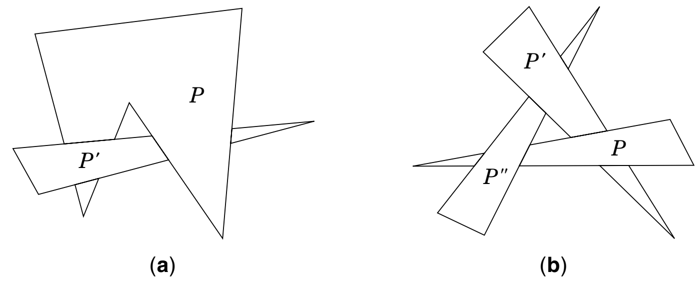


**Figure 1.** (a) $P$ and $P'$ overlap each other. (b) $P$, $P'$, and $P''$ cyclically overlap, forming what is often called a *priority cycle.*

# HIDDEN FEATURE REMOVAL

## BACKGROUND

To render a scene correctly, we must determine which parts of which objects can be seen from the given viewing position. Techniques used to identify the visible portions of objects are called visibility algorithms, hidden surface removal algorithms, or hidden feature removal algorithms. As an example, let us suppose that we are trying to render a scene where a man stands in front of a tree. It would be incorrect to have the tree showing where the man should be. One popular approach for producing a correct image is to render the tree first and then to render the man over the old image, thus obscuring the invisible portion of the tree. (This technique is called the painter's algorithm, which is explained later in more detail.) Now suppose we want to render a forest using the painter's algorithm. Most of the trees in the back will be completely obscured by the trees in the front. If we somehow knew in advance which trees would be completely obscured, then we could save time by not rendering these trees at all, which brings us to another reason that hidden feature removal algorithms are used—that is, efficiency. If only a small fraction of the environment is visible from a viewing position, then we can save a great deal of time by only rendering the visible objects.

There are two broad categories of visibility algorithms: *object-precision algorithms,* which work with the original object definition to determine which portion of each object is visible from the viewpoint and produces output in a similar format to the input, and *image-precision algorithms,* which determine which object is visible at each pixel. An image-precision algorithm produces a solution that has a particular level of resolution. In contrast, object-precision algorithms can produce solutions with the same level of accuracy as the original object definition. Because rendering databases have become larger, it has become increasingly common to use object-precision techniques to cull large sections of the rendering database and then to use image-precision techniques to resolve visibility for the remaining objects.

This article concentrates on visibility for rendering databases that consist of collections of polygons, as object-precision visibility computations for curved surfaces quickly become intractable because of the degree of the algebraic calculations required. Hidden line removal, where only the visible portions of the boundary segments are determined, without necessarily determining which face the segments bound, is treated only briefly (see Refs. 1 and 2 for more details) because, for raster displays, hidden line removal is subsumed by hidden surface removal. The rest of this section introduces two simple hidden feature removal algorithms. The second section discusses image-precision techniques that have become extremely important in practice. The third section then covers object-precision techniques. The fourth section covers hybrid techniques. The final section discusses recent advances in hidden feature removal algorithms.

If the polygons in the rendering database are known to be the boundaries of solid objects, only faces that face toward the viewer can be visible; other faces need not be rendered. Removing polygons using this criterion is known as *back-face culling.* Back-face culling is sufficient to produce correct renderings only for exceptional circumstances (a single, convex object). However, it is an extremely powerful technique for limiting the number of polygons that must be considered. Typically, polygons will be culled as early as possible in the rendering pipeline. To determine if a face is back-facing, we check the position of the viewpoint against the plane equation of the face and then test if the viewpoint lies in the appropriate halfspace.

If the rendering database is known to contain objects that are in layers that do not cross one another (e.g., Very Large-Scale Integration circuit masks), the layers can be rendered in order, with the bottom layer rendered first. Because the image of closer polygons overwrites the image of further polygons, the method resolves visibility, at the cost of *overrendering.* Overlap between polygons or priority cycles (see Fig. 1) in the rendering database will lead to incorrect results. A version of this algorithm, due to Newell et al. (3) and often known as the painter's algorithm, can produce correct renderings for arbitrary collections of polygons by detecting overlaps and subdividing the polygons involved; it is no longer widely used.

## IMAGE-PRECISION METHODS—THE *z*-BUFFER

The *z*-buffer (4) is the dominant image-precision method. The discussion of this algorithm assumes that parallel projection is used to project the scene onto the viewplane. (If perspective projection is desired, then we can perform a projective transformation on the 3-D scene and perform a parallel projection on the resulting distorted scene. This process is a standard component of the rendering pipeline. The appropriate transformation will take the focal point to infinity and will preserve relative depth, straight lines, and planes, so the algorithms described in this section produce the correct result for perspective viewing.) For simplicity, the discussion assumes that the viewplane lies on the $xy$-plane and the scene is projected along the $z$-direction. The $z$-coordinate corresponds to the distance from the viewing plane with higher $z$ being further away from the viewer.

The $z$-buffer requires a memory buffer—the $z$-buffer—that has one element per screen pixel. A $z$-value for screen pixels will be stored in this buffer, the size of whose elements may vary. Initially, all the entries in the $z$-buffer are set to the maximum $z$-value (corresponding to the back-clipping plane). To determine the visibility of a set of polygons, the polygons are scan-converted into the frame buffer.

During the scan-conversion process of a polygon, suppose that the pixel $(x, y)$ is being filled, that the value $z_o$ is stored at $(x, y)$ on the $z$-buffer, and that the current polygon has depth $z_n$ at $(x, y)$. If $z_n$ is greater than $z_o$, the current polygon is farther from the view plane than whatever is in the frame buffer already and must therefore be invisible at this pixel, and so nothing needs to be done. If $z_n$ is less than $z_o$, the current polygon is closer to the viewing plane than whatever was previously scan converted, and so should be written to the pixel at $(x, y)$. At this point, lighting calculations for the given pixel can proceed, and the result is written into the frame buffer at $(x, y)$. Because the frame buffer now contains the brightness of an object closer than $z_o$, the $z$-buffer is updated with the value $z_n$ at $(x, y)$.

Because the $z$-values are compared per pixel, the $z$-buffer produces the correct image regardless of the order in which the polygons are scan-converted. However, because the lighting calculations are done only for pixels whose $z$-value is smaller than that currently stored in the $z$-buffer, the order in which polygons are scan-converted can significantly affect the speed of the $z$-buffer algorithm. Typically, scan-converting polygons in a front-to-back order is significantly more efficient because overrendering will occur less frequently in a front-to-back order.

Underlying the $z$-buffer algorithm's popularity is the ease with which it is implemented (both in software and in hardware), the fact that it does not demand preprocessing of the rendering database, and its efficiency. Scan-line coherence can be used to update the $z$-value at each pixel efficiently. Suppose that the plane of the polygon is described by $Ax + By + Cz + D = 0$. Then

$$z = \frac{-Ax - By - D}{C}$$

(If $C = 0$, then the plane is projected as a line and, therefore, can be ignored.) Suppose that the polygon has depth $z$ at $(x, y)$. The depth $z'$ at $(x + 1, y)$ can be obtained by

$$z' = \frac{-A(x+1) - By - D}{C} = z - \frac{A}{C}$$

Therefore, succeeding depth values across a scan line are obtained from the preceding values with a single addition because the ratio $A/C$ is constant for each polygon. Given the depth value of the first pixel on the scan line, depth values of the succeeding pixels can be easily found. Also, given the first depth value of the current scan line, the first depth value of the next scan line can be found by a single addition, by using a similar reasoning.

The $z$-buffer algorithm is by far the most widely used visibility algorithm. Nonetheless, it has a number of significant disadvantages:

- Quantization error in the $z$-buffer can lead to annoying artifacts. As an extreme example, consider polygons $A$ and $B$ that are very close in depth and parallel and whose images overlap on the viewing plane. In one view, $A$'s depth and $B$'s depth will translate to different quantized values, and the correct polygon will appear in the image. If the view is moved slightly, the depths may translate to the same quantized values. In this case, it is not possible to determine which should lie in front, and some policy (e.g., always render most recent pixel) must be applied. Whether this policy will result in a correct image depends purely on chance factors in the structure of the rendering database. Therefore, it is possible to have a situation where moving the view backward and forward results in polygons flashing. Quantization error can also result in annoying artifacts where polygons interpenetrate. Current $z$-buffers typically have 24 bits per pixel to alleviate this difficulty.
- Overrendering can become a serious problem for large rendering databases. In collections of millions of polygons, the requirement that every polygon be scan-converted can lead to very slow rendering. This is most clearly inefficient when there is a structure to the rendering database; for example, in most building models, most of the polygons in the model are not visible from a given room.
- Aliasing and transparency are both poorly handled by the $z$-buffer algorithm because only one polygon can contribute to the brightness at each pixel. Increasing the resolution of the buffers can alleviate the aliasing problems at the cost of large $z$-buffers and interim frame buffers. In general, the $z$-buffer cannot render scenes containing mixtures of transparent and opaque objects correctly without using additional memory and incurring rendering overhead (5). The $A$-buffer (6) algorithm is an elaboration on the $z$-buffer, which accumulates a list of polygon fragments that affect the visible brightness of each pixel (rather than simply storing intensities). The list is then processed to determine the final pixel intensity. With an appropriate set of rules for insertion, the $A$-buffer can render mixed translucent and opaque surfaces.

