Temporally Coherent Conservative Visibility[*]
(Extended Abstract)

Satyan Coorg   Seth Teller
Synthetic Imagery Group
MIT Laboratory for Computer Science
Cambridge, MA 02139
{satyan,seth}@graphics.lcs.mit.edu

Abstract

Efficiently identifying polygons that are visible from a changing synthetic viewpoint is an important problem in computer graphics. Even with hardware support, simple algorithms like depth-buffering cannot achieve interactive frame rates when applied to geometric models with many polygons. However, a visibility algorithm that exploits the occlusion properties of the scene to identify a *superset* of visible polygons, without touching most invisible polygons, could achieve fast frame rates while viewing such models.

In this paper, we present a new approach to the visibility problem. The novel aspects of our algorithm are that it is *temporally coherent* and *conservative*; for all viewpoints the algorithm *overestimates* the set of visible polygons. As the synthetic viewpoint moves, the algorithm reuses visibility information computed for previous viewpoints. It does so by computing *visual events* at which visibility changes occur, and efficiently identifying and discarding these events as the viewpoint changes. In essence, the algorithm implicitly constructs and maintains a linearized portion of an *aspect graph*, a data structure for representing visual events.

Keywords: Conservative visibility, temporal coherence, hierarchical representations, octrees, visual events, linearized dynamic aspect graphs.

1 Introduction

In computer graphics, identifying visible polygons or eliminating hidden polygons is an important component of efficient scene rendering algorithms. Despite the availability of the depth-buffer algorithm in hardware [2], the number of polygons in many geometric models is larger than hardware alone can process at interactive frame rates. One way to address this problem is by developing algorithms that resolve visibility at a higher level, and render only the visible portions of the model. Approximation algorithms that *overestimate* the set of visible polygons are useful, if they run in time comparable to the rendering time, and produce supersets which are only slightly larger than the true visibility. Synthetic observers in visual simulation applications typically move smoothly through the model, experiencing little change in visibility between viewpoints. Thus, there is ample spatial and temporal coherence to be exploited in most such applications.

1.1 Visibility Algorithms

Given a viewpoint and a polyhedral scene, the problem of computing the *exact* visibility, *i.e.*, computing an exact description of the visible portions of the scene, has been extensively researched [22, 16, 17]. Due to the availability of cheap memory, the simple depth-buffer algorithm, typically implemented in hardware [2], is widely used. This algorithm resolves visibility at each pixel. Along with its color, each pixel stores the distance of the represented surface fragment from the viewpoint; all pixel depths are set to some far away distance at the start of each frame. Each polygon to be rendered is *rasterized* into pixels along with its depth values. A newly computed pixel color replaces an existing color iff its depth value is less than the old depth value; that is, if the fragment is closer to the eye than the existing fragment. A disadvantage of resolving visibility at this late stage is that expensive operations like coloring and texturing are performed even on invisible fragments, and there is no obvious way to exploit the presence of large occluders near the observer to avoid rendering invisible polygons..

The well-known aspect graph encodes a representation of exact visibility for every qualitatively distinct region of viewpoints [19, 12, 11]. One drawback of this approach is that the visible portion of the scene may have higher complexity than the input scene itself – a scene with n polygons can have a visible portion of size $\Theta(n^2)$. Also, the number of qualitatively distinct viewpoint regions may be very large ($\Theta(n^9)$).

Given the availability of fast hardware to resolve visibility per pixel, it seems promising to design algorithms that *overestimate* the visible polygons in the scene. The output of such an algorithm could then be fed into the depth-buffer to synthesize a final image. The overestimation guarantees that the generated image is identical to that which would be generated by rendering every polygon in the scene. The challenge, of course, is to produce a useful tight upper bound on the visible polygons. This idea of conservative visibility has been exploited to design fast architectural walkthrough systems [1, 23, 10]. The idea in [23, 10] is that the input scene can be divided into *cells*, roughly corresponding to rooms

[*]The research described in this paper was funded in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-92-J-1310.

To appear in the Twelfth Annual ACM Symposium on Computational Geometry, Phildadelphia, May 24–26, 1996.

in a building, and *cell-to-cell, eye-to-cell, and eye-to-object* visibility can be used to bound exact visibility from above. Though this method eliminates most invisible polygons in architectural models, its generalization to models with less apparent cell structure seems difficult.

Conservative visibility has also been used in [14] to design an optimal algorithm for determining visibility in a scene containing only rectangles with sides parallel to the x and y axes. Exact visibility algorithms have also been developed for viewpoints moving along pre-specified paths (e.g., lines [18, 5]).

The hierarchical z-buffer algorithm [13] makes some use of *temporal* coherence by maintaining a list of polygons that are visible from the current viewpoint. For the next viewpoint, the algorithm draws polygons from this list first. The contents of the hierarchical z-buffer can then be used to *cull* invisible polygons. To exploit spatial coherence, this algorithm requires that the z-buffer support visibility queries. Such queries are not efficiently supported in most graphics hardware, and simulating the z-buffer in software involves significant overhead. Because it operates in image space, this algorithm is also susceptible to aliasing artifacts.

A temporally coherent visibility algorithm for scenes comprised only of convex objects is presented in [15]. The algorithm is complex, as it computes exact visibility, and no implementation is described.

Our algorithm represents an advance over existing efforts in the following respects. First, our construction of *imminent visual events* enables us to avoid checking for events that are unlikely to occur in the near future. Second, *object hierarchies* enable us to avoid processing large sets of invisible polygons or their pairwise interactions. Finally, our algorithm exploits the availability of fast depth-buffers in its generation of *conservative visibility sets*, greatly simplifying the algorithmics of visibility determination.

1.2 Overview

This paper describes a novel algorithm for visibility determination, based on the following ideas:

- Polygons may usefully be defined as "visible" according to any superset visibility criterion. Here, we say that a polygon is visible if it is not occluded by any *single* convex object.
- Under this definition of visibility, visual events – changes in the visibility status of a polygon – occur only when the viewpoint crosses specific planes. Thus, these planes partition 3-dimensional space into regions of constant visibility in a manner analogous to, but much simpler than, that of an aspect graph.
- From a particular viewpoint, only a small subset of such planes are *relevant*; as the viewpoint changes, it is sufficient to consider only these planes to detect a visual event.
- Dynamic, hierarchical data structures can be effectively used to both detect and maintain the set of relevant planes, without ever constructing the entire arrangement of visual event surfaces.

Given an initial viewpoint, our algorithm computes the polygons visible from that viewpoint as well as the relevant planes there. As the viewpoint changes, the algorithm detects the relevant planes that have been "crossed", and reports any visibility changes so caused. The maintained set of polygons is periodically fed to a depth-buffering algorithm for per-pixel visibility computations and rendering.

