

# Conservative Visibility Preprocessing for Efficient Walkthroughs of 3D Scenes\*

Harry Plantinga  
Department of Computer Science  
University of Pittsburgh  
(412) 624-8407  
e-mail address: [planting@cs.pitt.edu](mailto:planting@cs.pitt.edu)

## ABSTRACT

In this paper I address the problem of pre-computing visibility information to increase the efficiency of interactive walkthroughs of large 3-D CAD models such as buildings. The approach presented assumes the use of graphics hardware including hidden-surface removal hardware. The goal is to automatically partition viewpoint space into regions and pre-compute visibility information for each region that is *conservative* in the sense that the display list computed for a region is a superset of the set of objects that are weakly visible from that region. To render a view from a region, the display list for the region is processed with hidden-surface removal hardware, so that the correct scene is rendered. However, only a small fraction of the faces in a scene may have to be rendered from any viewpoint. The conservative nature of the visibility computation makes possible the use of many techniques for speeding the preprocessing phase and reducing storage requirements for visibility information, including the partitioning of scene faces into *objects* and the selection of large scene faces (called *walls*) for occlusion tests. The amount of time and space required for the visibility pre-computation depends only on the number of objects and walls and not the size of the scene, so it is possible to adjust the amount of pre-computation time and space used, with corresponding effects on the efficiency of the on-line walkthrough. Results concerning the number of faces that must be rendered under various object selections are presented for an example model.

## INTRODUCTION

For architectural modeling, CAD, or virtual reality applications, it is desirable to be able to interactively “walk through” a model of a building or similar scene. A characteristic of this problem is that the number of polygons or other primitives in the model is often large, but the average number of polygons visible from a viewpoint is much smaller. It is possible to address this problem by rendering the whole scene with a Z-buffer at each frame, but this approach is inefficient when the number of visible polygons is much smaller than the scene size. The bottleneck in such an application is typically the number of Z-buffered shaded 3-D polygons that can be rendered per second by

the graphics hardware. The problem considered here is to reduce the number of polygons rendered per frame by identifying polygons that are not visible from viewing regions in a pre-computation phase, so that they need not be rendered on-line. In this paper, I assume that the size of the model is large, so that the amount of space available for additional visibility data is limited, perhaps to about the size of the model or another prescribed limit.

On-line culling is a partial answer to reducing the number of polygons rendered at each frame, but sophisticated culling algorithms do not remove enough polygons or require too much time. Another approach is a complete visibility pre-computation, described elsewhere [10]. In that approach viewpoint space is partitioned into regions of constant *aspect* or topological appearance. From any viewpoint, the corresponding region and thus the exact set of visible polygons (and parts of polygons) is known. However, that approach requires too much preprocessing time and storage for large models and unrestricted viewer motion. Furthermore, it doesn't take advantage of hidden-surface removal hardware.

In this paper I discuss modifications of the visibility-pre-computation approach that make it practical for large models and that take advantage of hidden-surface-removal hardware. The result is an approach to visibility pre-computation in which the amount of time and space needed to compute and store visibility information are adjustable. It is possible to decide in advance how much pre-computation time and space are available and to compute visibility data subject to those limits. The on-line walkthrough will be correspondingly more efficient.

## CONSERVATIVE VISIBILITY, OBJECTS, AND WALLS

In the complete visibility pre-computation approach to the walkthrough problem, the pre-computation results in a partition of viewpoint space into maximal regions of constant topological appearance. These regions are bounded by *events*, or viewpoints at which the image changes topologically. At each boundary is stored a representation of the change that occurs in the image: typically, a list of polygons that disappear, appear, or change in topological appearance. A data structure called an *event graph* stores all of these boundaries together with add and delete lists for each. A display list is maintained as the viewpoint moves. In this approach, only the polygons (and parts of

---

\*This research was supported in part by NSF Grant No. CCR-9007612 and by the Central Research Development Fund of the University of Pittsburgh.

polygons) that are actually visible in a frame are rendered. This approach maximizes the speed of the on-line portion of the algorithm, and a Z-buffer is not required.