The most important image-space visibility algorithm besides the $z$-buffer is ray tracing (7–9). For each pixel on the image plane, an "eye ray" is fired. This eye ray is intersected with every object, and the closest intersection point is determined. Then, the intensity value associated with the closest intersection point is written into the pixel. This algorithm can be made faster by making intersection calculation more efficient (2).

## OBJECT-PRECISION METHODS

Overrendering in the $z$-buffer makes it natural to consider using an object-precision method to cull polygons that could not possibly be visible. Even though the binary space partition tree (or BSP tree) overrenders as badly as the $z$-buffer, it is still popular in practice. In restricted geometries, a cell decomposition method can be extremely efficient. Finally, methods based on computational geometry achieve optimal complexity in various ways, although most of them are not used in practice.

### Binary Space-Partitioning Tree

The BSP tree (10,11) is a popular method for generating rendering order among objects. The visibility among objects is resolved by rendering objects in back-to-front order, like the painter's algorithm. The BSP tree splits all the polygons in the preprocess, so that it is possible to generate a rendering order among the polygons from any viewing position. The BSP tree is particularly useful when the viewpoint changes, and the objects stay at fixed positions.

Given a set of polygons, the corresponding BSP is constructed as follows (Fig. 2). (Assume that a surface normal is associated with each polygon, so that the front and back side of the polygon can be defined.) First, a splitting polygon is selected. The algorithm works correctly no matter which polygon is selected as the splitting polygon. The plane associated with the splitting polygon divides the environment into two half-spaces, namely the front and the back half-space. Then, each remaining polygon either lies entirely in the front half-space or lies entirely in the back half-space or lies in both half-spaces. Those polygons lying in both half-spaces are split along the splitting plane, so that the polygon fragments can be assigned to either the front or the back half-space [Fig. 2(a)]. The root node of the BSP tree is associated with this splitting polygon, and the rest of the BSP is constructed recursively. For example, given the set of polygons lying in the front half-space, a splitting polygon is chosen, and this polygon is associated with the front child of the root node [Fig. 2(b)]. All the polygons in the front half-space are divided into two sets with respect to this splitting polygon. The BSP tree is completed when there is only one polygon associated with each child node [Fig. 2(c)].

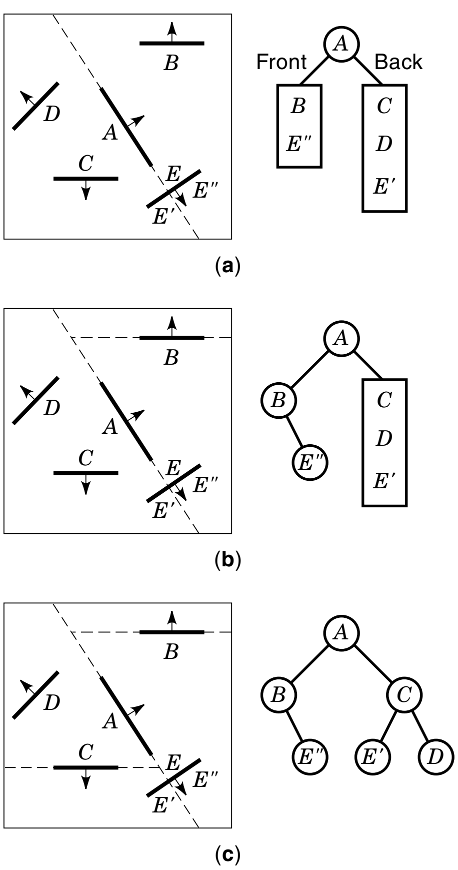



**Figure 2.** Building a BSP tree for this example set, which consists of vertical polygons viewed from above. (a) polygon $A$ is chosen as the splitting polygon associated with the root node; (b) polygon $B$ is chosen as the splitting polygon for the front half-space of $A$; (c) polygon $C$ is chosen as the splitting polygon for the back half-space of $A$. The BSP tree is completed.

Note that a BSP tree can be constructed for any set of objects, polygonal or otherwise, by choosing the appropriate set of splitting planes. It is particularly easy to build a BSP tree for a set of polygons, however, because the splitting planes can be conveniently chosen to lie along the polygons.

Given a BSP tree, the rendering order is produced as follows. Suppose that the viewpoint lies on the front half-space of the root polygon. Then, none of the polygons lying within the back half-space can obscure the polygons lying on the front half-space. Therefore, the rendering order is as follows: the set of polygons lying in the back half-space is rendered, then the splitting polygon is rendered, and then the polygons lying in the front half-space is rendered. The visibility among the polygons is resolved during scan-conversion, like painter's algorithm. If the viewpoint lies inside the back half-space, then the polygons lying on the front half-space is rendered first, the splitting polygon is rendered next, and finally the polygons lying on the back half-space is rendered. If the viewpoint lies exactly on the splitting plane, then the rendering order does not matter. Within each half-space, the rendering order is computed recursively in the exact same way. Thus, given a viewpoint, the BSP tree can be walked in-order, depending on the half-space in which the viewpoint lies, to produce a rendering order among the polygons.

Which polygon is selected to serve as the root of each subtree can have a significant impact on the algorithm's performance. Ideally, the polygon selected should cause the fewest splits among all its descendants. The algorithm outlined here potentially produces $O(n^3)$ faces (10). Paterson and Yao (12) show how to choose the splitting planes optimally to produce $O(n^2)$ faces in $O(n^3)$ time. A heuristic that produces an approximation to the best case is described in Ref. 10.

### Cell Subdivision in a 2-D World

Object precision visibility in a 2-D environment is much easier than the general 3-D case; a 2-D environment is often a useful approximation of the 3-D case in practice (e.g., mazes and floors of buildings can be approximated as 2-D environments). This section focuses on a maze consisting of opaque walls, with all the walls rising up to the ceiling. From a typical viewpoint, only a fraction of the environment will be visible. The main task is to determine the visible portions of the maze and ignore the rest. To do this, we will decompose the environment into a set of convex cells and then walk this cell structure in breadth-first order to enumerate the visible objects.

The environment can be decomposed into a set of convex cells using a technique called trapezoidal decomposition (13). In Fig. 3, the line segments correspond to opaque walls, and the dashed line segments correspond to transparent walls constructed through trapezoidal decomposition. To construct the trapezoidal cells, extend transparent walls vertically from each vertex until an opaque wall is reached. The result is a set of convex cells, where each cell is a trapezoid (or a triangle, in a degenerate case). We define a neighbor of a cell to be the cells that share a transparent edge with the given cell.

We can define a ray-casting operation that extends a ray from the viewpoint, through some other given point—which will always be a cell vertex and which we shall call the fulcrum—until the ray hits an opaque wall. Every cell that the ray passes through is marked so that for any cell it is possible to tell which rays pass through the cell. The ray separates the visible portion of the environment from the invisible portion. Because the ray affects visibility only between the fulcrum and the far end, the ray is recorded only in cells that lie in this span. With each ray, we make a record of which side of the ray is visible.