The paper is organized as follows. Section 2 defines the notion of conservative visibility and visual events. Section 3 considers the interaction of two convex polyhedra, and computes the relevant planes for a given viewpoint. Section 4 describes an algorithm based on octrees, for efficient detection and maintenance of occlusion relationships between pairs of objects. Section 5 describes techniques for efficiently maintaining visual events and inspecting them for crossings. Section 6 discusses an implementation of the algorithm, and its performance characteristics. Finally, Section 7 concludes.

2 Conservative Visibility and Visual Events

We assume that the input model is static, and specified as a set of convex polygons, and that incident polygons are grouped together to form polyhedra. In the degenerate case, each polyhedron could consist of a single polygon. We assume no *a priori* knowledge of observer motion.

In this paper, we make a distinction between *occluder* and *occludee* objects, and our algorithm considers only those interactions between occluders and occludees. This is motivated by the observation that, in many scenes, a few objects cause most occlusion and checking other objects for occlusion increases the overhead of the algorithm, without increasing the number of polygons found to be occluded. Crucially, the occluder/occludee determination is made dynamically as a function of the viewpoint. In particular, an object will typically act as an occluder for nearby viewpoints, and as an occludee for remote viewpoints.

Consider the following definition of visibility:

Definition 1 (Conservative Visibility) *A polygon is invisible iff all its vertices are occluded by a single convex polyhedron.*

Figure 1-a shows an example of an invisible polygon under this definition of visibility. This definition of visibility is justified by the following observations. First, the test for visibility of a polygon is greatly simplified; if all the vertices of a polygon are occluded by the same convex polyhedron, then the polygon is invisible. Second, our assumption greatly reduces the complexity of the visibility algorithm relative to that of an analogous exact visibility algorithm (see below). Of course, this definition fails to encompass some kinds of occlusions: those caused by *collusions* among convex polyhedra (Figure 1-b) and those caused by non-convex polyhedra (Figure 1-c). However, this is a reasonable tradeoff for many scenes encountered in practice.

Any dynamic visibility algorithm must track changes in visibility that occur as the viewpoint moves. The space of viewpoints can be partitioned into regions such that, within each region, the visibility remains constant. The boundaries separating these regions are called *visual events*. Under Definition 1, there is only one kind of visual event – that in which the projection of a vertex of the scene lies in the projection of an edge of the scene (Figure 2). This event is called a vertex-edge or VE event in [19, 12]. For example, this event could cause a change in visibility if polygon B was completely occluded by the object A, and some eye motion results in polygon B becoming (partially) exposed.

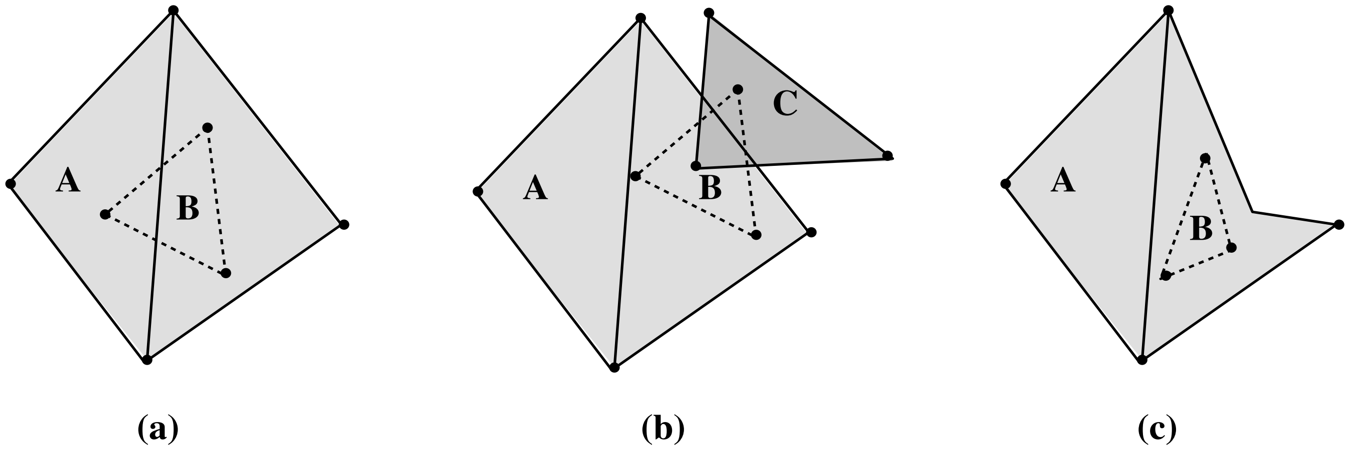

Figure 1: Figure (a) shows an invisible polygon B according to our definition of conservative visibility. Figures (b) and (c) show cases where B is determined visible, even though it does not contribute any pixels to the rendered image.

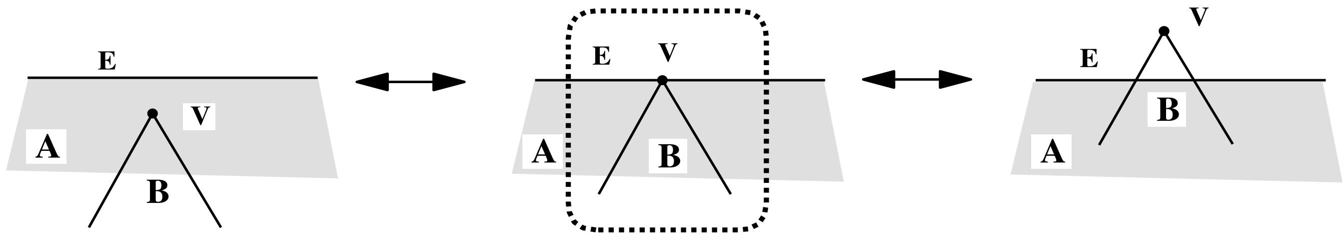

Figure 2: A VE event.

2.1 A Naive Visibility Algorithm

From the observer's point of view, VE events occur when the eye crosses the plane formed by the vertex V and the edge E. This observation immediately leads to a naive algorithm for tracking visibility changes. First, the algorithm generates planes formed by all pairs of scene vertices and edges. Using these planes, it divides 3-dimensional space into an arrangement of cells (see, e.g., [9]). Then, the algorithm associates with each cell the set of polygons visible from the cell, and associates cell boundaries with changes in the visibility set. Such a data structure is a "linearized" version of an aspect graph [12, 19].