In an earlier paper [10] this approach was applied to 360° rotations of polyhedral scenes of small-to-moderate size (up to about 2000 faces). A rotation of a scene is equivalent to moving the viewpoint in a circle around the scene, so for this problem viewpoint space is 1-D. Event viewpoints along the circle are stored, and at each there is an add list and a delete list for faces that appear and disappear. In that case, it was found that the approach is practical for models of this size. The amount of data that must be stored was a few times the amount of data in the model for visually complex models of that size. An implementation showed that using this approach makes possible the rotation of such polyhedra in real time (e.g. 20 frames per second), with hidden lines removed, on an IBM RS6000/320 workstation without graphics hardware. The frame rates were limited by vector drawing speed only; objects were rotated at about the same speed that would have been possible without hidden line removal.

Unfortunately, extending the above approach to larger models and greater viewer freedom presents problems. Plantinga and Dyer [9] show that the number of topologically-distinct regions into which viewpoint space is partitioned is  $\Theta(n^3)$ ,  $\Theta(n^6)$ , or  $\Theta(n^9)$  for 1-, 2-, or 3-dimensional viewpoint spaces in the worst case for scenes with  $n$  vertices. Even a convex polyhedron of  $n$  faces generates  $\Theta(n^2)$  regions in a 2-D viewpoint space. The worst case only occurs for pathological scenes (e.g. a grid behind a grid), and the number of regions for natural scenes is typically orders of magnitude better. Still, the amount of data that must be represented for large scenes can be astronomical. Furthermore, this approach does not take advantage of hidden-surface removal hardware when it is available.

This paper presents modifications of this approach to reduce the storage and preprocessing time requirements and take advantage of such hardware. Since walkthroughs are to be interactive, I will use a 2-D viewpoint space, typically the plane parallel to the ground and at eye level. This allows the user to move around and to look around or up or down, but the user cannot move up or down. It is possible to store many such planes, connected in various ways, to allow for buildings with multiple floors, stairs, etc. The approach also generalizes very simply to 3-D viewpoint space and unrestricted viewer motion, possibly at the expense of additional storage requirements and preprocessing time for a given efficiency.

The goals of taking advantage of both visibility pre-computations and hardware for hidden surface removal may at first seem difficult to reconcile. However, it turns out that they complement each other nicely. Graphics hardware always has a limit on the number of faces that can be rendered per frame, so visibility pre-computation is desirable. When hidden-surface removal hardware is available, it is

possible to relax the requirements on visibility computations, so that they need only be *conservative* rather than exact. A conservative visibility computation is one in which objects are represented as being invisible or possibly weakly visible from a region, where a face is said to be weakly visible when it is at least partly visible from some viewpoint in that region. Thus, if a face is represented as invisible from a region, it is known that the face is not even partly visible from any viewpoint in that region. If it is represented as possibly weakly visible, it may or may not be visible from some viewpoints. It is then necessary to perform hidden surface removal in order to correctly render the scene, but the number of faces that must be rendered may be much smaller than the number of faces in the scene. For any scene, many different conservative visibility computations are possible, varying in visibility data storage requirements from no additional data to as much data as is required for the exact visibility computation. In principle, it is possible to decide in advance how much space is available for visibility information and to compute exactly that amount of visibility information.

If visibility computations are conservative, it is no longer necessary to compute and represent a huge number of regions with different visibility characteristics. A smaller number of regions with conservative visibility information suffices. Also, it is no longer necessary to compute and store events concerning a change in a polygon's appearance due to occlusion; hidden portions may be automatically removed by the hidden surface removal hardware. Furthermore, a few moving objects may easily be handled by representing them as always possibly visible.