The algorithm for enumerating visible objects by traversing the cell structure is illustrated by the example in Fig. 4. The initial configuration before any rays were cast is shown in Fig. 4(a). The algorithm involves first marking the cell in which the viewpoint lies (cell *F* in the example) as processed. For each vertex that lies in this cell ($v4$ and $v5$ in the example), cast rays through these vertices and record their presence in every affected cell. Finally, add every neighbor of this cell to the *TODO* queue [Fig. 4(b)]. Pseudocode for the rest of the algorithm follows:

while (*TODO* queue is not empty)

1. Let *Cell* be the first cell of the *TODO* queue.
2. Cast rays through the *unprocessed, visible* vertices of *Cell.*

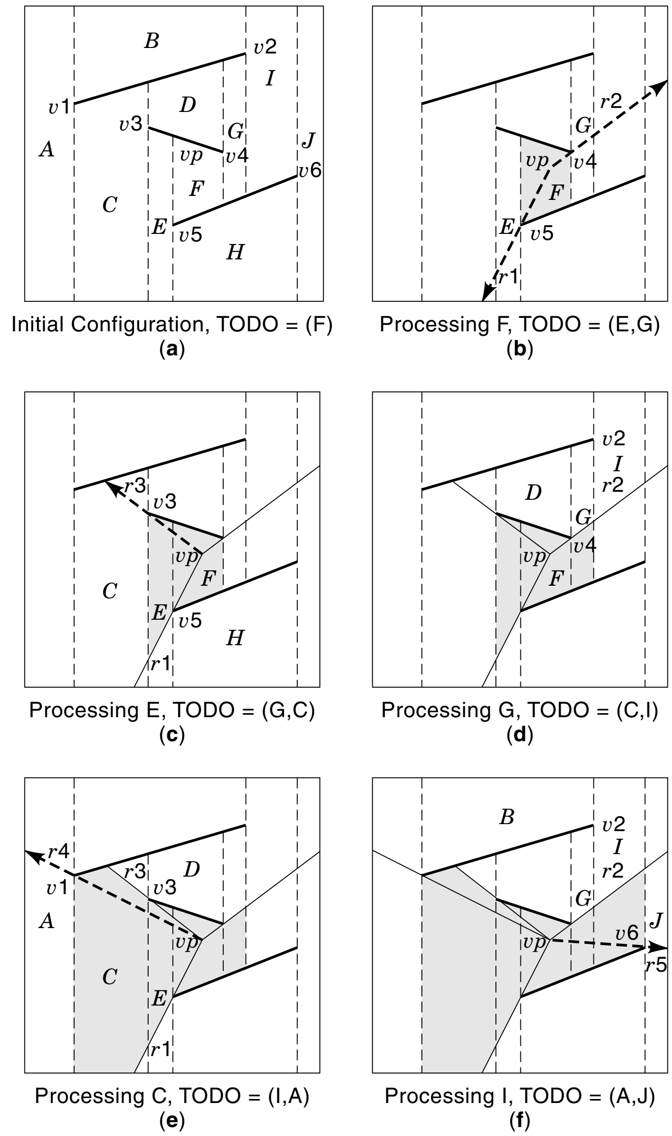



**Figure 4.** The cell structure is searched to enumerate the visible faces. Each stage is explained in the text.

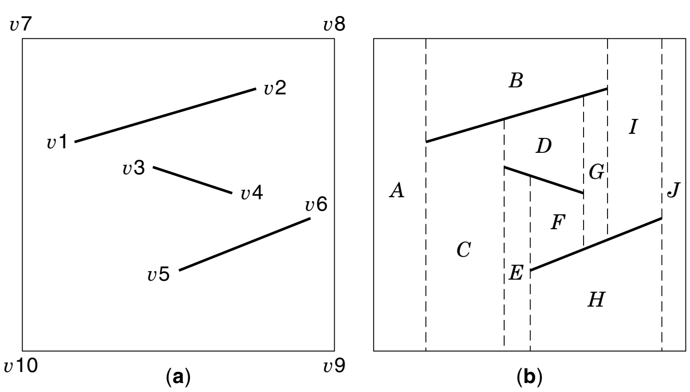



**Figure 3.** An example of trapezoidal decomposition. (a) A 2-D environment consisting of line segments. (b) Trapezoidal decomposition of this environment. The vertices are labeled {$v1$, . . ., $v6$}, and the cells are labeled {*A*, . . ., *J*}.

3. Add the *unprocessed* neighbors of *Cell,* which share a *visible* transparent edge with *Cell,* to the *TODO* queue.
4. Mark *Cell* as processed.

In steps 2 and 3, to determine whether a vertex or an edge is visible or not, it is tested against each ray that passes through the cell. Fig. 4(c)–(f) illustrates this algorithm.

- In Fig. 4(c), *Cell* is *E*, which contains $v3$ and $v5$.
  - $v5$ is ignored because it has already been processed.
  - $v3$ is tested against the ray that passes through *E*, namely $r1$, and is determined to be visible. Therefore a ray is cast through $v3$.

  *E* has three neighbors, namely, *F*, *H*, and *C*.
  - *F* is ignored because it has already been processed.
  - *H* is ignored because the transparent edge shared between *E* and *H* lies on the invisible side of $r1$.

- $C$ is added to *TODO* queue because the transparent edge shared between $C$ and $E$ partially lies on the visible side of $r1$.

Finally, $E$ is marked as processed.

The reader is encouraged to verify the remaining steps of the algorithm. (They are illustrated in Fig. 4(d)–(f).)

When this process finishes, the final set of rays are shown in Fig. 5. These rays are sorted either in clockwise or counterclockwise order, and the visible segments are determined between adjacent rays. These visible segments are then scan-converted into the frame buffer. This process is very deficient because most of the invisible cells are never visited. Versions of this technique are commonly used in video games and other simulation-type interactive programs. A 3-D analog of this algorithm is described in Ref. 13.

### Hidden Surface Removal in Computational Geometry

The Weiler-Atherton algorithm (14) is a good example of an early object-precision hidden surface removal algorithm. Given a set of polygons, this algorithm outputs all the visible fragments as lists of vertices. Before we discuss the algorithm, let us define the clipping operation which is used extensively in the algorithm. If polygon $Q$ is clipped against polygon $P$ (Fig. 6), $Q$ is divided into fragments and these fragments are collected into inside and outside lists. The fragments on the inside list lie inside $P$ when the fragments and $P$ are projected onto the viewing plane. The fragments on the outside list lie outside $P$ when projected onto the viewing plane. The algorithm works as follows. First, the polygons are sorted in $z$ (e.g., by the nearest $z$-coordinate). Let $P$ be the closest polygon, by this criterion. Then, every other polygon is clipped against $P$. All the polygons on the inside list that are behind the clip polygon are invisible and, therefore, deleted. If any polygon on the inside list is closer to the viewpoint than the clip polygon, the algorithm recurses with this polygon as the clip polygon. When the recursive call returns, the polygons on the inside list are displayed, and the outside polygons are processed. One of the advantages of the Weiler-Atherton algorithm is that it can be used to generate shadows (15). To generate shadows, the viewpoint is made to coincide with the point light source, and the visible fragments are generated from this point of view. These fragments correspond to the lit portions of the polygons. After these lit fragments are generated, they are used as surface-detail polygons.

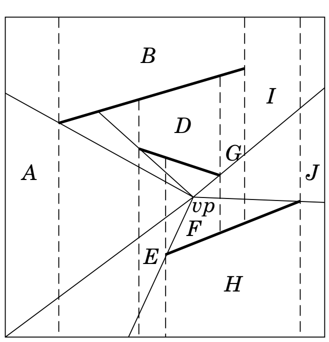



**Figure 5.** The ray-casting process results in an enumeration of the visible portions of the segments. Each visible portion, together with the viewpoint, forms a triangle as shown previously.

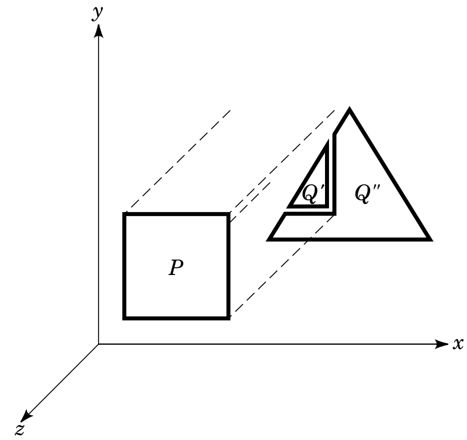



**Figure 6.** In Weiler-Atherton algorithm, polygon $Q$ is clipped against polygon $P$, so that each clipped polygon ($Q'$ and $Q''$) either lies completely inside $P$ or outside $P$ when projected onto the viewing plane.