Such a data structure could be used in an interactive setting as follows. For some initial viewpoint, the arrangement cell containing the viewpoint is located, and the visible polygons associated with that cell are reported. Given subsequent eye motion, any cell boundary crossings by the eye cause the visibility to be updated according to information stored with the boundary. In this way, the algorithm spends time reporting only visibility changes, rather than recomputing visibility for each new viewpoint.

However, a major drawback of the algorithm is the excessive time and storage cost of the preprocessing step. A scene of size n generates $\Theta(n^2)$ planes, which partition the 3-dimensional space into $\Theta(n^6)$ cells, requiring at least that much time for generating the linearized aspect graph. Also, the algorithm must store and read the arrangement when the scene is being viewed, making it impractical for scenes containing more than a few tens of polygons. In the next few sections, we describe a modified, practical (and implemented) version of the algorithm. The underlying idea is that any short sequence of viewpoints will typically visit a very small fraction of the arrangement cells. The naive algorithm can be modified to (implicitly) construct and (explicitly) examine only these cells, greatly reducing its storage and time complexity.

Finally, we reiterate that any algorithm that maintains *exact* visibility must also consider EEE events, in which three scene edges meet at a single image point [12, 19]. A data structure analogous to the one above, but incorporating EEE events, would have to consider $\Theta(n^3)$ triples of edges, and an induced arrangement of $\Theta(n^9)$ cells. Moreover, the event boundaries of EEE events are not planes, but quadric surfaces, greatly complicating the robust maintenance of cell boundaries in any practical implementation.

3 Relevant Planes

One way to reduce the complexity of the naive algorithm is to maintain only $O(n^2)$ planes and check each visual event after every viewpoint change[1]. We can do better by observing that the viewpoint must cross those planes that define the containing arrangement cell before crossing any other planes. As noted in Section 2, computing all possible cells is prohibitively expensive. However, it is possible to identify a subset of planes (called *relevant* planes) which is guaranteed to contain those planes which define the arrangement cell.

[1]Even with an efficient data structure such as that in [6], this prospect is impractical due to the space requirement. Also, as we describe, not all of these $O(n^2)$ planes are relevant.

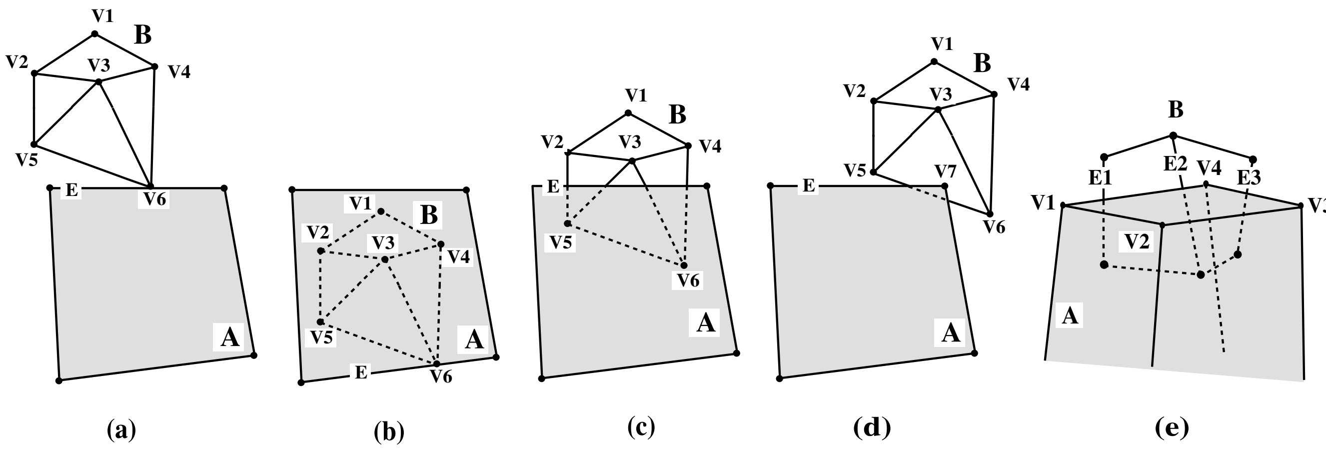

Figure 3: Interaction between polyhedra A and B, with A occluding B (shown as they would appear to the observer).

In this section, we consider the occlusion characteristics of two convex polyhedra. Later, in the context of n polyhedra, we show how our observations about pairwise occlusion lead to a selection algorithm that classifies only a fraction of the $O(n^2)$ arrangement planes as relevant for typical viewpoints.

Here is a brief overview of the terminology used below. A *silhouette* edge of a convex polyhedron A from a viewpoint is an edge E of A such that the projection of A lies completely on one side of the projection of E. An edge $E1$ *overlaps* an edge $E2$ if the projections of $E1$ and $E2$ intersect. $[V, E]$ denotes the plane formed by vertex V and edge E of the scene, and $[V1, V2, V3]$ denotes the plane formed by the three vertices $V1$, $V2$, and $V3$. We also use the plane corresponding to an event to label the event itself; *event* $[V, E]$ refers to the event which occurs when the eye crosses the plane $[V, E]$. *Separating* planes of two convex polyhedra are planes formed by an edge of one polyhedron and a vertex of the other such that the polyhedra lie on opposite sides of the plane [20]. *Supporting* planes are similar, except that both polyhedra lie on the same side of the plane.

Consider the occlusion relationship of two polyhedra A and B shown in Figure 3-a. For this viewpoint, only the separating planes of polyhedra A and B are relevant; the first visual event that can happen is for these two polyhedra to (begin to) overlap in the image. Similarly, supporting planes are relevant when A completely occludes B (Figure 3-b) as they detect the event during which the status of the occluded object changes from being completely occluded to being partially occluded.

The above mentioned events are sufficient for detecting (complete) invisibility of convex polyhedra. However, for determining visibility of individual polygons, events involving internal vertices of the occluded polyhedron are also relevant. Consider Figure 3-c. Clearly, the plane formed by an edge $[E, V3]$ is relevant; a small change in the viewpoint can result in the event $[E, V3]$, and the disappearance of polygon $V3, V5, V6$. Similarly, the planes formed by the edge E of A and vertices $V2 \ldots V5$ of B are relevant. These are the vertices of B that are "closest" to E; that is, some edge incident with a vertex in this set overlaps E in the image. However, the plane $[E, V1]$ is irrelevant, as any continuous motion of eye must cause one of the above events to occur first.

