bounding boxes, separation 3) Bubble, splitting 4) Ordered table 5) 1,F_r+split faces	Warnock Special 1) Faces with window 2) Depth, mini-max in X and Y, sum of angles 3) Radix 4 subdivision with overlap 4) Stacks of unordered tables 5) L_v, F_r/factor 1	X Merge 1) Edges, X value 2) Comparison 3) Merge (ordred 4) 2-way linked list 5) E_r, S_ℓ	X Sort 1) Edges, X value 2) Comparison 3) Z bucket 4) Table of lists 5) n, S_ℓ	X Merge 1) Edges, X value 2) Comparison 3) Merge (ordered) 4) Linked list 5) E_r, $2S_\ell$ (edges)
Y Sort 1) Face segment by Y range 2) Y intercept 3) Bucket 4) None 5) F_r + split faces H_f^r	Depth Search 1) Surrounder faces 2) 4-corner compare 3) Exhaustive 4) Answer/failure 5) L_v, F_r/factor 2	X Sort 1) Segments, X left 2) Comparison 3) Bubble 4) 2-way linked list 5) n, S_ℓ	X Priority Search 1) Edges, X value 2) Comparison 3) Priority search 4) Active segment list 5) n, m	X Sort 1) Edges, X value 2) Comparison 3) Bubble 4) 1-way linked list 5) N, $2S_\ell$ (edges)
X Merge 1) Segments, X intercept 2) Comparison 3) Ordered merge 4) Ordered list 5) S_r, $S_v/2$	TV Sort (Opt) Sort windows into scan-line order if needed	Span Cull 1) Segments, overlap with sample span 2) Double comparison 3) Cull ordered list 4) Active list 5) $n*S_v$ * f (>1), S_ℓ Z Search 1) Segments, Z 2) Depth by logarithmic search 3) Search (unordered) 4) Visible segment 5) $n*S_v*f(>1)$, D_c	Z Search 1) Segments, depth 2) Linear equations and comparison 3) Search (unordered) 4) Visible segment 5) $n*2S_\ell$, D_c (Omitted if X priorities same as last time)	Z Search 1) Segments, depth 2) Linear equations and comparison 3) Search of unordered active list 4) Visible segment 5) $n*2S_\ell$, D_c

Existing Uses of Coherence

The various algorithms make use of coherence in various combinations Knowingly or not, each of the various authors placed his principal bet on the form involved in his first sorting operation, for all sorting operations except bucket sorting grow more than linearly with the number of items sorted. Let us consider what the principal bets of the various authors were

Roberts bet only on object coherence. While this allows his algorithm to eliminate objects irrelevant to the obscuration of particular edges or particular groups of edges belonging to some other object, the cost of the sorting operation involved still grows with the square of the environment complexity

Appel, Galimberti and Montanari, and Loutrel all bet heavily on the edge coherence of the environment. While this avoids laborious computation of the visibility of each vertex, it still requires comparison of every edge with every other edge, hence growing with the square of environment complexity. The observation that only contour edges need be considered saves about 20% of the effort, but unfortunately makes no change in the fundamental growth of computation as the square of the complexity. These algorithms could realize a considerable improvement by making use of object coherence to eliminate whole groups of edges from consideration for conflicts with a particular edge. Indeed, their authors may well have made this kind of improvement, but the algorithms still face a square-law growth.

Newell and Schumacker bet principally on depth coherence. Newell's algorithm specifically sorts the surfaces into an order that does not conflict in depth; Schumacker's algorithm accomplishes the same thing with separating planes. Indeed these two algorithms are virtually interchangeable once the "priority order" is established. In fact, the Schumacker group built a post-processor much like Newell's for their own use. Newell, therefore, has principally contributed a special kind of bubble sort that produces the priority order for an arbitrary set of input faces.

Both the Newell and Schumacker algorithms make use of implied-edge coherence by dividing polygons wherever they intersect. Newell explicitly computed such intersections; Schumacker insisted on nonpenetrating surfaces in his environment, and included, by implication at least, a pre-processor to make the division if necessary The disadvantage of using this kind of coherence, however, is that the algorithm may have to compute intersections that later turn out to be invisible.

The Schumacker algorithm is the only one that makes effective use of frame coherence. Schumacker was interested in producing not one frame, but a sequence of frames depicting a moving viewpoint traversing a relatively static environment. He was therefore willing to invest heavily in pre-processing of the environment if such an investment could significantly reduce the per frame computation cost. The other authors have been unwilling to make or unable to use such an investment in environment pre-processing. It is clear that one should seek new algorithms that include Schumacker's initial investment ideas or some similar mechanism to capitalize on frame coherence.

Schumacker's key idea, which he calls *clustering*, is indeed a very simple and powerful one He observed that if back faces are culled out by some other process there are many faces which either cannot overlap in any view or always overlap in a specific order. Sets of such faces can have priority numbers assigned that are independent of viewpoint.

Such clusters of faces can be treated as a group in further priority computations, provided extraneous faces do not enter the volume occupied by the cluster. Thus, if a complex environment can be broken down into clusters, the per frame work of computing priority order reduces to computing the priority of the clusters.

Schumacker further observed that the priority of clusters could be computed easily if they were linearly separable. If a plane can be passed between two clusters, then their relative priority can be computed by calculating on which side of the plane the viewpoint lies. If a binary tree of separating planes can be

generated, then the sort into cluster priority order need take only as many computations as there are clusters, and Schumacker has found a linear growth rule. This is an idea whose power has not been fully appreciated.

The remaining algorithms all bet heavily on lateral coherence. Warnock, by specifically making an area sort first, intended to output data for rectangular parts of the picture. His notion has two defects: first, computation of the relation between a face and an area is difficult, and second, no suitable output device is available to absorb the output information an area at a time. The other algorithms, starting with Romney, use area coherence first in Y and then in X, betting heavily that the number of faces "involved" on any one TV scan line of output is significantly less than the total number in the environment. Moreover, a particularly nice form of sorting, bucket sorting, is available for determining the "order of appearance" of faces as scanning progresses.

All of the television output algorithms make some use of scan-line coherence. All observe, for example, that the X intercept of an edge can be computed incrementally by knowing its inverse slope (the change in X position per scan line) and merely updating the X intercept from the previous scan line. Romney also observed that if penetration is disallowed, and no new edge occurs in going from one scan line to the next, then the same visible segments apply in the succeeding scan line. As Romney also knew, his observation was more powerful than his implementation of it, for he applied it only in the unlikely event that an entire scan line was the same as its predecessor, or for a correspondingly unchanged left part of a scan line, rather than maintaining a constant sequence of visible segments in any region untroubled by edge crossing.

Watkins and Bouknight also capitalized on edge coherence by using a bubble sort to keep the active edges in X sort from one scan line to the next. The bubble sort is particularly well matched to this task because edges cross infrequently and, when they do, usually form a pair whose order need only be interchanged to restore correct ordering. Thus not only is the "bubbling" operation required infrequently, but the "bubble" does not have to move very far in the list.