One difficulty with the Weiler-Atherton algorithm is the number of polygon fragments it can generate. Even though the Weiler-Atherton algorithm is not optimal, this difficulty is intrinsic to 3-D visibility. The computational geometry community has developed a body of work on the space- and time-complexity of visibility, usually defined in terms of constructing a *visibility map,*, which is a subdivision of the viewing plane into maximal connected regions, in each of which either a single face or nothing is visible (Fig. 7). The complexity of the algorithm is mainly characterized by three variables: the size of the input $n$, which is the number of distinct boundary edges in the input set (equivalently, $n$ may measure the number of distinct vertices or faces in the input set), the number of intersections in the projection of the input set $k$ (which includes all intersections, not just visible ones), and the size of the output $d$, which is the number of distinct boundary edges in the visibility map (equivalently, $d$ may measure the number of distinct vertices or faces in the visibility map). Notice that given $n$ convex polygons (input size

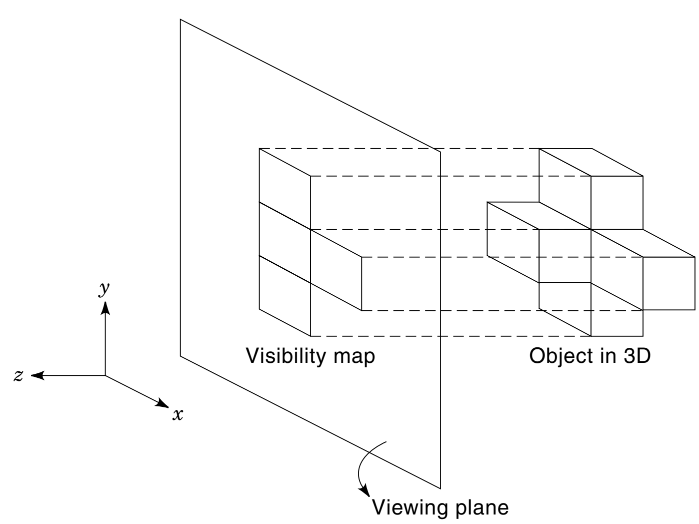



**Figure 7.** A visibility map is constructed on the viewing plane by performing a parallel projection of a 3-D object. The viewpoint is assumed to be at $(0, 0, \infty)$.

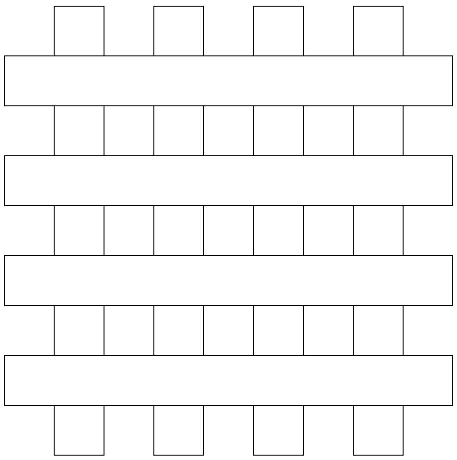


**Figure 8.** This visibility map has output complexity $\Omega(n^2)$, where $n$ corresponds to the number of boundary edges in the input set. The output complexity measures the number of boundary edges in the visibility map.

$\Theta(n)$), the visibility map may have output size $\Omega(n^2)$ (Fig. 8). Therefore, every hidden surface removal algorithm must have $\Omega(n^2)$ worst case lower bound (for a more extensive discussion, see for example, Ref. 16).

Constructing algorithms that are output-sensitive (i.e., running time depends at least partly on $d$) or are optimal in time and space requirements has been a major topic in computational geometry. An extensive review of recent results appears in Ref. 17. Although these algorithms have low time- and space-complexity, only a small number of them are used in practice, for the following reasons:

- Many of the hidden surface removal algorithms in this section are complicated and difficult to implement. Also, even if these algorithms have low time- and space-complexity, the complexity measurements may hide a huge constant coefficient.
- In practice, some visibility queries are more easily answered in image-space, as opposed to object-space. To render a tree with thousands of leaves, it would be impractical to construct the visibility map using an object-space algorithm, because the output complexity is extremely high. An image-space algorithm like the $z$-buffer may prove to be a more practical solution.

## HYBRID METHODS

For many situations, combinations of different visibility algorithms work well in practice. The intuition is that a sophisticated technique is first used to cull most of the invisible objects. Among the remaining objects, simple techniques can be used to determine the exact visibility. For example, for a walk-through of a building model, techniques outlined in this section can be used to determine the set of potentially visible objects quickly. Given this set, a $z$-buffer can be used to determine exactly which objects are visible. Also, for a complicated detail object (e.g., chandelier in the middle of a room), the first pass would determine whether the chandelier is potentially visible. If the first pass determined that the chandelier is potentially visible, a $z$-buffer can be used to render the visible portion of the chandelier effectively.

### The Hierarchical $z$-Buffer

The hierarchical $z$-buffer (18) algorithm uses a hybrid object- and image-precision approach to improve the efficiency of the $z$-buffer algorithm. There are two data structures: the object-space octree [Fig. 9(a)], and the image-space $z$-pyramid [Fig. 9(b)]. As a preprocess, objects are embedded in the octree structure, so that each object is associated with the smallest enclosing octree cube [Fig. 9(a)]. If an octree cube is invisible, every object associated with the cube must also be invisible, and these objects may be culled. The octree is traversed and the contents of the octree nodes are rendered into the framebuffer as follows:

1. Determine if the root cube of the octree is inside the viewing frustum. If it is outside the viewing frustum, then the entire set of objects is invisible.
2. Determine if the root cube is (partially) visible by testing each front-facing face against the hierarchical $z$-buffer. The hierarchical $z$-buffer is explained later in this section.
3. Scan-convert the objects associated with the root node if the root cube is determined to be visible. Otherwise, we are finished.
4. Recursively process the children of the root node, in the front-to-back order. Notice that it is trivial to determine the front-to-back order of the octree children nodes, by looking at the octant in which the viewpoint lies.

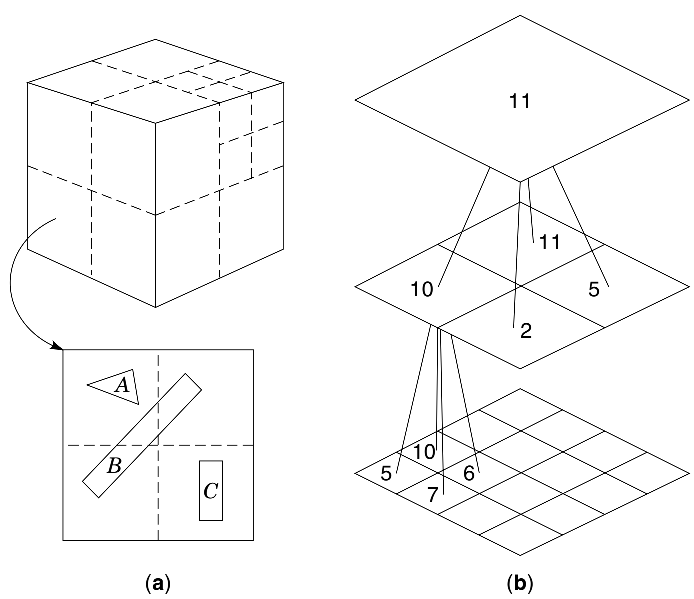



**Figure 9.** Data structures used in the hierarchical $z$-buffer. (a) The objects in the environment are embedded in an object-space octree. Objects $A$ and $C$ are associated with smaller cubes, whereas object $B$ is associated with the larger cube. (b) The image-space $z$-pyramid, which corresponds to a square region on the screen. The farthest depth value among four neighbors is passed up to the higher level. Because the highest entry has depth value 11, every other entry must have depth value 11 or less. Therefore, an object with minimum depth-value greater than 11 would not be rendered inside this region.

In order to cull the octree cubes, a $z$-pyramid is used. The basic idea of the $z$-pyramid is to use the original $z$-buffer as the finest level in the pyramid and then combine four $z$-values at each level into one $z$-value at the next coarser level. Given the four $z$-values, the farthest $z$-value among the four entries is passed up to the entry on the higher level. At the top level, there is a single $z$-value, which is the farthest $z$-value from the observer in the whole image. In the beginning, all the entries on all the levels are initialized to the maximum $z$-value.

Maintaining the $z$-pyramid is simple. Every time the $z$-buffer is updated, the new $z$-value is propagated through to coarser levels of the pyramid. As soon as this process reaches a level where the new $z$-value is no longer the farthest $z$-value from the viewpoint, the propagation can stop. Determining whether a polygon is visible or not works as follows:

1. Find the finest-level entry of the pyramid whose corresponding image region covers the screen-space bounding box of the polygon.
2. Compare the $z$-value at the entry to the nearest $z$-value of the polygon. The $z$-value at an entry indicates that every pixel in the corresponding region is no farther than this value. If the nearest $z$-value of the polygon is farther away than the $z$-value at the entry, then the polygon is hidden.
3. If the previous step did not cull the polygon, then recurse down to the next finer level and attempt to prove that the polygon is invisible in each of the quadrants it intersects. (In each quadrant, the new nearest $z$-value of the polygon can be calculated, or the old value can be reused. Reusing the old value provides a conservative estimate.)
4. If the finest level of the pyramid is reached, and there is at least one visible pixel, then the polygon is determined to be visible.