Another result of the use of conservative visibility computations is that it is possible to group sets of scene faces together into *objects*. The visibility status of an object as a whole is represented, rather than the visibility status of the individual faces. When any face of the object is possibly weakly visible from a region, the object is possibly weakly visible. If the faces selected for an object have approximately the same visibility characteristics, this technique can result in a great space savings without much penalty in the efficiency of the visibility computation. A set of faces will have approximately the same visibility characteristics if the faces are in close proximity and if no other large face divides the set. If a scene is constructed from a number of primitives from a library, such as pieces of furniture or mechanical parts, these primitives make natural choices for objects. If no such construction-time information is available, objects may be chosen as sets of the desired number of faces that are spatially compact and not separated by other large faces. In this paper I assume that the separation into objects is given.

Objects may consist of hundreds or thousands of faces, and visibility computations that involve all of the faces of an object would be expensive. For greater efficiency, the minimal bounding box with edges parallel to the coordinate axes is used for each object. Visibility computations

are done on the bounding box rather than on the individual object faces. From any viewpoint outside the box, if the box is invisible, then all of the faces in the box are known to be invisible. For viewpoints inside the box, the object is said to be possibly weakly visible. Thus, conservatism is preserved. An additional benefit of the use of bounding boxes is that curved or arbitrary primitives or moving objects may be used as long as a box bounding the primitive can be found.

Note that large faces such as walls of a building contribute significantly to the occlusion of objects, while small faces do not contribute as much. When determining the regions of viewpoint space from which an object is invisible, it is possible to ignore the small faces and test for occlusion only with the large faces, while maintaining the conservative nature of the visibility computation. This technique can result in significant time savings in the visibility computations when there are many small faces and a few large faces, as may be the case in a model such as a building with furniture. A possible approach is to use a fixed number of the faces of the scene that have the largest area for occlusion tests. These faces will be called *walls*, but they should not be confused with walls of buildings; if the model is a building, the actual set of faces used may be much larger or smaller than the set of building walls. Also, I will not try to represent events involving faces whose invisibility may easily be determined on-line by backface culling or viewing-pyramid clipping.

Viewpoint space is automatically partitioned by this algorithm into regions of constant conservative visibility characteristics. These regions will be called *cells*. With a 2-D viewpoint space, cells are bounded by edges and quadratic curve segments. Along with each cell boundary is stored a list of objects that become invisible and a list of objects that become possibly weakly visible.

Clearly, the choice of walls and objects affects how the display list for a cell (i.e. the set of possibly weakly visible objects) compares to the set of weakly visible faces for that cell. All of the faces on the display list must be sent into the coordinate-transformation, clipping, and culling part of the rendering pipeline. I have assumed that clipping and culling can efficiently be done on-line, so that the display list for a cell contains the faces that are visible from all viewing directions. However, it is also possible that for large enough display lists, most of which are clipped and culled away, transforming, clipping, and culling may also be a bottleneck. Therefore two measures of the efficiency of a conservative visibility computation will be used:  $e_t$ , the *transform efficiency*, which is a measure of the number of invisible faces on the display list for a cell that must be transformed and clipped or culled away, and  $e_r$ , the *render efficiency*, which is a measure of the proportion of invisible faces that remain on the display list after clipping and culling and must be rendered.

Given a particular scene with a bounded viewing region, let

$f_t$  = the total number of faces in the scene

$m_t$  = average number of faces partly visible from a viewpoint in the scene (before clipping and culling)

$c_t$  = the average number of faces on the display list for any viewpoint as a result of a conservative visibility computation for the scene.

The transform efficiency,  $e_t$ , of that conservative visibility computation for the scene is then defined as

$$e_t = (f_t - c_t) / (f_t - m_t) \quad (\text{Eq. 1})$$

Thus,  $e_t$  is a measure of how close display lists are to ideal. When each face is an object and a wall, efficiency is maximized ( $e_t = 1$ ); when no walls are used so that all objects are always on the display list,  $e_t = 0$ .