After each culling operation in the various algorithms the number of items left to consider is reduced, often by more than an order of magnitude. On the other hand, the number of times that the resulting smaller number of items must be considered is vastly increased. As one progresses through an algorithm it seems possible to be less and less careful about the type of sorting used because the lists are shorter. The selection of the types of sorts must account not only for the shortened lists, however, but also for the increased number of times they must be sorted. Thus, for example, the linear merge used by Watkins to enter new material into the X sort list after Y sorting is costly only for very complex environments. Similarly, because Watkins made no use of depth coherence at the final stage of his algorithm, preferring rather to search for the frontmost output element, his final output step is very costly for cases with many layers of polygons.

New Uses of Coherence

Having enumerated all evident forms of coherence and having shown how the various authors used them, we can indulge our hindsight to make "improvements" in the algorithms. Our intent in this section is to apply the various forms of coherence to the algorithms that do not already use them to see if useful information emerges. No doubt the various authors have anticipated some of our suggestions, and no doubt our readers may think of more, perhaps better, ideas.

Frame and Object Coherence

We believe that the principal untapped source of help for hidden-surface algorithms lies in frame and object coherence These types of coherence are closely related because the objects presumably do not change between frames and thus any computation done on objects may be preserved from one frame to the next Only Schumacker was able to make significant use of frame coherence, although

Roberts, and the Appel group made some use of object coherence. Frame and object coherence should be powerful aids to the hidden-surface problem because the objects rendered are usually well-behaved and the changes between frames are often minimal. Unfortunately, making use of object and frame coherence requires computation of redundant information about the environment, which must then be saved. It is really hard to make much use of object and frame coherence, so perhaps it is not surprising that not much has been made. Here, however, are a few ideas.

- It makes very good sense to remove interpenetrating faces from the environment at an early stage. For individual objects this process can be done once before the object is ever used. For interpenetrating objects moving with respect to each other some per-frame processing is required. Such processing need not compare every face with every other one if the offending objects can somehow be detected.
- Newell et al might make use of frame coherence by saving the priority order of faces from one frame to the next. Because the priority list already has interpenetration and cyclic-overlap problems resolved, it will presumably require less effort in the succeeding Newell special sort Saving an initial Z depth sort might also make a bubble sort applicable for Z sorting of succeeding frames.
- Newell mentions in his paper the notion of speeding up his special sorting process by using groups of faces rather than individual faces in the comparison process. Schumacker's *clustering* provides an excellent guide for actually doing so.
- Similarly, Warnock might make use of frame coherence by saving a partially sorted list of window/face interactions. Such a breakdown might list which surfaces interact with the 64 or 256 principal regions of the screen and what their interaction is. Incremental changes to such a breakdown on a frame-by-frame basis might be less costly than a complete recomputation.
- The algorithms of Warnock, Romney et al, Bouknight, and Watkins might make use of object coherence in clever ways. However, the bucket sort in Y and the bubble sort in X are already very efficient operations and therefore hard to improve. The use of Schumacker's clustering notions might ease the final Z depth computations, a notion that is considered in the section on sorting order. Knowledge of which edges or surfaces were previously found to be visible might prove very useful in picking sample spans aggressively.

Edge Coherence

Edge coherence might easily be used with the scan-line algorithms.

- If penetration were disallowed or otherwise accounted for, Watkins and related algorithms might ignore non-contour edges as candidates for changing the visibility of some other edge which they cross. Information about which edges are contour edges might also be valuable in computing anti-raster effects on smoothly shaded objects, since contour edges almost always cause discontinuities in shade where the sawtooth effects of rastering are significant.

Scan-Line Coherence

Romney's notion that an area free of edge crossings preserves the visibility of segments might be a powerful tool for reducing the work involved in other algorithms.

- Watkins and Bouknight could compute the number of scan lines until an edge crosses its neighbors in the X sort list, and thus avoid, for that many scan lines, the task of sorting, or even of computing the X intercept for, edges whose quantitative invisibility is high.

Area Coherence

Area coherence can be a powerful tool for the hidden-surface algorithms. If one can sort information by the area of the screen in which it appears, far fewer comparisons than would

otherwise be necessary may suffice The Warnock algorithm is a good example

- Appel, and the related algorithms could achieve a substantial gain in efficiency by sorting edges according to the upper-left corner of the smallest bounding rectangle. An edge could then be compared with a collection of edges very much smaller than the total set of edges, namely those whose bounding boxes overlap. Sorting the bounding boxes in advance could often aid such comparisons
- Similarly, Newell could make a substantial improvement in the cost of his special sorting operation if he were to sort laterally rather than merely culling the list. His X and Y face overlap tests determine as a byproduct whether a given face which does not overlap the test face is above or below it on the screen. This ordering information is neither preserved nor utilized in subsequent testing.
- If Watkins were to sort edges by X position of upper vertex prior to doing his Y bucket sort, his Y bucket sorted list of edges would automatically be sorted by X within each bucket. This presort by X could save him substantial time in the X merging operation

Depth Coherence

Depth coherence might be used in many ways.

- The scan-line algorithms might keep a depth-sorted list of polygons "active" during each segment in the scan line rather than merely remembering the single one which is frontmost. As edges are crossed, new polygons might be entered into this list in the same numerical position, e.g., third, in which they were found to lie in the previous scan line. The correct position would then be confirmed by comparison with adjacent surfaces and corrected by bubbling if necessary. This procedure would not only provide relevant polygons against which to test a new arrival, but also would permit the exciting effects of transparency that Newell et al demonstrated.

Sorting Order

In searching for a new combination of coherences to use, we are struck by the fact that one can consider the order of sorting as a measure of the types of coherence used. This approach suggests considering sorting orders not represented in the existing algorithms. Sorting can occur along a specific dimension, X, Y or Z, or along a combination of dimensions as in Warnock's area (XY) sort. Enumerating all possible orderings of such sorts, we find:

ZYX	Newell and Schumacker
ZXY	Uninteresting variant given use of TV output
YXZ	Romney, Watkins, Bouknight
XYZ	Uninteresting variant given use of TV output
YZX	Untried
XZY	Uninteresting variant given use of TV output
$(XY)Z$	Warnock
$Z(XY)$	Newell's algorithm if a frame buffer memory had been available to him. This scheme was implemented by Schumacker.

An Untried Sorting Order

It is interesting that there is an untried order of sorting. Let us consider what its properties might be. The initial Y sort uses lateral coherence to reduce the number of faces that need be considered at the next sorting step. Use of bucket sorting for the limited resolution required in this sort is particularly attractive. The output of this first step is a table of faces sorted by order of appearance in the scanning process.

The intermediate Z sort might be accomplished by a method similar to Newell's, and because there are fewer faces to sort, the effect of the square law involved will be reduced. Moreover, because newly-entering faces are added to a list from the previous scan

line which is already in order, depth coherence from scan line to scan line can be preserved. Newell's sort is thus simplified to finding an appropriate position in the priority list for each newly-entering face While we do not presume to know whether this indeed helps, the approach seems promising.