When the $z$-pyramid determines that an octree cube is visible, the objects associated with the cube are scan-converted into the $z$-buffer, and the $z$-pyramid is updated. Thus, the octree is walked in the front-to-back order (the objects are rendered in roughly front-to-back order), and the $z$-pyramid is used to determine whether the current cube is visible or not. This approach has the advantage that large nearby objects will generally be rendered first and that whole sections of the octree may be culled with a single test. For a complex scene with very high depth complexity, Greene, Kass, and Miller (18) report that the hierarchical $z$-buffer achieves orders of magnitude speedup over the traditional $z$-buffer. For simple scenes with low depth complexity, hierarchical $z$-buffer performs slightly worse than the traditional $z$-buffer because of the overhead of maintaining the $z$-pyramid and performing visibility tests on octree cubes. Meagher (19) describes a similar algorithm, which precedes the work of Greene et al. (18). In this algorithm, an image-space quadtree is used to render the octree efficiently.

### Cell Decomposition in Architectural Models

An architectural model can be seen as a combination of large occluders and geometric detail. The large occluders form a natural collection of cells separated by boundaries such as the walls, doors, floors, and ceilings; these cells, which would usually correspond to rooms or corridors, are typically quite simple in shape. The detail consists of such things as furniture, books, and telephones, which can be associated with individual cells. It is natural to attempt to perform object-precision visibility on the large occluders and then use a $z$-buffer to render detail that could be visible, thereby culling large numbers of polygons without incurring high costs.

Teller (20) offers an attractive approach to hybrid visibility, which uses a conservative algorithm—one that will not omit a visible polygon, but may not cull all invisible polygons—to determine visibility among the cells. A cell boundary consists of occluders (opaque portion of the boundary such as walls) and a collection of convex portals (transparent portion of the boundary such as doors or windows). There are many possible subdivisions of the same model; a heuristic to obtain good subdivisions appears in Ref. 20. The subdivision yields a cell adjacency graph, where a vertex corresponds to a cell and an edge corresponds to a portal. For example, if cells $A$ and $B$ share a portal, then the vertices corresponding to $A$ and $B$ are connected by an edge. Consider a *generalized observer,* an observer who is free to move anywhere inside a given cell and look in any direction. Given a generalized observer in a cell, we want to determine which set of cells is visible to the observer, wherever the observer is. (This set of cells is called the *potentially visible set.*) Determining which cells are potentially visible requires determining whether any ray can be cast through a sequence of portals; the process is described in some detail in the next section.

After it is known which cells are potentially visible from each given cell, relatively efficient rendering is simple. We render every polygon in the cell containing the viewpoint, every polygon in every cell that is potentially visible from this cell, and all the detail associated with these cells, using a $z$-buffer to determine the exact visibility. The resulting algorithm is accurate and relatively efficient and is not particularly difficult to implement. The main drawbacks are that there is no principled mechanism for distinguishing between detail and cells and that there is no principled mechanism for cell decomposition. As a result, it can be difficult to apply this technique to geometries that do not offer an immediate cell decomposition. These difficulties may be finessed in the modeling process by building modeling tools that encourage a modeler to help distinguish between detail and large occluders and offer hints about the cell decomposition. At present, this is the algorithm of choice for architectural models.

## CELL DECOMPOSITION IN ARCHITECTURAL MODELS

### Determining Potentially Visible Sets in 2-D

If a generalized observer in cell $A$ can see into cell $B$, there must exist a *stabbing line* from cell $A$ to cell $B$ through a particular sequence of portals. Consider the two-dimensional case, where portals and occluders correspond to line segments. For cells $A$ and $B$ to be mutually visible, there must exist a sequence of edges in the cell adjacency graph that leads from (the vertex corresponding to cell) $A$ to (the vertex corresponding to cell) $B$. We can orient each portal in the sequence, so that the stabbing line must cross each portal in a particular direction.

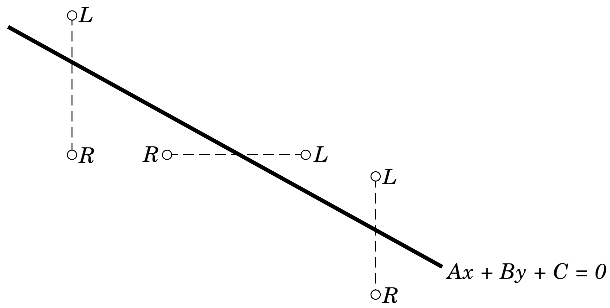



**Figure 10.** An example of a stabbing line in 2-D. Each portal is oriented (i.e., the stabbing line must cross the portal in such a way that the vertex labeled $L$ must lie on the left side of the stabbing line), and the vertex labeled $R$ must lie on the right side. A stabbing line exists through the portal sequence because the vertices labeled $L$ are linearly separable [i.e., satisfy Eq. (1)] from the vertices labeled $R$.

Depending on the direction in which the portal is crossed, we can determine the left and right vertices of the line segment corresponding to the portal. Given a portal sequence, we can separate the set of vertices into sets $L$ and $R$. A line stabs this portal sequence if and only if it separates the point sets $L$ and $R$, that is, if and only if there exists a line $Ax + By + C = 0$ such that (Fig. 10)

$$\begin{aligned} Ax + By + C &\geq 0, \quad \forall (x, y) \in L \\ Ax + By + C &\leq 0, \quad \forall (x, y) \in R \end{aligned} \tag{1}$$

Standard algorithms from linear programming can be used to determine whether a feasible point exists for the set of inequalities in Eq. (1). Enumerating the potentially visible set from a given cell now becomes a matter of depth first search. Given a portal sequence, we test to see if the next cell is visible by adding a portal to the current portal sequence. Once a cell is determined to be invisible, its "children" need not be explored.

### Potentially Visible Sets in 3-D

To solve the three-dimensional case, we must be able to represent lines in three dimensions. One way to represent lines in 3-D is to use the Plücker coordinates (21,22). Suppose we want to represent a directed line $l$ which passes through points $x$ and $y$ in this order. Using homogeneous coordinates, the points can be represented as $x = (x_0, x_1, x_2, x_3)$ and $y = (y_0, y_1, y_2, y_3)$. Let us define the six Plücker coordinates as $(p_{01}, p_{02}, p_{03}, p_{12}, p_{23}, p_{31})$, where $p_{ij} = x_i y_j - x_j y_i$. Since the points are described by homogeneous coordinates, scaling each coordinate by a constant will describe the same point, but each Plücker coordinate will be scaled by the same constant. Therefore, the Plücker coordinates are unique up to a factor of scale, and they describe a point in 5-D in homogeneous coordinates.

Each Plücker coordinate corresponds to a $2 \times 2$ minor of the matrix

$$\begin{pmatrix} x_0 & x_1 & x_2 & x_3 \\ y_0 & y_1 & y_2 & y_3 \end{pmatrix} \tag{2}$$

We can verify that the Plücker coordinates remain the same, no matter which two points are used to describe the line. Notice that taking a linear combination of the homogeneous coordinates of $x$ and $y$ will produce a point $(\alpha x_0 + \beta y_0, \ldots, \alpha x_3 + \beta y_3)$ that lies on the line that contains $x$ and $y$. In matrix form, this is equivalent to multiplying the matrix in Eq. (2) by a $2 \times 2$ matrix on the left

$$\begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix} \begin{pmatrix} x_0 & x_1 & x_2 & x_3 \\ y_0 & y_1 & y_2 & y_3 \end{pmatrix} \tag{3}$$

The reader may verify that the ratios among the $2 \times 2$ minors of the matrix in Eq. (3) remain invariant. Therefore, the Plücker coordinates remain the same no matter which two points are used to describe the line.

If $P$ and $Q$ are two directed lines and if $p_{ij}$, $q_{ij}$ are their corresponding Plücker coordinates, the relation $side(P, Q)$ can be defined as the permuted inner product

$$\begin{aligned} &side(P, Q) \\ &\quad = p_{01}q_{23} + p_{23}q_{01} + p_{02}q_{31} + p_{31}q_{02} + p_{03}q_{12} + p_{12}q_{03} \end{aligned} \tag{4}$$

This sidedness relation can be interpreted geometrically with the right-hand rule (Fig. 11). If the thumb of one's right hand is directed along $P$, then $side(P, Q)$ is positive if $Q$ goes by $P$ along one's fingers. If $Q$ goes by $P$ against one's fingers, then $side(P, Q)$ is negative. If $P$ and $Q$ are incident, then $side(P, Q) = 0$.