The listed planes determine exactly those visual events that occur when the observer moves from the current viewpoint. It is sufficient to maintain these planes to track visibility changes. However, as this set of planes itself changes with the viewpoint, the algorithm must also track events that cause any change to the relevant set. In addition to the events mentioned, two other kinds are relevant, as they affect the set of edges that overlap in the image. In Figure 3-d, the planes $[V7, V3, V5]$, $[V7, V5, V6]$ and $[V7, V6, V3]$ are relevant. If an event corresponding to one of these planes occurs, the edge E will overlap a new edge of B, changing the set of relevant planes. Finally, any change in the silhouette edges of the two polyhedra can also result in changes to edges that overlap (Figure 3-e), and thus events that detect changes to silhouette edges are relevant. For a silhouette edge E, the two planes formed by faces adjacent to E detect exactly this event.

To summarize, to maintain the occlusion relation between two convex polyhedra A (the occluder) and B (the occludee), the following planes are relevant:

- For each silhouette edge E of A or B, the planes corresponding to the faces adjacent to E;
- For each silhouette edge E of A:
  - Supporting and separating planes containing E;
  - For each edge $E' = (V, V')$ of B overlapping E, the planes formed by $[V, E]$ and $[V', E]$.
- For each silhouette vertex V of A:
  - Supporting and separating planes containing V;
  - For each edge E of B adjacent to face F that contains V, the plane $[V, E]$;

The following theorem demonstrates that the set of relevant planes from a given viewpoint is sufficient to capture all the visual events that occur when the viewpoint changes.

Theorem 2 (Relevant Planes) *Let A (occluder) and B (occludee) be two polyhedra and RP be the set of relevant planes from a viewpoint $P1$. Let $P2$ be a different viewpoint corresponding to a visual event. Then, either the visual event at $P2$ is in RP or there exists a viewpoint in the segment $[P1, P2]$ that corresponds to a visual event in RP.*

Proof: (Sketch) We prove the theorem by contradiction. Suppose a visual event occurs at viewpoint $P2$, but none of the relevant planes intersect the line segment $[P1, P2]$. We refer to the 2-dimensional images of the two polyhedra as seen from $P1$ and $P2$ as $IP1$ and $IP2$, respectively.

First, for a convex polyhedra, any change to the set of silhouette edges (and any corresponding change to the set of "front" faces) can occur only if the viewpoint crosses the plane of some face adjacent to one of the silhouette edges. As these planes are present in RP, it follows that individually, A and B are topologically identical in $IP1$ and $IP2$. In addition, they are also identical in the image generated for any viewpoint between $P1$ and $P2$. This fact ensures that the trajectory of vertices of A and B on the image plane as the viewpoint moves from $P1$ to $P2$ is a continuous line segment.

Let the visual event at $P2$ be $[V, E]$. Any visual event must involve a silhouette edge or vertex of the occluder; *i.e.*, either V is a silhouette vertex of A or E is a silhouette edge of A. We consider two cases:

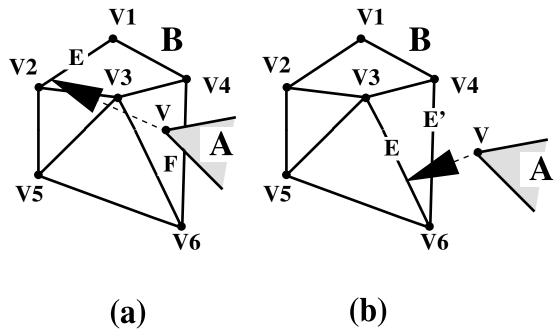

Figure 4: Silhouette vertex case

- (V is a silhouette vertex of A) First, consider the case when V lies in some face F of B in $IP1$ (Figure 4-a). As the visual event at $P2$ is not in RP, E is not one of the edges bounding F, and thus E lies outside F. By continuity, there must be an intermediate viewpoint between $P1$ and $P2$ such that V overlaps one of the edges bounding F, a contradiction.

  Second, let V lie outside B in $IP1$ (Figure 4-b). Again E cannot be one of the silhouette edges of B, as planes corresponding to such events (separating planes) are present in RP. Thus E lies inside B. Then, by continuity, there should be an intermediate viewpoint where V overlaps one of the silhouette edges of B (say E'). As the plane $[V, E']$ is a separating plane, it is contained in the set RP, a contradiction.

  A corollary to the above observations is that each vertex of A projects onto precisely the same face of B in $IP2$ as in $IP1$.

- (E is a silhouette edge of A) First, consider the case when E intersects some set of edges of B in $IP1$ (Figure 5-b). The crucial property is that if E intersects an edge in $IP1$, it also intersects it in $IP2$. For example, consider the edge $[V3, V5]$. If E does not intersect this edge in $IP2$, by continuity, there must exist a viewpoint from where the edge E overlaps either $V3$ or $V5$, a contradiction. Thus, the set of edges intersected by E in $IP1$ is a subset of the set of edges intersected by E in $IP2$. In addition, the incident vertices of E lie in the same faces both in $IP1$ and $IP2$. This implies that E intersects exactly the same edges in $IP2$, and E cannot participate in a visual event.

  If E does not intersect any edge of B, either E lies completely inside a face F of B (Figure 5-b) or E lies outside B (Figure 5-c). In the first case, E should also lie in F in $IP2$, as its incident vertices are constrained to lie in F. As F is convex, E cannot be a participant in a visual event with one of F's vertices. In the second case, the visual event corresponds to a separating plane (supporting plane, if V was occluded by A) – which is present in RP, a contradiction.

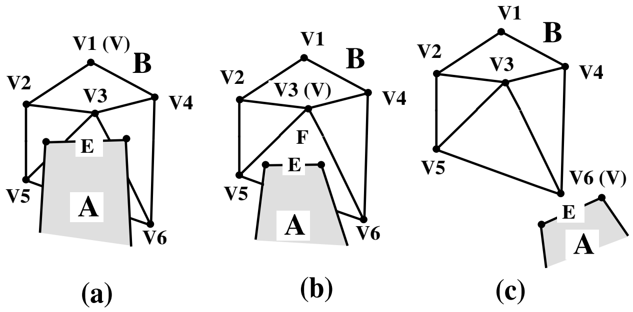

Figure 5: Silhouette edge case

In each case, we have proved that either the visual event at $P2$ does not exist, or it corresponds to one of the planes in RP, which proves the theorem. □

A naive algorithm for relevant plane maintenance between two objects of size m and n (whose faces are bounded by a constant number of edges) would maintain all nm planes. However, it is straightforward to show that the number of relevant planes is $\Theta(n+m)$. In practice, we need maintain even fewer planes, as the number of silhouette edges is always less than the total number of edges. Given two objects and a viewpoint, the set of relevant planes can be computed in $\Theta(n+m)$ time, by combining an algorithm for identifying supporting and separating planes of two convex polyhedra with an algorithm for computing the intersection of two convex 2-dimensional polygons [20].