The final X sorting step would make full use of the scan-line coherence properties familiar in the other algorithms. X intercepts would be computed incrementally, and kept in X order by a bubble sort. When depth computations are required, a simple test of priority number would suffice. One would make use of scan-line depth coherence by remembering which segments are visible from scan line to scan line and by repeating a visible segment if no edge crossing had occurred involving one of its visible edges. Note that penetration conflicts will have been resolved for entire faces during the Z sort process.

Finally, one would capitalize on depth coherence by keeping an ordered list of involved segments during a span. As scanning progressed, new segments would be entered in this list in the same position as in the previous scan line. From there the correct position would be found by bubble sort. This process would make available an ordered set of surfaces involved with the scanning ray at each point, and thus provide for shadow effects and for the transparency effects Newell so attractively portrayed

Other Combinations of Sorting

One might combine the existing types of sorting in other ways. In its simplest form this approach involves taking ideas from several of the algorithms and reassembling them into some other grouping. In a more complicated form this approach involves using one whole algorithm as a sub-part of another, thus capitalizing on the best features of each. In this section we will explore both of these approaches.

One might use one whole algorithm as a part of another in several ways.

- Warnock's approach is excellent for reducing the number of polygons under consideration in a given area of the screen but suffers from considering too many individual areas. After some subdivision by the Warnock special sort, say until fewer than 100 polygons remained, one might use Newell's method, or Watkins' method to figure out the actual picture within that area.
- Schumacker's method is excellent for capturing frame coherence by use of clusters and linear separating planes One could use Schumacker's approach to compute the inter-cluster priority of objects, passing them in priority order to a Newell-type algorithm which would compute the intra-cluster priority (which would no longer need to be fixed) and write them into a frame buffer. Alternatively a Watkins-type algorithm could be used for the second stage.

In effect, these techniques sort in more than three stages. Instead of sorting in the order $(XY)Z$, the first proposal is sorting only partially (subscript p) in (XY) so that the sorting order is in effect $(XY)_pYXZ$. The second proposal sorts in the order Z_pZYX or Z_pYXZ. Proposals which sort partially in Z first depend on having an overwritable output buffer so that the rendering can be painted in layers. Proposals which sort only partially in other dimensions need not depend on such an output device.

One may also recombine parts of the existing algorithms in new ways.

- Both Catmull and Newell have suggested the application of a Newell-type priority sort as a prelude to Watkins' algorithm. In this proposal one would save the depth computations required by Watkins in the final stages of his algorithm by, instead, computing a priority list as a preliminary step. Thus at the final stage one could determine which of two surfaces was in front by comparison of their priority indices rather than by the more difficult computations that Watkins now uses
- The object-space algorithms of Appel, Galimberti and Montanari, Loutrel, and Roberts could profit a great deal from laterally sorting the edges or objects to be

compared. Such a presort would allow the algorithms to reduce vastly the number of edge/edge or edge/object comparisons.

Warnock recently reported having programmed such an algorithm. Edges are bucket-sorted by northmost Y. Computation moves down the screen in the Y direction. An X-sorted active edge list is kept of all edges involved at the Y position currently being considered As computation progresses, new edges enter the active list. At the Y coordinate corresponding to each such entry the active edge list is resorted in X by a bubble sort. Any edge interchanges are noted, for they indicate edge crossings.

Simple application of new types of sorting might also be productive.

- The algorithms which perform bubble sorts in X might use a sorted tree rather than a sorted list. This would greatly assist in merging in new data, and might significantly reduce the high cost of the X merge step in the Watkins algorithm, for example.
- The image-space algorithms have gone to considerable trouble to avoid sorting in Z. The reason, it seems, is that the resolution preserved in Z is substantially higher than that preserved in X and Y, typically 18 or 20 bits rather than the 9 or 10 used laterally. But if algorithms are to avoid the problems of loss of information between finite scan lines, as implied in Figure 24, additional resolution in X and Y will have to be preserved; indeed the preprocessors used by these algorithms make this resolution available. The X and Y bucket sorts are thus just high-radix quicksorts on the first 9 or 10 bits of 18 or 20 bit key fields. Why not a similar sort in Z?

Perhaps bucket sorting in Z is avoided because the distribution of depths over the available range of Z is uncertain. The extensions of clipping proposed in [20], however, provide a mechanism for guaranteeing that data will be well distributed over the range of Z and thus render this final argument specious. Accordingly, we should consider bucket sorting in Z as a potentially powerful tool for easing the burden of the depth computations.

Sorting Order and Computing Cost

In choosing a sorting order we would like to know if one ordering is likely to lead to a more efficient algorithm than another. Obviously the number of things to be sorted is largest in the first sorting step and decreases rapidly thereafter until very few items remain in the lists that are sorted in the final step. Because, as Watkins observed, the number of overlapping layers of polygons in most objects being rendered is very small, and because efficient sorting processes are available at the finite lateral resolution of the screen, it is tempting to conclude that the lateral sorting algorithms enjoy some fundamental advantage. Indeed, in early drafts of this paper we made this apparently correct assertion. In light of very powerful symmetry arguments advanced by Tom Sancha in his review of these early drafts, we now believe that there is no intrinsic advantage to either lateral or depth sorting as the first sorting step.

Sancha's symmetry argument assumes that the environment is isotropic, i.e., that the statistical distribution of faces is the same in X, Y, and Z. If this is true, a minimax overlap test will be equally effective in reducing the number of polygons that need be further considered regardless of whether the minimax is computed on X, Y, or Z. Thus it follows that the first step of the Newell algorithm, in which polygons are considered only if they overlap the depth of a test polygon, is exactly as effective as the first step of the Watkins algorithm, in which polygons are considered only if their vertical extent on the screen overlaps the current Y coordinate of the scan line. The number of polygons remaining after each initial step is statistically identical. Moreover, the Newell algorithm follows its depth overlap test with overlap tests in X and Y, while the Watkins algorithm follows its Y test with overlap tests in X and Z, and therefore at each of the first three stages both algorithms will be equally effective in reducing the total number of polygons to be considered.

In both cases complex polygon-to-polygon computations are performed only for the very few polygons which overlap each other in all three coordinates. Thus, when viewed in light of environment isotropy, these two very different algorithms can be expected to produce identical performance statistics.

Watkins' statistics [23] showed that relatively few layers of overlapping polygons exist in most environments being rendered. In Appendix A we have quantified this notion as the "depth complexity," D_c, of an environment. D_c is essentially the number of layers of polygons that would be seen at any place on the screen were all polygons transparent. If the environment is isotropic, then this is also the expected number of polygons that will be pierced by an infinite line passing through the environment in *any* direction. A set of such polygons can be culled out by two minimax overlap tests. Similarly, the number of segments in any scan line, S_l, is just the number of polygons that intersect the plane defined by the eyepoint and the horizontal scan line on the display. If the environment is isotropic, then S_l is also the expected number of polygons that will be cut by a plane passing through the environment in any direction. A set of such polygons can be culled out by one minimax overlap test.