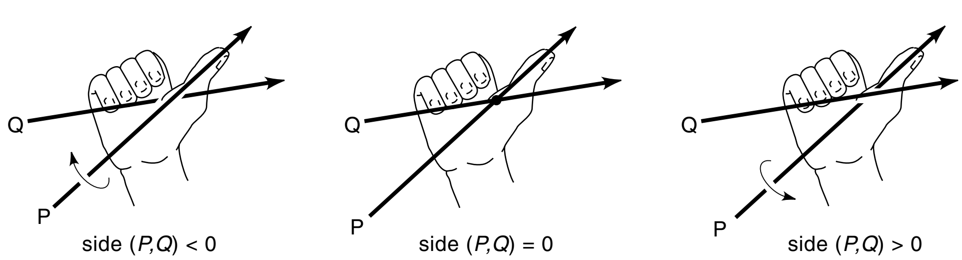
Notice that every line must be incident upon itself. Therefore, every real line $P$ in 3-D must satisfy $side(P, P) = 0$. The 4-D hypersurface that satisfies the previous equation is called the Klein quadric. Notice that not every homogeneous six-tuple corresponds to a real line in 3-D; only points on the Klein quadric do.

Suppose an oriented portal has $n$ edges. Then we can associate a directed line $e_i$ with each edge so that it is oriented clockwise, viewed along a stabbing line. Then, for a directed line $S$ to stab the portal (Fig. 12), $S$ must satisfy

$$side(e_i, S) \geq 0, i \in 1, \ldots, 4 \tag{4}$$

If such $S$ exists and $side(S, S) = 0$, then $S$ stabs the portal. For a stabbing line $S$ to stab a portal sequence, $S$ must satisfy

**Figure 11.** The right-hand rule applied to $side(a, b)$. The curved arrow indicates the direction in which $b$ goes by $a$ (either clockwise, or counterclockwise, as viewed along $a$). $side(a, b)$ is positive, negative, or zero, depending on this direction.



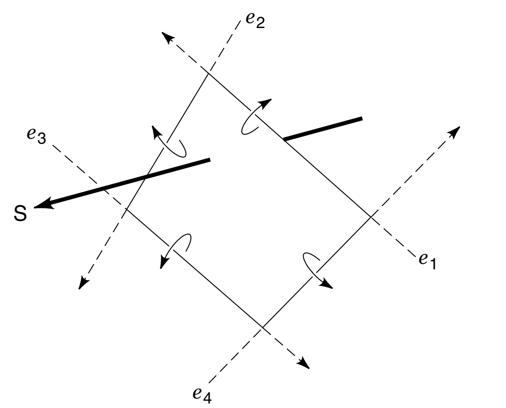



**Figure 12.** An example of a stabbing line in 3-D. A stabbing line must pass to the same side of each $e_i$ [i.e., satisfy Eq. (2)].

Eq. (4) for each portal, and $side(S, S) = 0$. In (20), Teller describes an algorithm that determines whether a portal sequence admits a stabbing line by associating an oriented hyperplane with each $e_i$, by forming a convex polytope $\cap_i h_i$, and by checking whether this polytope intersects the Klein quadric.

### Alternative Approaches to Computing the Potentially Visible Sets

An early attempt at calculating the potentially visible set is presented in Ref. 23 where discrete sampling and shadow volumes [10] are used to compute the set of cells visible from a portal polygon. This method only offers an approximate solution, since discrete sampling underestimates the potentially visible set, while using shadow volume overestimates it.

An algorithm which generates the potentially visible set on the fly is presented in Ref. 24 where every time the viewer changes position, the cell adjacency graph is traversed and the potentially visible set is computed per viewpoint. This algorithm is based on the observation that in order for a cell to be visible, the portal leading to that cell must be visible. To determine whether a portal is visible through a portal sequence, we associate a screen-space axial bounding box with each portal. If the intersection of these bounding boxes is nonempty, we can conservatively estimate that the portal is visible through the portal sequence. In this manner, determining the potentially visible set reduces to depth-first search on the cell adjacency graph. One advantage of computing the potentially visible set on the fly is that walls and portals can be interactively modified, and the visibility algorithm requires no off-line processing to respond to these changes.

## RECENT ADVANCES IN HIDDEN FEATURE REMOVAL

### The Visibility Complex

As the previous section indicates, lines are the basic currency of visibility. For example, object $A$ can see object $B$, if and only if there exists a stabbing line from $A$ to $B$. One way to reason about lines is to think of them as if they are points. The process of associating a point to a line is called *dualization*. A region in the dual space corresponds to a set of lines.

There are several ways to associate a point with a line (e.g., Plücker coordinates associate a point in 5-D with a line in 3-D. The readers are referred to Refs. 25 and 26 for additional examples of dualization schemes.) For illustration purposes, this article will use the following dualization of lines in 2-D (this representation is similar to the dualization used in Ref. 25). Given a directed line $L$, a vector $u$ that starts from the origin and meets $L$ perpendicularly is constructed (Fig. 13). Let $\theta$ be the angle formed by $u$ and the $x$-axis and let $d$ be the directed distance between $L$ and the origin. $d$ is positive if $u \times L$ is positive, as in Fig. 13. Otherwise, $d$ is negative. The directed line $L$ is associated with the point $(\theta, d)$ in the dual space.

We illustrate the visibility complex with two examples. Figure 14(a) shows an environment that consists of one object, $O_1$. Given a directed line, there are infinitely many rays that are collinear with the directed line and that point in the same direction as the directed line. Now, we want to divide the set of rays according to which object the ray "sees." For example, in Fig. 14(a), all the rays that are associated with $L_1$ (e.g., $r_1$ and $r_2$) symbolically see the "blue sky." When we consider $L_2$, however, some of the rays associated with $L_2$ see $O_1$ (e.g., $r_3$), whereas the rest of the rays see the blue sky (e.g., $r_4$).

Given a set of rays that see the same object, we can associate with it a "sheet" in the dual space. For example, $r_3$ and $r_4$ correspond to the same point in the dual space (because they have the same $\theta$ and $d$ values), but they belong to different sheets, because they see different objects. $r_1$ and $r_2$ correspond to a point on the same sheet, because they see the same object. Now, consider Fig. 14(b). The curved region in the middle (which is bounded by two cosine curves) corresponds to the set of lines that intersect $O_1$. There are two sheets associated with this region: the rays associated with one sheet see the blue sky, whereas the rays associated with the other sheet see $O_1$. All the points outside this curved region correspond to one sheet, whose rays see the blue sky. This data structure is called the *visibility complex*. Figure 14(b) shows the cross section of the visibility complex.

Figure 15(c) shows an environment with two objects, and Fig. 15(a) shows the corresponding visibility complex. Notice

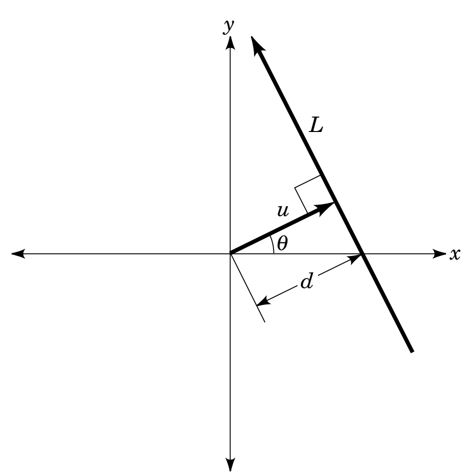



**Figure 13.** Dualization of a ray. A directed line $L$ is associated with the point $(\theta, d)$ in the dual space. $\theta$ is the angle between $u$ and the $x$-axis, and $d$ is the directed distance from the origin to $L$.