In order to measure render efficiency, let

$f_r$  = the average number of faces in the scene that remain after clipping after clipping and culling from a viewpoint

$m_r$  = average number of faces partly visible from a viewpoint in the scene, after clipping and culling

$c_r$  = the average number of faces on the display list for any viewpoint as a result of a conservative visibility computation for the scene, after clipping and culling

The render efficiency,  $e_r$ , for a conservative visibility computation for the scene is then defined as

$$e_r = (f_r - c_r) / (f_r - m_r) \quad (\text{Eq. 2})$$

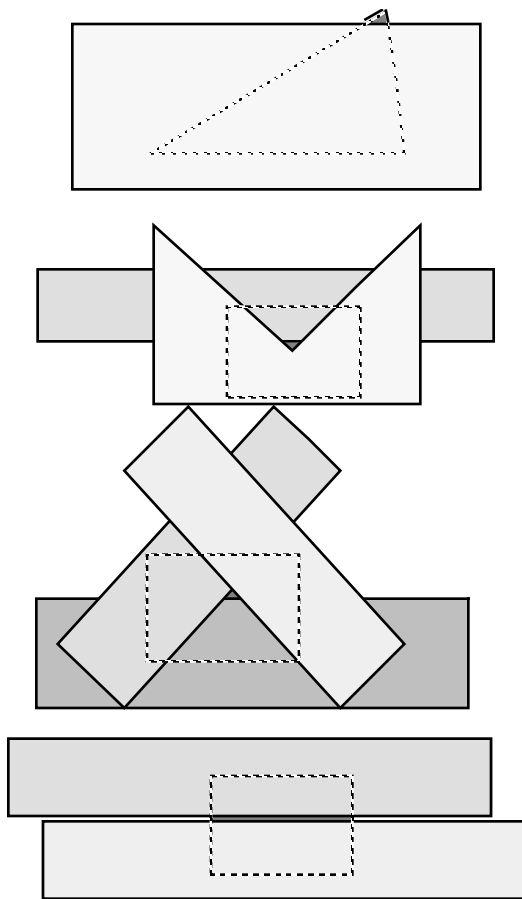
Render efficiency is a measure of the number of extraneous faces that must be rendered on average for a particular scene and conservative visibility computation. When each face is an object and a wall, only partly visible faces are rendered and efficiency is maximized ( $e_r = 1$ ). When all faces in the scene must be rendered except those clipped or culled away, efficiency is minimal ( $e_r = 0$ ).

## COMPUTING THE VIEWPOINT SPACE PARTITION

For a given set of objects and walls, viewpoint space is partitioned into maximal cells from which the same set of objects is possibly weakly visible, using only the given walls in occlusion tests. This partition is called the *Viewpoint Space Partition* (VSP). The algorithm for constructing the VSP works by computing events, which are viewpoints from which the image changes topologically upon an arbitrarily small change in viewpoint. These events form the boundaries of the VSP and the faces that generate

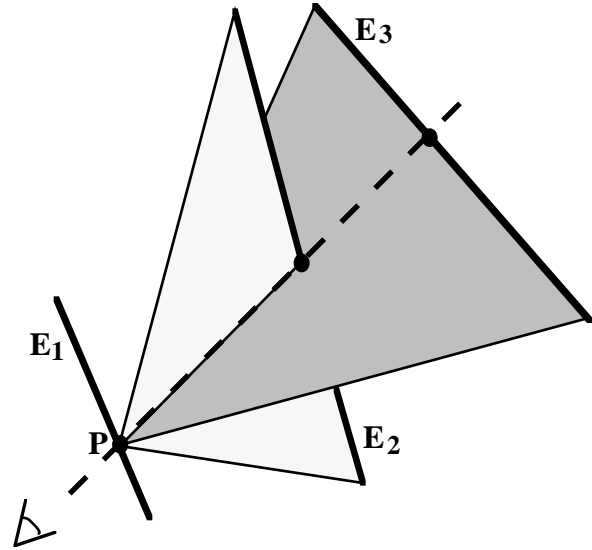
them determine the add and delete lists for the VSP boundaries. The description of the algorithm here will be brief, with emphasis on differences from earlier work [9-11].