Of course, one must be alert to capitalize on any anisotropy of the environment. If one knows in advance that there will be little overlap of surfaces in a particular direction, for example in making a picture of a single bumpy surface, then one should choose the sorting order accordingly. Unfortunately, it is relatively difficult to predict a priori what anisotropies may be expected in the environments being considered.

V. CONCLUSIONS

Concluding that sorting is at the heart of the hidden-surface problem seems inescapable in view of the considerable light such an approach sheds on the various algorithms. In every algorithm examined, the sorting steps are easily defined, clearly separated, and simply described. This view provides the basis for a framework within which to categorize the various algorithms, and thus an approach towards seeking improved algorithms.

The effect of square-law growth of computational complexity is devastating. Although many approaches to the hidden-surface problem are applicable to simple situations, the square-law approaches rapidly become too inefficient as complexity increases. Because the hidden-surface computation is difficult at best, great care must be taken in selecting sorting methods which will conserve computation time by capitalizing on the coherence available in the environments being rendered.

The fact that isotropy of the environment might lead to identical statistical performance of algorithms with very different sorting philosophies suggests that one may have to seek subtler criteria for choosing algorithms. The special characteristics of buffer memories in which to store pictures, or a desire to provide highly accurate lateral computation to avoid the "staircasing" effects produced by the interaction of edges and the picture raster may be the only basis for a choice Of course, whatever basic philosophy is chosen, one must be very careful to provide efficient sorting steps

Finally, and perhaps most satisfying of all, our taxonomic approach seems to provide a substantial basis for future research. Our various suggestions for improvements in algorithms and combinations of algorithms mostly come from systematic examination of our characterization of the ten algorithms studied. Having suggested a framework, the framework itself suggests how to look for improved algorithms.

ACKNOWLEDGEMENTS

We would like to thank Ed Catmull and Martin Newell for their continuing interest in our activities and for many discussions with us. Thanks are also due to the several authors of algorithms who have spent time with us in person, and by telephone, helping us gather an understanding of what they have done. We would especially like to thank Tom Sancha,

Rich Riesenfeld, Elliott Organick, Ted Lee, and the Computing Surveys referees for their critical reading of our manuscript, and for many helpful suggestions. The work reported here was supported by the Office of Naval Research under contract N 00014-72-C-0346. The flexibility of ONR's contracting procedure allowed us to proceed without delay when the ideas and opportunity for this work arose.

BIBLIOGRAPHY

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

[2] Bouknight, W J, "An Improved Procedure for Generation of Half-tone Computer Graphics Representations," University of Illinois, Coordinated Science Laboratory, R-432, (Sept 1969)

[3] Bouknight, W J, "A Procedure for Generation of Three-Dimensional Half-toned Computer Graphics Representations," *Comm. ACM*, 13, 9, (Sept 1970), 527

[4] Cohen, D, "Incremental Methods for Computer Graphics," ESD-TR-69-193, Harvard University, April 1969, (Ph D Thesis)

[5] Galimberti, R, and Montanari, U, "An Algorithm for Hidden-Line Elimination," *Comm ACM* 12, 4, (April 1969), 206

[6] General Electric Corporation, "Modifications to Interim Visual Spaceflight Simulator," Final Report, NASA Contract NAS 9-3916, Feb 1968

[7] General Electric Corporation, "Electronic Scene Generator Expansion System," Final Report, NASA Contract NAS 9-11065, Dec 1971

[8] Gouraud, H, "Computer Display of Curved Surfaces," University of Utah, UTEC-CSC-71-113, June 1971 Abridged version in *IEEE Trans. on Computers*, C-20, (June 1971), 623.

[9] Knuth, D E, *The Art of Computer Programming*, Volume 3, "Sorting and Searching," Addison-Wesley, 1973

[10] Loutrel, P P, "A Solution to the Hidden-Line Problem for Computer-Drawn Polyhedra," Department of Electrical Engineering, New York University, Bronx, New York, Technical Report 400-167, (Sept 1967) (Available from University Microfilm, Ann Arbor, Mich)

[11] Loutrel, P P, "A Solution to the Hidden-Line Problem for Computer-Drawn Polyhedra," *IEEE Trans on Computers*, C-19, 3, (March 1970), 205.

[12] Martin, W A, "Sorting," *Computing Surveys*, 3, 4, (Dec 1971), 147-174

[13] Newell, M E, Newell, R. G, and Sancha, T. L, "A New Approach to the Shaded Picture Problem," *Proc ACM National Conf*, 1972

[14] Newman, W M., and Sproull, R F, *Principles of Interactive Computer Graphics*, McGraw-Hill, 1973

[15] Roberts, L G, "Machine Perception of Three-Dimensional Solids," MIT Lincoln Laboratory, TR 315, (May 1963) Also in *Optical and Electro-Optical Information Processing*, Tipper et al, (Eds) MIT Press, 159

[16] Romney, G W, "Computer Assisted Assembly and Rendering of Solids," Department of Computer Science, University of Utah, TR-4-20, 1970.

[17] Schumacker, R A, Brand, B, Gilliland, M, and Sharp, W, "Study for Applying Computer-Generated Images to Visual Simulation," AFHRL-TR-69-14, US Air Force Human Rescources Laboratory, (Sept 1969)

[18] Sproull, R F; and Sutherland, I E, "A Clipping Divider," *Proc AFIPS FJCC* 1968 *Conf*, Vol. 33, part 1, 765-776

[19] Sutherland, I E, "Computer Inputs and Outputs," *Scientific American*, New York, N Y, (Sept 1966).

[20] Sutherland, I E; and Hodgman, G. W., "Reentrant Polygon Clipping," *Comm. ACM*, 17, 1, (Jan 1974), 32

[21] Sutherland, I E, Sproull, R F, and Schumacker, R. A, "Sorting and the Hidden-Surface Problem," *Proc AFIPS* 1973 *National Computer Conference*, Vol 42, 685-693

[22] Warnock, J E, "A Hidden-Surface Algorithm for Computer-Generated Halftone Pictures," Computer Science Department, University of Utah, TR 4-15, (June 1969)

[23] Watkins, G S, "A Real-Time Visible Surface Algorithm," Computer Science Department, University of Utah, UTECH-CSC-70-101, (June 1970)

[24] Wild, C, Rougelot, R S, and Schumacker, R. A, "Computing Full Color Perspective Images," paper presented at XDS Users Group, 20 May 1972 Also published in General Electric Technical Information Series R71ELS-26, (May 1971)

[25] Wylie, C, Romney, G W, Evans, D C, and Erdahl, A, "Halftone Perspective Drawings by Computer," *Proc AFIPS FJCC* 1967, Vol. 31, 49.