When visual events occur due to a change in the viewpoint, the algorithm must update either the visibility or the set of relevant planes or both. Using a winged-edge data structure [3], we can update the visibility using only "local" computation. For example, in Figure 3-a, the edges adjacent to vertex $V6$ can be used to identify the edges of B that overlap E, and thus the new set of relevant planes. We omit detailed description of the various changes that occur when the eye crosses the different types of relevant planes. However, these are implemented in our system (described in Section 6).

The events that do not involve any changes in silhouette edges cause the insertion (deletion) of $\Theta(1)$ planes into (from) the set of relevant planes. However, an event that affects a silhouette edges could result in $\Omega(n+m)$ changes to the set of relevant planes, so the worst-case complexity of maintaining the occlusion relation is $\Theta(n+m)$ for each

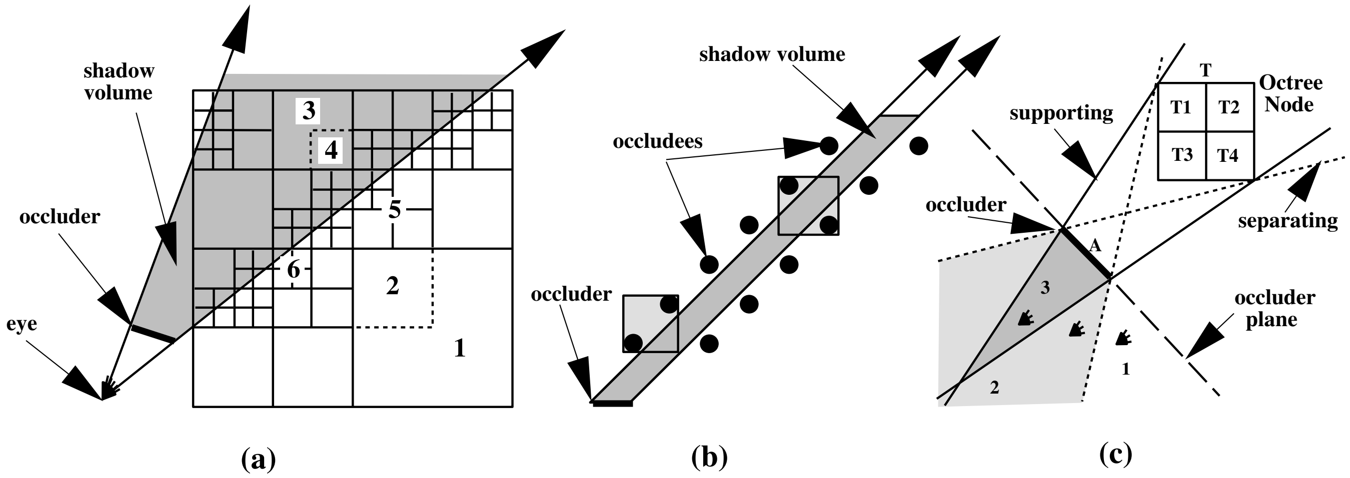

Figure 6: Figure (a) shows the octree nodes visited by the algorithm *Visible*. For clarity of the figure, we plase the occluder outside the octree. Figure (b) shows a worst-case input to the algorithm. Figure (c) shows the separating and supporting planes formed by an occluder A and an octree node T.

change in the viewpoint. In practice, we have found that the relevant plane set changes slowly with viewpoint (Section 6).

4 Object Hierarchies

Section 3 described techniques to efficiently maintain the occlusion relationships between two objects. For a set of n objects, maintaining all $\Theta(n^2)$ pairwise interactions is too expensive. It is also wasteful; most of the relationships are probably between objects that do not occlude each other from the instantaneous viewpoint, and such relationships can cause no polygons to be classified as invisible. Thus, it is advantageous to maintain relations between objects that are actually occluding or likely to occlude in the near future. This section describes an efficient hierarchical method for maintaining these relationships.

Consider the problem of identifying visible objects in the presence of a *single* occluder. One efficient way to achieve this is by organizing all objects in an octree data structure [21]. A simple way to build an octree is to associate the bounding box containing all the objects in the scene with the root node of the octree, and then recursively subdivide until some termination criteria is satisfied (*e.g.*, the number of objects in a leaf is less than some constant, or the dimensions of octree node are less than some constant).

Given an octree, the following algorithm gathers the objects in an octree that are *not* occluded by the occluder. *Gather*(T, S) simply collects all objects reachable from an octree node T and unions them to the set S. Also, $O(T)$ denotes the set of objects associated with a leaf T.

Visible(Octree Node T, Occluder A, View Point P,
    ObjectSet S)
  `if` A `completely occludes` T `then`
    `return`
  `else if` T `is a leaf then`
    `for each` $B \in O(T)$ `do`
      `if` A `does not occlude` B
        $S = S \cup \{B\}$
  `else if` T `is visible with respect to` A `then`
    Gather(T, S)
  `else if` T `is partial with respect to` A `then`
    `for each child` T' `of` T `do`
      Visible(T', A, P, S)

The algorithm *Visible* exploits spatial coherence in two ways. If an octree node is completely occluded, no further processing is made on its subtree. If the octree node is not occluded at all, the whole subtree is reported visible, without any further processing. In other words, the algorithm visits only those nodes T in the octree whose visibility status is different from T's parent (Figure 6-a). The region of space occluded by the occluder is its *shadow volume* [8].

If the number of octree leaves is m, the worst-case complexity of this algorithm is $\Theta(m)$, that is, the algorithm might make $\Theta(m)$ constant-time occlusion tests without reporting any occlusion relations. In Figure 6-b, the shadow volume does not intersect any objects, so there are no occlusion relations. However, the shadow volume intersects the bounding box of every two objects. Thus, the algorithm degenerates to an occluder test against each object, requiring $\Theta(m)$ time. In practice, the algorithm performs better, as many octree nodes are either fully occluded, or not at all occluded, by the occluder (Section 6).