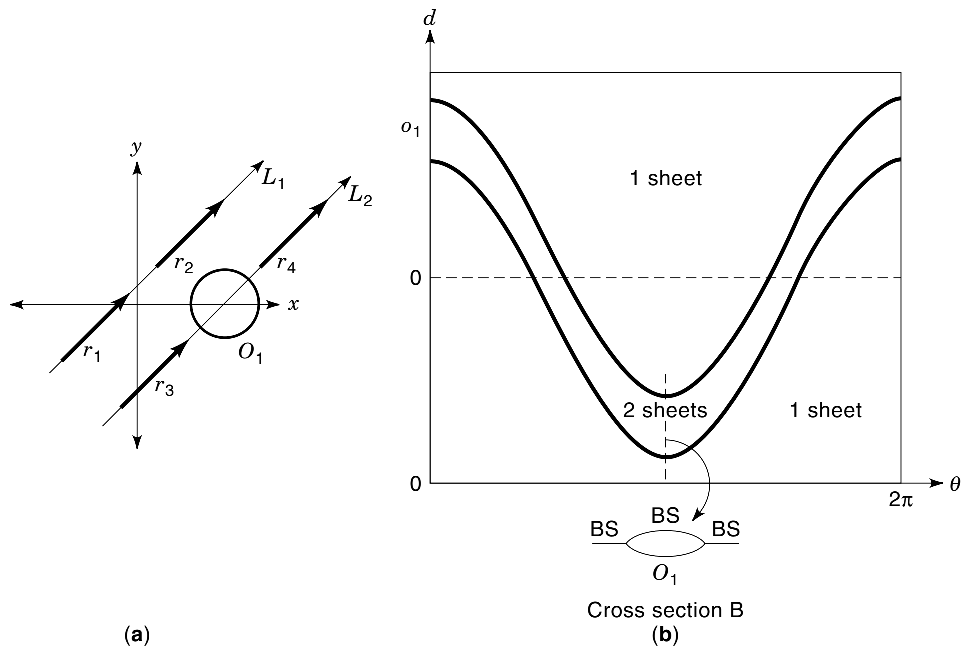



**Figure 14.** (a) An environment with one object and (b) the corresponding visibility complex. $r1$ and $r2$ share the same $\theta$ and $d$ values and, therefore, correspond to the same $(\theta, d)$ point on the visibility complex. $r1$ and $r2$ see the same object, so they correspond to the same sheet. $r3$ and $r4$, however, see different objects and therefore correspond to different sheets. On the visibility complex shown in (b), $r3$ belongs to the sheet labeled $O_1$ and $r4$ belongs to the sheet labeled BS.

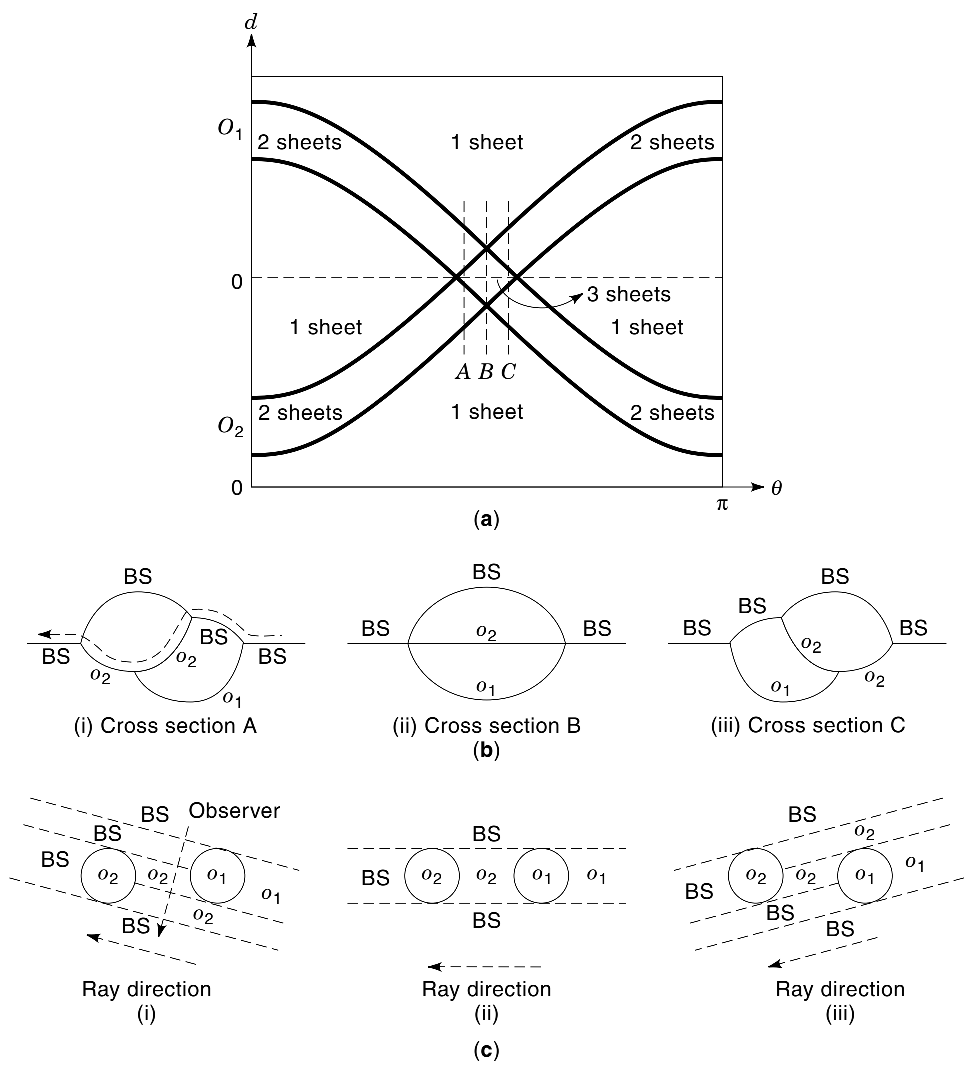



**Figure 15.** (a) The visibility complex of an environment with two objects, shown in (c). (b) Cross sections of the visibility complex at different $\theta$-values. A cross section at a particular $\theta$-value corresponds to a set of rays with the same direction, but different $d$ values. (c) The regions correspond to sets of rays with the same view (e.g., every ray starting from a region labeled $O_1$ sees $O_1$, along the given ray direction). You can easily verify that a region in (c) corresponds to a sheet in the cross section. In (c)(i), suppose that an observer moves along the arrow shown, while looking along the ray direction. On the visibility complex, this corresponds to tracing along the trajectory shown in (b)(i). Thus, given how an observer changes position and viewing direction over time, the corresponding view is computed by "walking" the visibility complex.

that adding a new object corresponds to introducing a new sheet in the visibility complex. Additional layers are added to some regions when a new sheet is added. The following algorithm computes the view of a user who looks around the environment. Sweeping the viewing ray around a fixed point traces a (cosine) curve in the dual space. To compute the view, the sheet corresponding to the starting point of the curve is determined, and the curve is traced on this sheet. When the curve crosses an edge (where two sheets join), the algorithm determines which sheet to follow by determining which corresponding object the current ray sees and traces the curve on this sheet. This process is continued until the final point of the curve is reached. As the user changes position, a new curve is traced in the dual space, and the algorithm begins again. Figure 15(b)(i) illustrates a simple example of how the visibility complex is "walked" to compute a view. Suppose that an observer walks along the indicated path shown in Fig. 15(c)(i), while looking along the given ray direction. The corresponding path on the visibility complex is shown in Fig. 15(b) to (i). The sheets encountered along this path correspond to the set of objects seen by the observer.

Building the visibility complex follows the incremental algorithm suggested by the example; details appear in Ref. 26, and Ref. 25 discusses the visibility complex for three-dimensional environments. Both works use a cumbersome parameterization of lines. A simpler parametrization could be achieved in 2-D by using the projective dual space (21) and in 3-D using Plücker coordinates. These parametrizations offer the advantage that a "sheet" will correspond to a polygon or a polytope in the dual space, both of which are easy to manipulate. The visibility complex approach offers numerous attractions:

- The rendering cost is output-sensitive. For example, rendering a room in a building would involve investigating only those elements of the visibility complex that correspond to visible polygons. This means that, even though the visibility complex cannot help distinguish between detail and large occluders, it avoids difficulties with defining cells.
- The visibility complex is particularly well suited to integration with modeling tools. Ideally, a modeler would create a representation that aided visibility calculations at the same time as creating the model. The incremental nature of the algorithms for constructing the visibility complex is particularly attractive here.
- Recent work (27,28) builds representations of objects by sampling the radiance along a set of rays. This representation is particularly suitable for very complicated real objects (e.g., a furry toy), which cannot be represented with current geometrical and photometric modeling techniques. Line representations of 3-D geometry interact particularly well with this object representation.

Overall, the visibility complex offers an effective method of calculating exact visibility in a very large environment. Moreover, the visibility complex offers a principled approach to hidden feature removal since it obviates the need to define an arbitrary cell structure over the input set (as required by the techniques that use potentially visible sets).

## Other Recent Approaches

One of the themes that emerge from recent research on hidden feature removal is the observation that a small number of occluders hide a great number of occludees in a typical scene. The algorithms outlined in Refs. 29 and 30 are based on this insight. These algorithms are conservative in the sense that the set of occluders do not cull all of the invisible objects. These algorithms dynamically maintain a set of occluders as the viewer changes position, and they embed the objects in a spatial hierarchy so that a set of objects may be culled with one visibility query. In Ref. 29, to test if a convex occluder hides an axial bounding box, check the viewpoint against a set of tangent planes formed between the edges of the occluder and the vertices of the bounding box. By checking whether the viewpoint lies in the appropriate half-spaces, we can determine if the bounding box is completely hidden, partially hidden, or unoccluded by the occluder.