APPENDIX A: STATISTICAL PROPERTIES OF THE RENDERING

This appendix presents a set of statistical measures of the complexity of a rendering. Some of these properties depend directly on the complexity of the environment, i.e., on the number of faces, number of objects, etc. of the environment itself. Some of the measures depend also on the resolution of the picture, the apparent size of the faces, their position with respect to the observer, etc. If some of the environment faces are behind the observer, for example, they need not contribute to the difficulty of the rendering. Similarly, a small isolated object, no matter how complicated, can be rendered as a single dot if viewed from far

enough away. Accordingly we have gathered together a set of statistics characteristic of a particular rendering. We hope this set may provide a basis for meaningful comparison of the performance of different algorithms by providing a basis for specifying the complexity of the rendering task.

Definitions

Many of the environment terms used in this appendix are defined in Section II under the heading, *Environment Complexity Definitions*, to which the reader is referred. The direction and magnification of the view itself and the type of picture being produced might also be described by many "viewing parameters," of which the only one we need is the resolution of the output picture. Some notions result from the interaction between the environment and the viewing parameters, and so we define:

The *depth complexity* is a measure of how many front faces are pierced, on the average, by an arbitrary ray from the viewpoint. If the environment is composed of a large cube standing in front of a back-drop face, the depth complexity would be nearly 2. If the depth complexity of a scene is 1 throughout, the hidden-line or hidden-surface problem is trivial; all relevant faces and edges are visible. As the depth complexity increases, so does the difficulty of rendering the environment.

An environment is said to be *isotropic* if the depth complexity is independent of viewing direction. Another way of looking at this is that the expected number of faces penetrated by any randomly chosen line is independent of the direction of the line. For isotropic environments the average face width and average extent of faces in depth are equal to the average face height, possibly multiplied by a factor to account for different measuring units in the different directions. Surprisingly enough, most environments of any great degree of complexity appear to be nearly isotropic.

The *average face height* of an environment is the average extent of faces in the Y direction measured in resolution units as seen by the observer. Average face height is related to the average number of segments found on a scan line unless the environment is laterally anisotropic. The average face height is also related to the number of faces and the depth complexity of the environment.

A *segment* is the straight line portion of a face defined by its intersection with a plane containing both the observer's eye and a horizontal line in the picture. Segments are of interest because many hidden-surface algorithms compute the output picture one horizontal scan line at a time to correspond with television output devices.

Visible segments are those segments or portions of segments actually seen.

Definition of Environments

It is easy to show that there is a simple relationship between the number of faces, the average face height, and the depth complexity. This relationship is:

$$D_c = F_r \left[(H_f/n)(H_f/m) \right]$$

H_f/n and H_f/m are just the average size of faces expressed as a fraction of the picture height and width, and so the term in brackets is the fraction of picture area occupied by an average face.

Because of this relationship, one can choose freely only two of the parameters F_r, H_f and D_c in setting the complexity of the environment. Any increase in face height for a given number of faces will automatically increase the depth complexity. Notice, however, that moving around in the environment does not change the depth complexity, for if one moves closer to a set of faces, thus increasing their apparent size, one also reduces the number of faces that appear in the picture because some are bound to move beyond the edges of the picture. In fact, halving the distance to a set of faces will double H_f but reduce the number of faces in

TABLE I ENVIRONMENT STATISTICS

F_t	Total number of faces in the environment
F_r	Number of relevant faces in the environment
D_c	Depth complexity of the environment (average).
C_t	Total number of clusters in the environment
C_r	Number of relevant clusters in the environment.
F_c	Number of faces per cluster (average)
E_t	Total number of edges in the environment
E_r	Number of relevant edges in the environment
E_s	Number of relevant edges if sharing is allowed
E_c	Number of contour edges in the environment.
X_r	Total number of edge crossings in the viewing plane
X_v	Number of intersections of visible edges
X_f	Number of face intersections
H_f	Height of a face in resolution units (average)
S_r	Total number of segments, visible or not
S_l	Number of segments on a scan line, visible or not (average)
S_v	Number of visible segments on a scan line (average)
L_v	Total length of visible edges (measured in resolution units)
n	Vertical resolution of screen (number of scan lines)
m	Horizontal resolution of screen.

the picture by a factor of four, thus maintaining D_c. For this reason we have chosen to specify F_r and D_c as independent variables and to compute the resulting value of H_f. Thus, environments with small F_r values become valid representations of detailed views of environments with larger F_r values which are presumably more complex.

We have chosen three levels of complexity for particular environments (see Table II). We chose an intermediate environment first whose complexity seems to be difficult but not impossible with today's state of the art. Finding it too difficult in many cases, we retreated to an environment 25 times simpler, that being about the complexity of many of the test objects used in demonstrating various algorithms. Finally, to think ahead just a few years, we envisioned a giant environment 25 times more complex than our first one.

We have chosen to call our intermediate environment a "harbor scene." It is inspired by thinking of a representation of New York harbor as it might be presented on a ship simulation system. The complexity afforded would provide quite a recognizable appearance. The giant environment we envision is the same environment with much more detail included, e.g., windows in buildings, finer representation of curved surfaces. The simple environment we call "Roberts' house" having in mind his early architectural example which can be seen in [19].

The most controversial number in our environments is the depth complexity. Most of the other numbers in the environment statistics can be derived by simple reasoning once one has decided how many faces there shall be and what the depth complexity is. The number of faces is, of course, our variable, and so we can select it relatively free of criticism. The depth number, however, represents the degree to which faces overlap In the harbor environment the depth number will vary from unity, when looking at the sky or ocean, to quite a high number, when looking at a collection of tall buildings.

Deriving the Environment Statistics

The various statistics for the environments are mostly derivable from the number of faces and the depth number. The following discussion describes the derivation of the elements of Table II. We assume that the

TABLE II STATISTICS FOR THREE ENVIRONMENTS

Statistic	*Rule of Thumb*	*Roberts' House* (1/25)	*Harbor* (1)	*Big Harbor* (25)
n	given	500	500	500
m	given	500	500	500
F_r	given	100	2500	60000
F_c	given	10	25	200
D_c	given	3	3	3
F_t	$2\,F_r$	200	5000	120000
C_t	F_t/F_c	20	200	600
E_t	$4\,F_t$	800	20000	480000
E_r	$E_t/2$	400	10000	240000
E_c	$E_r/(F_c/2)^{1/2}$	180	2800	24000
E_s	$(E_r-E_c)/2+E_c$	290	6400	130000
X_r	$(D_c-1)E_r/4$	200	5000	120000
X_v	X_r/D_c	70	1700	40000
H_f	$(nmD_c/F_r)^{1/2}$	86	17	4
S_t	$(D_cF_rm/n)^{1/2}$	17	87	420
S_v	S_t/D_c	5	29	140
L_v	$2\,n\,S_v$	5000	29000	140000

number of faces selected for consideration is the number of relevant faces rather than the total number, since the initial clipping cull and back-face cull are universally applicable. The number of clusters is derived by choosing 25 faces per cluster, a number said by Schumacker to be reasonable. In the simple environment we have cited only 10 faces per cluster, reasoning that that is appropriate, and in the complex environment we have 200 faces per cluster, arguing that the greater detail will not prevent a clustering similar to that of the harbor environment.