Visual events occur at viewpoints from which an edge and a vertex of the scene appear to intersect (EV events) or from which three edges of the scene appear to intersect (EEE events). Examples of image scenes near event viewpoints are shown in Figure 1; it is possible to enumerate all configurations of an edge and a vertex or three edges and a face of the scene that may result in an event [9]. I will speak of the edges participating in an event and the face occluded at the event. The face occluded at the event may contain one of the edges participating in the event, but in some cases it does not.



**Figure 1. The image from viewpoints near events. Four different examples are shown.**

When a vertex and an edge or when three edges appear to intersect, there is a line of sight from the event viewpoint that passes through the edges participating in the event without intersecting any other face. For EV events, the line of sight in question lies in the plane containing the vertex and the edge. In the case of EEE events, the line of sight passes through three object edges (see Figure 2).



**Figure 2. The line of sight through point P on E<sub>1</sub> that passes through E<sub>2</sub> and E<sub>3</sub>.**

In Figure 2, the dotted line is the line of sight that passes through the three edges. It is the intersection of the plane containing P and E<sub>2</sub> and the plane containing P and E<sub>3</sub>. If edge E<sub>1</sub> is given by  $\mathbf{p}_1 + s \mathbf{a}_1$ , E<sub>2</sub> by  $\mathbf{p}_2 + s_2 \mathbf{a}_2$ , and E<sub>3</sub> by  $\mathbf{p}_3 + s_3 \mathbf{a}_3$ , then for any point  $\mathbf{P} = \mathbf{p}_1 + s \mathbf{a}_1$  on E<sub>1</sub>, the direction of the line of sight is given by

$$\mathbf{d} = [(\mathbf{p}_1 + s \mathbf{a}_1 - \mathbf{p}_2) \times \mathbf{a}_2] \times [(\mathbf{p}_1 + s \mathbf{a}_1 - \mathbf{p}_3) \times \mathbf{a}_3] \quad (\text{Eq. 3})$$

Notice that  $\mathbf{d}$  is a quadratic function of  $s$ . The locus of event viewpoints for an EEE event is a surface in 3-D made up of lines of sight intersecting all three edges; but since  $\mathbf{d}$  is a quadratic function of  $s$ , the lines do not lie in the same plane. The surface that results is a warped quadric surface. The intersection of such a surface with the viewing plane is a quadratic curve.

*Potential event points* are defined as points that are on lines of sight that start at a viewpoint  $v$  and intersect edges of faces in such a way that an event would be generated if no other face were occluding that line of sight. For EV events, potential event points form lines of sight on a plane, and for EEE events they form lines of sight on a warped quadric surface. This plane or warped surface is said to be the potential event surface. If any of these lines of sight is unoccluded by other faces in the scene, the intersection of that line of sight with viewpoint space is an event viewpoint.

In the computation of weak visibility events, the face occluded at an event is a face of the bounding box of one of the objects. The edges participating in the event are edges of walls. In order to construct the VSP in this case, it thus suffices to find all sets of three wall edges and an object face configured in a way that results in a potential event.

The corresponding event surface is intersected with all walls to find the lines of sight that reach the viewing plane unoccluded. The intersection of these surfaces with viewpoint space is a set of line segments (for EV events) or quadratic curve segments (for EEE segments) that partition viewpoint space. The set of faces visible from the initial viewpoint is computed using a standard hidden-surface removal algorithm such as that of Mulmuley [6].