An inefficient way of invoking *Visible* would be to compute new occlusion relations using the entire octree for every change in the viewpoint. However, this strategy can be modified to exploit coherence in observer motion, processing only those octree nodes whose visibility status changes. This modification involves maintaining a set of relevant separating and supporting planes corresponding to an occluder and an octree node. This idea is illustrated in Figure 6-c. First, occluder A can occlude octree node T only if the viewpoint and T are on opposite sides of the plane (set of planes) formed by the polygon(s) of A. This region can be divided into three qualitatively distinct regions as shown in the figure. If the viewpoint is in region 1, the planes formed by A and $T1 \dots T4$ are irrelevant. Such planes lie completely outside region 1, except at its boundary, and hence need not be considered as long the eye remains in region 1. These planes become relevant only if there is partial occlusion between A and T (*i.e.*, the viewpoint is in region 2). If the observer moves into region 3, where A fully occludes T, the

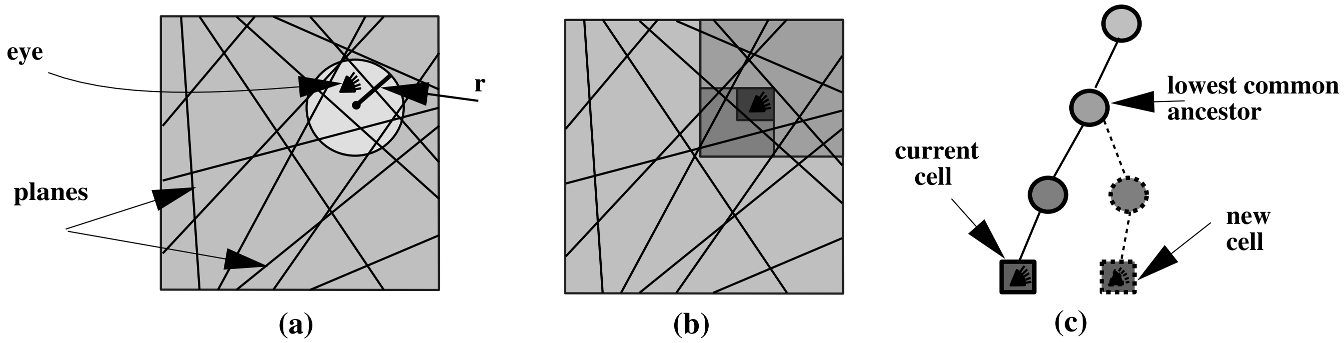

Figure 7: Illustration of the ball and octree algorithms for maintaining planes. The ball algorithm maintains a set of planes that intersect the sphere. The octree algorithm maintains the octree nodes along the path from the root node to the current viewpoint. Figure (c) shows the execution of the *UpdateEye* operation using the octree algorithm.

planes formed by A and $T1 \dots T4$ are again irrelevant; they can at most form the boundary of region 3, but region 3's boundary is already determined by the supporting planes of A and T.

The above observations suggest the following method of identifying relevant planes corresponding to an octree and an occluder. Using the algorithm *Visible* the nodes of an octree can be classified into four different categories (Figure 6-a). First, the octree contains *non-visited* nodes, the set of nodes T for which the test for occlusion is resolved by some ancestor of T. Examples are nodes 2 and 4, whose visibility can be determined by checking nodes 1 and 3. Visited nodes can further be classified into *fully occluded* (e.g., node 3), *not occluded* (e.g., node 1), and *partially occluded* (e.g., nodes 5 and 6). When the observer moves, the state of a non-visited node can change only after the state of its ancestor changes. Thus, the separating/supporting planes of a non-visited node are irrelevant. For visited nodes, the set of relevant planes depends on the type of the node. For fully occluded nodes, they are the supporting planes (boundaries of region 3 in Figure 6-c); for nodes that are not occluded, they are the separating planes (boundaries of region 1); and for partially occluded nodes, they are the union of supporting and separating planes (boundaries of region 2).

When the viewpoint crosses a relevant plane and enters a new region, the status of any affected octree nodes must be updated. For example, if the eye crosses a separating plane of an octree node and enters a region of partial occlusion (e.g., region 1 to region 2, in Figure 6-c), the algorithm updates the status of the octree node corresponding to the separating plane (node T), as well as the status of any of its children (nodes $T1 \dots T4$), terminating whenever the status is found to be unchanged. Thus, using these relevant planes the algorithm processes only changes in octree status, rather than the entire octree, for each viewpoint.

Multiple occluders can be handled by using the above algorithm individually for each occluder, or by organizing the occluders into a hierarchy and using an algorithm similar to *Visible* to report occlusion relations.

5 Dynamic Plane Maintenance

In previous sections, we have shown that maintaining occlusion relations in a scene can be cast as a problem of maintaining a set of relevant planes with respect to a changing viewpoint. In this section, we discuss some efficient data structures for maintaining the relevant plane set under the motion of the eye. The data structure should implement the following operations efficiently:

- *Insert* and *Delete* – As the set of planes varies from one viewpoint to the next, the data structure should support the appropriate insertions and deletions of planes to the set.
- *UpdateEye* – If the observer moves from the current viewpoint CV to a new viewpoint NV, the data structure should report all planes P that intersect the line segment $[CV, NV]$; that is, all planes that are crossed by the observer's motion.

In this section, p is used to denote the number of planes in the set.

A naive algorithm that simply maintained these planes as a set (e.g., with a hash table) would result in efficient $\Theta(1)$ *Insert/Delete* operations. The *UpdateEye* operation would make $\Theta(p)$ plane checks for each new viewpoint.

An alternative would be to construct the arrangement of the planes in the set. Using this data structure, the operation *UpdateEye* is very efficient, requiring only $\Theta(k)$ time if k planes are crossed [9]. However, the operations *Insert* and *Delete* take $\Theta(p^2)$ time. Since these operations occur at least as frequently as the *UpdateEye* operation, maintaining an arrangement would be impractical in our application. Also, the $\Theta(p^3)$ storage cost quickly becomes prohibitive. A simple improvement would be to maintain two sets of planes; the set of planes maintained by the naive algorithm as well as the set of planes (called *near planes*) that intersect a ball of radius r around some sample point (Figure 7-a). If the viewpoint remains inside the ball after an *UpdateEye* operation, the algorithm only checks whether any of the near planes have been crossed. Otherwise, the algorithm reexamines all the planes with respect to the new viewpoint, and generates a new set of near planes. The worst-case complexity of this algorithm is $\Theta(1)$ for *Insert* and *Delete* and $\Theta(p)$ for *UpdateEye*, and its space requirement is $\Theta(p)$.

A generalization of the above algorithm is to maintain planes with respect to a hierarchy (Figure 7-b). The idea is to maintain a *path* of an octree node formed by a root node (we elaborate on choosing the dimensions of the root node later) and $\log p$ descendants of the root node, each node containing the viewpoint. Each octree node in the path is associated with the set of planes that intersect the node. When the observer moves to a new viewpoint, the algorithm ascends the tree to the lowest common ancestor (LCA, Figure 7-c) of the current leaf node and the leaf node that contains the new viewpoint. The algorithm then reports the planes that are crossed by checking against the planes that are associated with the LCA, and updates the octree nodes to reflect the viewpoint change. The worst-case complexity of this algorithm is $\Theta(\log p)$ for *Insert* and *Delete* and $\Theta(p \log p)$ for *UpdateEye*. The algorithm uses $\Theta(p \log p)$ space in the worst-case.