The total number of edges is computed as 4 times the total number of faces, since most faces are rectangular. The number of relevant faces or edges is computed by assuming half of the faces are back faces and will be eliminated by the early cull. The number of contour edges for large arrays of faces can be found by dividing the number of relevant edges by the square root of the number of visible faces of an object.

Because our definition of an edge is at variance with the definitions used in some of the papers, we define E_s, the number of relevant edges if edge-sharing is permitted. This is simply the sum of the contour edges (which cannot be shared), and 1/2 the non-contour relevant edges.

The total number of edge crossings in the viewing plane is a difficult statistic to estimate. Our formula is admittedly just a guess, but it has the virtue of growing with both D_c and E_r, and is correctly zero when D_c=1. In fact, we estimate that for D_c=3, every edge will cross approximately one other, and thus that the number of edge crossings is $E_r/2$.

The height of a face in resolution units is computed as the square root of the average area of a face, considering the number of faces and the depth number. If there are nm square resolution units of total picture area and a depth number of D_c, then there are nmD_c units of picture area covered by F_r faces, and each must occupy an area of nmD_c/F_r. The average length of a segment on each scan line is the same as the average face height, and from this we can compute the number of segments on a scan line, $S_l = mD_c/(\text{average length}) = mD_c/(nmD_c/F_r)^{1/2} = (mD_cF_r/n)^{1/2}$. Another way of looking at this is that there are D_c layers of uniform faces laid out over the screen area, each layer having F_r/D_c faces. If n and m are equal, then there must be $(F_r/D_c)^{1/2}$ faces per layer involved in any one scan line. Thus $S_l = D_c(F_r/D_c)^{1/2}$. The number of visible segments is obviously $1/D_c$ times this number.

The total length of visible edge is computed as twice the number of visible segments. Each visible segment contributes one unit of visible edge length per scan line on which it lies if the edge is vertical. But edges which are horizontal are not represented at all, so we apply the factor of two.

Analysis of the Algorithms

The environments given in Table II have been used to obtain crude estimates of the performance of the ten hidden-surface algorithms we have considered in this paper. The results of this analysis are contained in Appendix B.

APPENDIX B: SOME STATISTICAL ESTIMATES OF COMPUTING COST

This appendix explores the performance of the algorithms on the three artificial environments. The numbers we have derived are but crude estimates: nevertheless the vast variance, typically three orders of magnitude, among the performances of the various steps serves to highlight the relative difficulty of the various steps. There may be substantial disagreement with the environments we have used, with the complexity factors we have ascribed for each step, and thus with the resulting "cost" numbers. Furthermore, these "cost" measures in no way quantify other important properties of the algorithms: space required, feasibility of hardware implementations, etc. This appendix is intended to be helpful, not authoritative.

Complexity Factors

In our rough calculations we associate a "complexity factor" with each of the operations listed in Figure 29. These factors are based on the difficulty of the particular tests or computations performed in the various steps. A complexity factor of 1 is assigned to simple operations such as comparing two numbers or solving a plane equation. A complexity factor of about 10 is assigned to more difficult problems, for example, computing the relationship between two segments in two dimensions (X-Z) The complexity factor of about 100 is assigned to very complicated operations, such as Roberts' edge/volume test, and Warnock's cull of faces based on their relation to a sample window. The actual complexity factor chosen is usually immaterial to the lessons of the computations which follow.

Statistics for the Various Algorithms

The statistics for the three scenes are given in Table II, and the "performance" of the algorithms in Table III. Brief justifications of the expressions listed in Table III follow:

Roberts

(1) The back-edges cull is performed on the set of edges; complexity factor 1.

(2) The clipping cull is performed on relevant edges (the algorithm permits edge sharing); complexity factor 100.

(3) The edge/volume test must test each relevant edge against all objects in the environment. The factor f is included to account for two facts: more than E_s edges are tested because they become split into fragments by previous tests; C_ℓ must be somewhat higher for Roberts' algorithm than for others because it allows only convex objects. The complexity factor of this test is 100.

Appel, Loutrel, Galimberti and Montanari

(1) The back (and contour) edge cull is performed on the total number of edges; complexity factor 1.

(2) The initial visibility search is performed once for each object in the environment, and must examine all faces to count the number of hiding faces; complexity factor is 100.

(3) The edge intersection test must test each relevant edge against each contour edge; the complexity factor is 30.

(4) The invisibility correction is performed twice for every relevant edge, and must examine about 3 faces each time; the complexity factor is 100.

(5) The sort along the edge sorts X_r/E_s things, on the average (less than 1 for our three scenes), for each relevant edge; complexity factor 1.

Schumacker, et al

(1) The determination of intra-cluster priority requires comparing each face of a cluster with other faces of the cluster, and must be performed for each cluster. The complexity factor is 100 (this is generous· this step often involves human intervention in the programs implemented at General Electric)

(2) The inter-cluster priority computation requires locating the viewpoint with respect to all separating planes, and traversing the binary tree set up by step (1): the determination must be made for each cluster in the environment; complexity factor 10.

(3) The back-face cull removes from consideration all back faces; complexity factor 1.

(4) The Y cull is performed on each scan line, and processes E_s potentially-visible edges, complexity factor 1.

(5) The X sort and priority search steps are combined, S_l potentially-visible faces are examined at each raster element; complexity factor 1.

Newell, et al

(1) The Z sort can be performed with a two-pass radix 512 Quicksort, hence the performance 2 F_r; complexity factor 1.