The algorithm for constructing the conservative VSP is as follows. To handle EV-events, every set consisting of a wall edge and a wall vertex, or a wall edge and an object vertex, or a wall vertex and an object edge must be considered. Each set is tested to determine whether the items are configured in a way that results in a potential event by comparing the configuration with the list of potential event configurations. In the case of a wall edge and a wall vertex, the object face that appears or disappears at the event must also be found. Then for appropriately arranged sets, the parts of the potential event surface that intersect other walls of the scene are removed. The remaining lines of sight—those not blocked by any scene wall between viewpoint space and the faces participating in the event—intersect viewpoint space in conservative event viewpoints. The intersection will be a set of segments on a line. EEE events are handled similarly, except that the sets of object features that may form an EEE event are sets consisting of three wall edges or two wall edges and an object edge. In the case of three wall edges, the object face that becomes occluded at that event must be found. In this case, the VSP boundaries that result in viewpoint space are quadratic curve segments rather than line segments. Miller and Goldman [5] describe an efficient and robust method for finding the intersection of a quadric surface with a plane.

The above description of the algorithm does not say how to find the lines of sight of a potential event surface that do not intersect any walls. This may be done for  $w$  walls of bounded size in  $O(w \lg w)$  time using a data structure such as a segment tree [12]. More details of working with these surfaces are found elsewhere [9, 11]. The line segments and curve segments that result partition the plane into the conservative VSP. The event graph is used to represent the conservative VSP; it is a graph with edges for the line or curve segments and vertices for the intersections of these segments. At each vertex, the coordinates of the intersection point are stored, and at each edge are stored add and delete lists consisting of lists of pointers to objects and the constants representing the quadratic curve on which the segment lies, if it is a curve segment. Then, for a small change in viewpoint it is possible to determine the objects that become invisible or possibly weakly visible by checking whether the viewpath leaves the current cell, and if it does, updating the display list according to the add and delete lists for the cell boundary crossed. It is then necessary to check whether the viewpath leaves the next cell, and so on. If  $s$  segments are generated in computing conservative events, then for a 2-D viewpoint space the con-

servative VSP has size  $O(s^2)$  and can be constructed in  $O(s^2)$  time in the worst case, using the algorithm of Mulmuley [7]. However, for typical scenes with significant occlusion, the size and construction time are likely to be closer to linear.

The worst-case runtime for finding the segments constituting the VSP is  $O((w^4 + w^3 k) \log w)$ , where  $w$  is the number of wall vertices and  $k$  is the number of objects, since a computation taking time  $O(w \log w)$  is done for each potential event surface, of which there are  $O(w^3 + w^2 k)$  in the worst case. The preprocessing stage is thus combinatorially complex in the worst case. However, the time required depends only on the number of walls and objects selected and not on the size of the original model. This approach can therefore be used for models of any size. However, for a fixed number of walls and objects, the efficiency of the on-line part of the algorithm is likely to decrease as the model gets larger. Theoretically, the number of segments that result is  $O(w^3 + w^2 k)$  in the worst case. However, the worst case only occurs for pathological scenes such as a grid behind a grid; for typical scenes the amount of data is much smaller. But note that even partitioning a scene into just a few cells is likely to result in major improvements in on-line efficiency.

Practically speaking, the amount of space required and efficiency of this approach vary dramatically with the choices made for objects and walls and with the “visual complexity” of the scene (which is related to the number of objects and walls visible from a typical viewpoint). The more objects and walls, the more space required and the more efficient the result is likely to be in the on-line phase. The best that can be hoped for is that an efficiency close to one is achievable with a limited amount of storage, say approximately equal to the size of the original model.

## RESULTS

In order to determine the kinds of efficiency possible with limited storage for visual events, estimates of efficiency were made for a rough model of one floor of the Department of Computer Science at the University of Pittsburgh (see Figure 3). This model was created with Virtus Walkthrough, a commercial CAD program from Virtus Corp. Two typical views from within this model are shown in Figure 4.