Although the worst case complexities of these data structures are high, it is possible to show that these algorithms perform well under certain assumptions. If we assume that the number of planes that intersect a ball of radius r is proportional to r and the observer moves only a constant distance in each *UpdateEye* operation, the ball algorithm takes $O(\sqrt{p})$ expected time and the octree algorithm takes $O(\log p)$ expected time for an *UpdateEye* operation. Briefly, the worst case behavior of the octree algorithm results if the LCA of the current and the new viewpoint is the root of the octree node. The lower the LCA is in the tree, the smaller the number of planes checked. For a viewpoint moving in constant sized steps, it is more likely for the LCA to be lower than higher in the tree. The expected complexity, that is, the total cost of visiting a node at some level (the number of planes associated with a node), weighted with probability of reaching that level, is $O(\log p)$.

In practice, we employ a dynamic strategy to choose the radius r in the ball algorithm and the dimensions of the root node in the octree algorithm. For the ball algorithm, it is advantageous to increase r as long as the benefit (more *UpdateEye* operations are inside the ball) outweighs the cost (more planes need to be checked). For the octree algorithm, it is advantageous to expand the root node until it intersects all relevant planes.

6 Implementation

We have implemented the algorithms described in this paper in about 3,500 lines of C++ code. The programs are embedded in the SGI OpenInventor toolkit, and run on a SGI Indy workstation with one 133 MHz R4000 processor and 64 Mb of physical memory. The timings presented in this section reflect only time spent in the visibility algorithm and exclude time required to draw the visible polygons.

Vertices	Init. (sec)	Planes
32	0.04	49
64	0.05	69
128	0.07	93
256	0.10	133
512	0.16	184

Table 1: Initialization times and relevant plane set sizes, for different sized polyhedra.

Our first experiment was designed to analyze the performance characteristics of the algorithm presented in Section 3. The scene consists of two polyhedra of equal size generated by choosing points on the surface of a sphere of unit radius and computing the convex hull of these points. The observer moves in a circle of fixed radius around the two polyhedra. Table 1 reports the number of relevant planes maintained by the algorithm and the time spent by the algorithm in initialization for polyhedra of increasing complexity. Note that the number of relevant planes increases *sub-linearly* with the size of the polyhedra (and proportional to the silhouette complexity), in contrast to the quadratic behavior of the naive algorithm.

Speed	Insert	Delete	Cross
64	0.314	0.314	0.114
32	0.157	0.157	0.061
16	0.079	0.079	0.031
8	0.039	0.039	0.015
4	0.020	0.020	0.008
2	0.010	0.010	0.004
1	0.005	0.005	0.002

Table 2: This table gives the average fraction of relevant planes inserted, deleted, and crossed for a change in the viewpoint. The observer moves at different speeds (given relative to the slowest speed).

Table 2 shows some important performance characteristics of the algorithm: the fraction of relevant planes that are inserted/deleted/crossed for varying speeds of the observer. The results are reported for polyhedra containing 512 vertices and the speeds are chosen such that they correspond to the range of speeds of an observer smoothly walking through the scene. Note that a change in viewpoint causes only small changes to the relevant set of planes, reflecting the temporal coherence exploited by the algorithm.

Speed	View Change (msec)		
	Naive	Ball	Octree
64	13.60	17.39	22.03
32	6.81	8.63	10.7
16	3.52	4.34	6.01
8	1.87	2.17	3.28
4	1.03	1.08	1.65
2	0.57	0.52	0.83
1	**0.30**	**0.25**	**0.40**

Table 3: This table gives time spent in the algorithm to process a viewpoint change for an observer moving at different speeds.

Table 3 shows the time spent in the algorithm for processing a viewpoint change in the above experiment. The table also compares the various techniques of dynamic plane maintenance described in Section 5. We do not provide data about the polygons culled in this experiment as it is highly dependent on the viewpoint.

Table 4 shows the fraction of relevant planes that intersect a ball of radius r centered around the viewpoint (averaged over different input sizes). Note that the fraction of planes that intersect this sphere is very nearly linear in the radius of the sphere. Thus, for slowly moving observers, the ball and octree algorithms check only a fraction of relevant planes for each viewpoint change (Table 5). However, for faster speeds of the observer, the advantage is offset by the additional complexity of *Insert* and *Delete*, and the ball and octree algorithms perform better only for slower speeds (Table 3).

Radius of Sphere	Fraction of Relevant Planes
1	0.02
2	0.05
4	0.10
8	0.19
16	0.34
32	0.53
64	0.73
128	0.86

Table 4: This table illustrates the (almost) linear relation between the radius of a sphere and the fraction of relevant planes that intersect it. The radius is given relative the distance the viewpoint moves in a step.

Speed	Frac. of. Planes Checked		
	Naive	Ball	Octree
64	1.000	0.775	0.853
32	1.000	0.641	0.682
16	1.000	0.495	0.481
8	1.000	0.374	0.310
4	1.000	0.279	0.189
2	1.000	0.204	0.109
1	**1.000**	**0.148**	**0.063**

Table 5: This table compares the fraction of relevant planes checked by the algorithms Naive, Ball, and Octree.

We also studied the performance of the algorithm on actual scenes: the fifth floor model of the Berkeley Soda Hall building (Soda), and a city database from Viewpoint DataLabs (City). Though we described the algorithm in terms of octrees, the actual implementation used kD-trees [4], a similar hierarchical data structure. For these experiments, the objects are just single polygons. The height of the octree containing the polygons and the number of occluders (k) are inputs to the program; these were chosen to achieve interactive frame rates. As the viewpoint changes, the program automatically chooses k occluders based on their approximate area in the image. For these two scenes, initialization took about 3 seconds of CPU time, about half of which was spent constructing the octree. On average, the algorithm took around 0.03–0.07 seconds of CPU time to process each viewpoint change. Table 6 shows other performance characteristics of the algorithm averaged over many "random walks" of an observer walking through these two

Scene	Polys		Octree Nodes		Frac. of
	Initial	Final	Total	Relevant	Invis. Polys
Soda	1685	2971	255	31	0.68
Town	2857	4722	255	35	0.36

Table 6: In the table, the column *Final* denotes the total number of polygon fragments after octree construction. Relevant octree nodes are presented per occluder. The last column shows the fraction of truly invisible polygons culled by the algorithm (averaged over viewpoints visited in the random walks).

scenes. Note that the number of relevant octree nodes detected by the algorithm is significantly less than the total number of octree nodes.