TABLE III COSTS FOR THREE ENVIRONMENTS

Roberts			
1 Back-facing edges cull			
E_t	800	20K	480K
2 Clipping cull			
100 E_s	29K	640K	13M
3 Edge/volume test			
100 $E_s f C_t$	2 3M	510M	31B
f = 4, split edges, and C_t should be higher			
Appel, Loutrel, Galimberti and Montanari			
1. Back (and contour edge) cull			
E_t	800	20K	480K
2 Initial visibility search			
100 $C_t F_r$	200K	50M	36B
3 Edge intersection			
30 $E_s E_c$	16M	540M	93B
4 Invisibility correction			
30 (2 E_s 3)	52K	12M	23M
5 Sort along edge			
$E_s(X_r/E_s) \log_2 (X_r/E_s)$	290	64K	130K
Schumacker et al			
1 Intra-cluster priority			
100 $F_c^2 C_t$	200K	12M	2.4B
2. Inter-cluster priority			
10 C_t	200	2K	6K
3. Back-face cull			
F_r	100	25K	60K
4. Y cull			
$n E_s$	150K	32M	65M
5. X sort and priority search			
$nm S_t$	42M	22M	100M
Newell et al			
1. Z sort			
2 F_r	200	5K	120K
2. Newell special			
$F_r^2(f+f^2+100f^3)$	45K	650K	60M
$f = 2 H_f/n$			
3 Segment generator and Y sort			
10 $F_r H_f$	86K	420K	2.4M
4. X merge			
$F_r H_f S_v/4$	11K	310K	84M
Warnock			
1. Z sort			
2 F_r	200	5K	120K
2 Warnock special cull			
100 $L_v D_c$	15M	87M	42M
3. Depth search			
$L_v D_c$	15K	87K	420K
Romney et al			
1 Y sort			
2 F_r	200	5K	120K
2 X sort			
$n S_t$	85K	43K	210K
3 X priority search			
nm	250K	250K	250K
4. Depth search			
20 n 2 $S_t D_c f$	510K	26M	13M
f = 1/2, due to depth coherence			
Watkins, Bouknight			
1. Y sort			
E_r	400	10K	240K
2 X merge			
$E_r S_t/2$	34K	430K	50M
3 X sort			
$n(S_t + 10 X_r/(nS_t))$	85K	43K	210K
4. Span cull			
$n S_t$	85K	43K	210K
5 Depth search			
30 $n D_c$ $min(m, fS_v)$	450K	26M	13M
f = 2, spans include not only visible segments			
Brute-force image space			
No memory			
100 nmF_r	25B	62B	1500B
Large memory			
10 $H_f^2 F_r$	75M	7.5M	7.5M

[*Note*: K=1,000, M=1,000,000]

(2) The Newell special sort examines faces for X, Y, or Z overlap with all other faces. Only those faces that overlap in all three components require complex tests and polygon divisions The overlap tests are assigned complexity 1, and the complex tests complexity 100. Thus the total cost is $F_r^2 (f + f^2 + 100f^3)$ where f is the probability that two faces overlap along one dimension (this probability is the same for all three dimensions if the scene is isotropic). We can estimate the probability of overlap in Y as follows: a region H_f scan lines high will overlap another such region with probability $f = (H_f + H_f)/n$.

(3) The segment generator and Y sort required to place the segment in the simulated frame buffer are performed for all relevant faces; approximately H_f

segments are generated for each face; complexity factor 10.

(4) The X merge into the list of segments for each scan line is performed $F_r H_f$ times; the average size of the list at the time of the merge is $S_v/2$, but we must search only about 1/2 this list to find the correct spot for the new segment; complexity factor 1.

Warnock

(1) Again, we assume the initial Z sort may be accomplished with a radix 512 Quicksort, complexity factor 1.

(2) The Warnock special cull is performed approximately L_v times (the tree of sub-windows has most of its nodes near the terminal nodes); the approximate average number of faces intersecting a sample window is D_c (again, this number is representative of the terminal nodes rather than of the root node); the complexity factor is 100.

(3) The depth search is performed as often as the special cull on D_c faces but has complexity factor 1.

Romney, et al

(1) The Y sort of faces is a bucket sort (actually, *two* bucket sorts); complexity factor 1

(2) The X bucket sort is performed on each scan line for approximately S_l faces; complexity factor 1.

(3) The X priority search is performed for each raster element; complexity factor 1.

(4) The Z depth search is performed at each edge intersection ($2\ S_l$), on D_c faces, on each scan line. The factor f is less than one because Romney can sometimes avoid depth searching altogether because of depth coherence, the complexity factor of the search is 20 because plane equations must be solved Although the numbers for Romney's algorithm appear very favorable, the algorithm handles only non-penetrating triangular faces in the environment.

Watkins

(1) The Y bucket sort is performed on all relevant edges; complexity factor 1.

(2) The X merge is performed once for every relevant edge; the average length of the list at the time of the merge is S_l; we need search only 1/2 of the list to find the correct spot; complexity factor 1.

(3) The X bubble sort is performed on each scan line; at least S_l compares are required to verify that the list is in sort; about $X_r/(nS_l)$ elements are out of sort: they are usually only interchanged with immediate neighbors, hence a complexity factor of 10.

(4) The span cull involves testing each segment on each scan line.

(5) The Z search is performed for each sample span (the maximum number of such spans is the number of visible segments, multiplied by a factor to account for the fact that sample spans must occur slightly more frequently than visible segments). D_c elements are searched in each span; the complexity factor of the logarithmic search is 30.

Brute-Force Image-Space

This expression represents the cost of computing the visible surface at each raster element by a brute-force examination of each plane to see which ones fall on the raster element, and which of these is closest. This statistic is presented for comparison with the other algorithms.

If a large memory is available, large enough to store a color and a depth at each point in the output picture raster, one can simply compare the already-recorded depth at each point within a polygon with the depth for the new polygon. If the new polygon is closer, its data replaces that already in the memory. This method results in a computing cost which depends only on the depth number D_c and not otherwise on the environment complexity. Some hazard with edge effects exists with this algorithm, however.

Statistical Results

In spite of the crude nature of the statistical numbers that we have presented in this appendix, some interesting observations can be drawn from them. One must, of course, refrain from concluding that one algorithm is "better" than another by virtue of having a "cost" number half that of the "worse" one, because the estimates are not nearly accurate enough for such a conclusion. On the other hand, when one algorithm appears to be 100 or 1000 times the cost of another, it is harder to argue that errors in our estimates are at fault. Thus we feel free to make order of magnitude comparisons between the various algorithms to learn something about the effectiveness of the various methods.

We will be examining the numbers in Table VII to see which algorithms provide roughly equivalent "costs" for various levels of environment complexity. In each case we will refer back to Tables IV, V and VI to discover which steps in the algorithms account for the bulk of the cost. It is interesting that in almost every case one step in the algorithm accounts for a preponderance of the "cost." The effects of square-law growth of "cost" with complexity show clearly in these numbers. Although the specific values of the numbers may be wrong by an order of magnitude or two, the lesson of the importance of avoiding square-law growth remains valid and important.