One of the drawbacks of this approach is that we cannot easily check if a set of occluders collectively hides an object. This problem can be solved by using an image-space hierarchical occlusion map (30). The image of the occluders is written onto the occlusion map, and this map is organized in a

**Table 1. Comparison of Different Visibility Algorithms**

| Algorithm | Easy to Implement? | Hardware Support? | Can it Support Large Databases? | How Much Overrendering? | Optimal Cases | Amt. of Preprocessing |
|---|---|---|---|---|---|---|
| $z$-buffer | Very easy | Yes | No | Maximum | Small data set w/no clear structure | None |
| BSP tree | Easy | No | No | Maximum | Small data set with obvious splitting planes | Fair |
| Hierarchical $z$-buffer | Easy | Maybe | Yes | Fair | Large data set where octree structure is likely to be respected | Fair |
| 2-D maze algorithm with ray casting | Fair | No | Yes | None | Interactive maze environment | Fair |
| Conservative Vis. + $z$-buffer | Fair | Partial | Yes | Fair | Large data set w/obvious cell structure | Very much (Fair, if the potentially visible set is computed on-the-fly.) |
| Visibility complex | Difficult | No | Maybe | None | Unstructured? | Very much |

hierarchy (similar to the hierarchical $z$-buffer described previously in this article) so that it is easy to check if an occludee bounding box is completely overlapped by the occluders when they are seen from the viewing position. If it is and if the conservative depth test determines that the occludee bounding box is behind the set of occluders, the occludee is culled.

One promising future direction in hidden feature removal is ray tracing. One advantage of ray tracing is that it is easily parallelizable and therefore particularly well suited for today's parallel machines. The readers are referred to the section on ray tracing in Ref. (10) for standard techniques used to improve the speed of ray tracing.

## SUMMARY

Today, rendering engines must deal with very large rendering databases. Also, for many interactive programs like video games and virtual reality systems, very complicated scenes must be rendered very efficiently to reach an interactive frame rate. One common theme, developed in this article, is that a combination of object–space culling and simple image-based techniques (such as the $z$-buffer) can be effectively used to render complicated environments by quickly eliminating most of the invisible objects in the first pass and resolving exact visibility using the $z$-buffer. Table 1 summarizes how different algorithms perform, according to a set of useful criteria. In comparing the performance of different algorithms, we make the assumption that the entire rendering database fits inside the random access memory. For special virtual memory techniques used in conjunction with conservative visibility, the readers are referred to Ref. 31.

## BIBLIOGRAPHY

1. J. L. Encarnação, A survey of and new solutions to the hidden line problem, *Proc. Interactive Comput. Graphics Conf.,* Delft, Holland, October 1970.
2. J. D. Foley et al., *Computer Graphics: Principles and Practice,* Reading, MA, Addison-Wesley, 1996.
3. M. E. Newell, R. G. Newell, and T. L. Sancha, A solution to the hidden surface problem, *Proc. ACM National Conf.,* 443–450, New York, ACM Press, 1972.
4. E. Catmull, *A Subdivision Algorithm for Computer Display of Curved Surfaces,* Ph.D. Thesis, Report UTEC-CSc-74-133, Computer Science Department, University of Utah, December 1974.
5. A. Mammen, Transparency and Antialiasing Algorithms Implemented with the Virtual Pixel Map Technique, *IEEE Comp. Grap. Appl.,* **9** (4): July 1989, pp. 43–55.
6. L. Carpenter, The A-buffer, an antialiased hidden surface method, *Proc. SIGGRAPH 1984,* 103–108, New York, ACM SIGGRAPH, July 1984.
7. A. Appel, Some techniques for shading machine renderings of solids, *Proc. Spring Joint Comput. Conf.,* 37–45, 1968.
8. Mathematical Applications Group, Inc., 3-D simulated graphics offered by service bureau, *Datamation,* **13** (1): 69, 1968.
9. R. A. Goldstein and R. Nagel, 3-D visual simulation, *Simulation,* **16** (1): 25–31, 1971.
10. H. Fuchs, Z. M. Kedam, and B. F. Naylor, On visible surface generation by a priori tree structures, *Proc. SIGGRAPH,* 124–133, New York, ACM SIGGRAPH, 1980.
11. H. Fuchs, G. D. Abram, and E. D. Grant, Near real-time shaded display of rigid objects, *Proc. SIGGRAPH,* 65–72, New York, ACM SIGGRAPH, 1983.
12. M. Paterson and F. F. Yao, Efficient binary space partitions with applications to hidden surface removal and solid modeling, *Discrete and Computational Geometry,* **5**: 485–503, 1990.
13. K. Mulmuley, *Computational Geometry: An Introduction Through Randomized Algorithms,* Englewood Cliffs, NJ: Prentice-Hall, 1994.
14. K. Weiler and P. Atherton, Hidden surface removal using polygon area sorting, *Proc. SIGGRAPH,* 214–222, New York, ACM SIGGRAPH, 1977.
15. P. R. Atherton, K. Weiler, and D. Greenberg, Polygon shadow generation, *Proc. SIGGRAPH,* 275–281, New York, ACM SIGGRAPH, 1978.
16. E. L. Fiume, *The Mathematical Structure of Raster Graphics,* San Diego: Academic Press, 1989.
17. S. E. Dorward, A survey of object-space hidden surface removal, *Int. J. Computat. Geometry Appl.,* **4** (3): 325–362, 1994.
18. N. Greene, M. Kass, and G. Miller, Hierarchical $Z$-buffer visibility, *Proc. SIGGRAPH,* 231–238, New York, ACM SIGGRAPH, 1993.
19. D. Meagher, Efficient synthetic image generation of arbitrary 3-D objects, *Proc. IEEE Conf. Pattern Recognition Image Process.,* 473–478, Long Beach, CA, IEEE Computer Society, June 1982.
20. S. J. Teller, *Visibility Computations in Densely Occluded Polyhedral Environments,* Ph.D. Thesis, U.C. Berkeley, 1992.
21. D. M. Y. Sommerville, *Analytical Geometry of Three Dimensions,* Cambridge: Cambridge University Press, 1959.
22. M. Pellegrini, Stabbing and ray-shooting in 3-dimensional space, *Proc. 6th Annu. ACM Symp. Computational Geometry,* 177–186, New York, ACM, 1990.
23. J. M. Airey, J. H. Rohlf, and F. P. Brooks, Jr., Towards image realism with interactive update rates in complex virtual building environments, *ACM SIGGRAPH Special Issue on 1990 Symposium on Interactive 3D Graphics,* **24** (2): 41–50, New York, ACM, 1990.
24. D. Luebke and C. Georges, Portals and mirrors: Simple, fast evaluation of potentially visible sets, *Proc. 1995 Symp. on Interactive 3D Graphics,* 105–106, New York, ACM, 1995.
25. F. Durand, G. Drettakis, and C. Puech, The 3D visibility complex, a new approach to the problems of accurate visibility, *Proc. Eurographics Workshop Rendering,* Amsterdam, North-Holland Pub. Co., June 1996.
26. M. Pocchiola and G. Vegter, The visibility complex, *Proc. Int. J. Computational Geometry Appl.,* **6** (3): 279–308, Teaneck, NJ, World Scientific, 1996.
27. S. J. Gortler et al., The lumigraph, *Proc. SIGGRAPH,* 43–54, New York, ACM SIGGRAPH, 1996.
28. M. Levoy and P. Hanrahan, Light field rendering, *Proc. SIGGRAPH,* 31–42, New York, ACM SIGGRAPH, 1996.
29. S. Coorg and S. Teller, Real-time occlusion culling for models with large occluders, *Proc. 1997 Symp. Interactive 3D Graphics,* 83–90, New York, ACM, 1997.
30. H. Zhang et al., Visibility culling using hierarchical occlusion maps, *Proc. SIGGRAPH,* 77–88, New York, ACM SIGGRAPH, 1997.
31. *Proc. 1992 Symp. Interactive 3D Graphics,* New York: ACM Press, 1992, pp. 11–20.

FRANKLIN CHO
DAVID FORSYTH
U. C. Berkeley

**HIDDEN SURFACE REMOVAL.** See HIDDEN FEATURE REMOVAL.

**HIGH-CONFIDENCE COMPUTING.** See FAULT TOLERANT COMPUTING.