Using depth-buffer hardware, we computed the "true" set of visible polygons (and its complement, the truly invisible polygons) at each viewpoint visited by the observer. As the observer has a spherical field of view, we used the following method to compute the truly visible polygons. The polygons were drawn on the six faces of a unit cube centered at the viewpoint. Polygons that paint some pixel in one of the faces of the cube are truly visible from that viewpoint. The average fraction of truly visible polygons varied between 5% and 10% of the number of (final) input polygons. The efficacy of the culling algorithm, as defined by the fraction of truly invisible polygons culled by the algorithm, is shown in Table 6. Note that the algorithm is effective in culling away a significant fraction of the scene, even for models like City, where the cell/portal technique of [23] is less effective.

7 Conclusions and Future Work

This paper described a visibility algorithm that exploits the temporal coherence in the motion of an observer through a geometric model. Casting the algorithm as one of conservative visibility maintenance greatly simplified its design, and reduced its time and storage complexity. Nevertheless, the polygons output by the algorithm can be used to synthesize the correct image using a depth-buffer. Our results show that temporal coherence can be exploited to design an efficient algorithm for visibility that is able to support viewing of actual scenes in interactive frame rates, for scenes much larger than exact visibility algorithms can handle.

An alternate approach to this problem, using dynamically chosen sets of connected polygons to perform visibility culling, and caching across viewpoints to exploit temporal coherence is presented in [7]. This algorithm tends to have a better response to occasional "jumps" in the movement of the observer, albeit at the expense of performing slightly worse for smooth motion of the observer.

One idea to improve the efficiency of the algorithm is to make it *asynchronous*, that is, predict the motion of the observer a few frames in advance and use this prediction to compute the visible polygons for some region before the observer reaches the region. Also, there may be some utility in precomputing the relevant planes and visible polygons for all points on some 3D grid, and using the nearest grid point to "reinitialize" the dynamic data structure whenever the viewpoint changes abruptly.

Another way to extend the algorithm is to identify some invisible polygons that are occluded by multiple colluding convex objects or non-convex objects. One possible approach is to find a region where sets of objects "act" as a single convex object, and process them appropriately.

8 Acknowledgements

We thank Viewpoint Datalabs for providing the City model. Also, thanks to Nina Amenta, Tom Funkhouser, Doug Voorhies, and the anonymous referees for their useful comments, and to Prof. Arvind for his continuing support.

References

[1] AIREY, J. M., ROHLF, J. H., AND BROOKS, JR., F. P. 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 (1990), 41–50.

[2] AKELEY, K. RealityEngine Graphics. *Computer Graphics (Proc. Siggraph '93)* (1993), 109–116.

[3] BAUMGART, B. G. A Polyhedron Representation for Computer Vision. In *Proc. AFIPS Natl. Comput. Conf.* (1975), vol. 44, pp. 589–596.

[4] BENTLEY, J. Multidimensional binary search trees used for associative searching. *Communications of the ACM 18* (1975), 509–517.

[5] BERN, M., DOBKIN, D., EPPSTEIN, D., AND GROSSMAN, R. Visibility with a Moving Point of View. *Algorithmica 11* (1994), 360–378.

[6] CLARKSON, K. L. New Applications of Random Sampling in Computational Geometry. *Discrete Computational Geometry 2* (1987), 195–222.

[7] COORG, S., AND TELLER, S. A Spatially and Temporally Coherent Object Space Visibility Algorithm. Tech. Rep. TM-546, Laboratory for Computer Science, MIT, 1996.

[8] CROW, F. C. Shadow Algorithms For Computer Graphics. *Computer Graphics (Proc. Siggraph '77) 11*, 2 (1977), 242–248.

[9] EDELSBRUNNER, H., BRAUER, E. W., ROZENBERG, G., AND SALOMAA, A. *Algorithms in Combinatorial Geometry.* EATCS Monographs on Theoretical Computer Science, 1987.

[10] FUNKHOUSER, T., SÉQUIN, C., AND TELLER, S. Management of Large Amounts of Data in Interactive Building Walkthroughs. In *Proc. 1992 Workshop on Interactive 3D Graphics* (1992), pp. 11–20.

[11] GIGUS, Z., CANNY, J., AND SEIDEL, R. Efficiently Computing and Representing Aspect Graphs of Polyhedral Objects. *IEEE Transactions on Pattern Analysis and Machine Intelligence 13*, 6 (1991), 542–551.

[12] GIGUS, Z., AND MALIK, J. Computing the Aspect Graph for Line Drawings of Polyhedral Objects. *IEEE Transactions on Pattern Analysis and Machine Intelligence 12*, 2 (1990), 113–122.

[13] GREENE, N., KASS, M., AND MILLER, G. Hierarchical Z-Buffer Visibility. In *Computer Graphics Proceedings, Annual Conference Series, 1993* (1993), pp. 231–240.

[14] GROVE, E., MURALI, T. M., AND VITTER, J. The Object Complexity Model for Hidden-Surface Elimination. In *Proc. 7th Canad. Conf. Comput. Geom.* (Québec City, Canada, 1995), pp. 273–278.

[15] HUBSCHMAN, H., AND ZUCKER, S. W. Frame to Frame Coherence and the Hidden Surface Computation: Constraints for a Convex World. *ACM Trans. on Graphics (USA) 1* (Apr. 1982), 129–162.

[16] MCKENNA, M. Worst-case Optimal Hidden Surface Removal. *ACM Transactions on Graphics 6*, 1 (1987), 19–28.

[17] MULMULEY, K. An Efficient Hidden-Surface Removal Algorithm, I. *Computer Graphics (Proc. Siggraph '89) 23*, 3 (1989), 379–388.

[18] MULMULEY, K. Hidden Surface Removal with Respect to a Moving View Point. In *Proceedings of the Twenty Third Annual ACM Symposium on Theory of Computing* (New Orleans, Louisiana, 6–8 May 1991), pp. 512–522.

[19] PLANTINGA, W., AND DYER, C. Visibility, Occlusion, and the Aspect Graph. *Int. J. Computer Vision 5*, 2 (1990), 137–160.

[20] PREPARATA, F. P., AND SHAMOS, M. I. *Computational Geometry: an Introduction.* Springer-Verlag, 1985.

[21] SAMET, H. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS.* Addison-Wesley, 1990.

[22] SUTHERLAND, I. E., SPROULL, R. F., AND SCHUMACKER, R. A. A Characterization of Ten Hidden-Surface Algorithms. *Computing Surveys 6*, 1 (1974), 1–55.

[23] TELLER, S., AND SÉQUIN, C. H. Visibility Preprocessing for Interactive Walkthroughs. *Computer Graphics (Proc. Siggraph '91) 25*, 4 (1991), 61–69.