At the level of complexity represented by the simple environment nearly all of the algorithms come out with a "cost" on the order of one million. The variation from 500K for Watkins to 4M for Schumacker et al, is not significant, considering the crudity of our estimates and the fact that Schumacker's final computation steps, though many in number, are particularly simple to implement in

TABLE IV. COST SUMMARY: SIMPLE ENVIRONMENT

Roberts	*Appel, Loutrel, Galimberti and Montanari*	*Schumacker et al*	*Newell et al*	*Warnock*	*Romney et al*	*Watkins, Bouknight*	*Brute force*
		Intra-cluster priority (200K)[*]					
Back edges cull 800	Back edges cull 800	Inter-cluster priority 200	*Z* sort 200	*Z* sort 200	*Y* sort 200	*Y* sort 400	
Clipping cull 29*K*	Initial invisibility search 200*K*	Back faces cull 100	Newell special 45*K*	Warnock special 1 5*M*	*X* sort 8 5*K*	*X* merge 3 4*K*	
Edge/volume tests 2 3*M*	Edge intersection tests 1 6*M*	*Y* cull 150*K*	Segment generator 86*K*	Depth search 15*K*	*X* priority search 250*K*	*X* sort 8 5*K*	
	Invisibility correction 52*K*	*X* sort and priority 4.2*M*	*X* merge 11*K*		Depth search 510*K*	Span cull 8 5*K*	
	Sort along edge 290					Depth search 450*K*	
2 4*M*	1 8*M*	4 2*M*	140*K*	1 5*M*	770*K*	470*K*	2 5*B* or 7 5*M*

[*] Not charged in per frame cost.
[*Note*: *K*=1,000; *M*=1,000,000]

TABLE V. COST SUMMARY: MIDDLE ENVIRONMENT

Roberts	*Appel, Loutrel, Galimberti and Montanari*	*Schumacker et al*	*Newell et al*	*Warnock*	*Romney et al*	*Watkins, Bouknight*	*Brute force*
		Intra-cluster priority (12*M*)*					
Back edges cull 20*K*	Back edges cull 20*K*	Inter-cluster priority 2*K*	*Z* sort 5*K*	*Z* sort 5*K*	*Y* sort 5*K*	*Y* sort 10*K*	
Clipping cull 640*K*	Initial invisibility search 50*M*	Back faces cull 2.5*K*	Newell special 650*K*	Warnock special 8.7*M*	*X* sort 43*K*	*X* merge 430*K*	
Edge/volume tests 510*M*	Edge intersection tests 540*M*	*Y* cull 3.2*M*	Segment generator 420*K*	Depth search 87*K*	*X* priority search 250*K*	*X* sort 43*K*	
	Invisibility correction 1.2*M*	*X* sort and priority 22*M*	*X* merge 310*K*		Depth search 2.6*M*	Span cull 43*K*	
	Sort along edge 6.4*K*					Depth search 2.6*M*	
510*M*	590*M*	25*M*	1.4*M*	9*M*	2.9*M*	3*M*	62*B* or 7.5*M*

* Not charged in per frame cost
[*Note:* *K*=1,000; *M*=1,000,000]

hardware. Thus the only variation of interest here is Newell et al, an order of magnitude less "costly" and the brute-force approach which is already ridiculously expensive.

This remarkable similarity in the "cost" of algorithms at the simple model level of complexity may not be entirely coincidental, for this is the level at which most demonstration programs operate. An algorithm several orders of magnitude more expensive at this environmental complexity would not survive long enough to appear in the literature, and one significantly less expensive would already have been hailed, and recognized, as a major breakthrough.

Reference to Table IV shows that the edge/volume test of Roberts and the edge/edge tests of Appel and related algorithms are already their dominant costs. Similarly, the Warnock special (*XY*) sort is the major contribution in his case. The Watkins and Romney algorithms, on the other hand, are dominated by the depth search which they laboriously perform over and over on a scan-line by scan-line basis for the relevant segments. It appears that Newell, by avoiding the square-law growth of edge tests and the fine resolution of *Z* depth tests, has managed to gain an order of magnitude edge in "cost" for this level of complexity. Of course, this advantage may be an illusion brought about by incorrect estimation of the complexity of some operation. It nevertheless seems reasonable to point out that for simple pictures the image-space algorithms are forced by the resolution of the screen.

At the middle level of complexity, a quite complicated picture by today's standards, the situation is quite different. The edge/edge and edge/volume algorithms of Roberts and the Appel group have become impractical. Reference to Table V shows that this is

TABLE VI COST SUMMARY: COMPLEX ENVIRONMENT

Roberts	*Appel, Loutrel, Galimberti and Montanari*	*Schumacker et al*	*Newell et al*	*Warnock*	*Romney et al*	*Watkins, Bouknight*	*Brute force*
		Intra-cluster priority (2.4*B*)[a]					
Back edges cull 480*K*	Back edges cull 480*K*	Inter-cluster priority 6*K*	*Z* sort 120*K*	*Z* sort 120*K*	*Y* sort 120*K*	*Y* sort 240*K*	
Clipping cull 13*M*	Initial invisibility search 36*B*	Back faces cull 60*K*	Newell special 60*M*	Warnock special 42*M*	*X* sort 210*K*	*X* merge 50*M*	
Edge/volume tests 31*B*	Edge intersection tests 93*B*	*Y* cull 65*M*	Segment generator 24*M*	Depth search 420*K*	*X* priority search 250*K*	*X* sort 210*K*	
	Invisibility correction 23*M*	*X* sort and priority 100*M*	*X* merge 84*M*		Depth search 13*M*	Span cull 210*K*	
	Sort along edge 130*K*					Depth search 13*M*	
31*B*	97*B*	170*M*	71*M*	43*M*	14*M*	64*M*	1500*B* or 7.5*M*

[a] Not charged in per frame cost.
[*Note:* *K*=1,000, *M*=1,000,000]

TABLE VII. COST SUMMARY: THREE ENVIRONMENTS

Roberts	*Appel, Loutrel, Galimberti and Montanari*	*Schumacker et al*	*Newell et al*	*Warnock*	*Romney et al*	*Watkins, Bouknight*	*Brute force*
2.4*M*	18*M*	42*M*	140*K*	15*M*	770*K*	470*K*	24*B* or 7.5*M*
510*M*	590*M*	25*M*	14*M*	9*M*	29*M*	3*M*	62*B* or 7.5*M*
31*B*	97*B*	170*M*	71*M*	43*M*	14*M*	64*M*	1500*B* or 7.5*M*

[*Note.* *K*=1,000; *M*=1,000,000]

because of the square-law growth property of their basic edge/volume and edge/edge tests. The other algorithms are all about equal in performance, but notice that the "special sort" is now the major contribution to the Newell algorithm cost. The cost of depth searching for the image-space algorithms has not increased very much, nor has the cost of Warnock's special sort.

At the level of complexity of our very complicated environment, only the Schumacker et al, Newell et al, Warnock, Romney, and Watkins algorithms remain in contention. We see Romney again dominated by the depth-searching operations, but Watkins is now dominated by the X merge step, again a step with square-law growth. The Newell et al algorithm is intermediate. If D_c remains fairly constant in the environments, the cost of the Newell special sort grows as $F_\gamma^{3/2}$, thus

staving off only slightly longer the explosion of computation. Only Schumacker et al, Romney et al and Warnock remain free of this distressing difficulty.

The lessons of these numbers are quite clear: avoid those types of sorting whose cost grows as the square of complexity. The bucket sorting technique used by Romney, Watkins and Bouknight has a most desirable linear cost growth The depth searches required by these algorithms grow in cost only with the square root of the number of surfaces, because the surfaces get smaller as there become more of them. In effect the image-space programs allow a cruder approximation to the correct picture as the environment becomes more complex and the detail becomes correspondingly smaller.