Note that this model has many doors and windows, all open, but that there is still considerable opportunity for efficiency improvements by visibility pre-computation for an interactive walkthrough. The model has about 34 rooms and hallways and 131 pieces of furniture. There are 8895 faces in all. In all cases, the largest unbroken rectangular floor-to-ceiling sections of interior walls will be used as “walls” for occlusion tests. There are 128 such sections, or 64 if two-sided faces are used. Four partitions of scene faces into objects are considered: first, each face is an object; second, pieces of furniture and walls are objects; third, all the furniture in any room and the walls surround-

ing the room are objects, and fourth, all the faces in the scene constitute one object. Furniture always consists of one-sided faces, and walls are considered to be two-sided except in the case where rooms are objects.

In order to estimate the efficiency of the conservative visibility computation resulting from each partition of the scene into objects, 20 random viewpoints and viewing directions were generated. Then, for each viewpoint, the number of faces on the display list and the number that

must be rendered after clipping and culling were computed for each viewpoint by visual inspection. The results are summarized in Table 1. In computing these numbers, two assumptions were made: that the number of faces removed by backface culling is half of those on the current display list and that the portion of faces of an object removed by clipping when the object intersects the clipping boundary is one half. Also, the field of view is 90° and the front clipping plane is at the viewpoint.

**Figure 3. An example model. This is a floor of the Department of Computer Science at the University of Pittsburgh.**

**Figure 4. Two typical views from viewpoints in the model shown in Figure 3.**

	Object partition			
	each face	furniture	rooms	whole scene
Average number of faces to be transformed, clipped, and culled	471	546	1312	8895
Average number of faces to be rendered	68.5	68.5	196	1059
Transform/clip/cull efficiency	1	0.99	0.90	0
Render efficiency	1	1	0.87	0

**Table 1. Efficiency results for the sample model under for object selections.**

The numbers in the top two rows are highly variable: the standard deviation of these averages is roughly the size of the number in most cases. Therefore the confidence intervals for these averages are fairly large: plus or minus approximately half of the average in most cases. Furthermore, they represent averages for only a single model. Still, they are suggestive of the kind of efficiencies that may be obtained by these techniques with similar models.

Note that the use of even a few objects (34 in the case of rooms as objects) results in relatively high efficiency for this model. Only 15% of the faces in the scene must be transformed, clipped, and culled, on average, and only 19% of the faces must be rendered compared to the case where no visibility pre-computations are used. Using pieces of furniture as objects yielded nearly optimal results, and only 195 objects and 64 walls are used. It appears that for applications in which preprocessing time is available and rendering speed is important, this technique can yield excellent efficiency.

Lacking a complete implementation of the algorithm, I am unable to report on the amount of visibility data that would have to be represented in each case. However, since that information is important, I will attempt a rough estimate. I estimate that while the worst case number of segments in the VSP is  $O(w^3 + w^2k)$ , the average case for a model like a building is less than  $n^2$ , where  $n$  is  $\max(w, k)$ . I believe this to be the case because there are typically very few EEE-events in a building; most events are of the EV type, and the worst-case number of EV-events is  $O(w^2 + wk)$ . By that estimate, the number of regions in the VSP for the fourth case would be one. For the third case, where objects are rooms, the number would be less than 1,000. For the second case, where pieces of furniture and walls are objects, the number would be less than 40,000. For the first case, the number would be huge. Since VSP edges generally require less space to represent than scene faces together with their attributes, all but the first partition may result in a reasonable amount of visibility data.

#### DIRECTIONS FOR FURTHER RESEARCH

In models with a large number of objects, some of which are far from other objects, many or most of the objects in a scene may be far from the viewer. It is possible that from some viewpoints (say, near a window) many distant objects will be visible. If distances are such that these objects will be small in the image, significant savings are available by rendering simpler approximations to these distant objects. Thus, I am investigating the use of distance-based events and a hierarchy of progressively simpler and easier-to-render approximations to objects.

It is also possible to extend this approach to a 3-D viewpoint space. In that case, the viewer's movements are not constrained in any way, but the storage requirements for visibility information may be higher: a partition of a 3-D viewpoint space rather than 2-D must be stored. How-

ever, for many 3-D models such as buildings, it may be that the 3-D VSP is not much larger than the 2-D VSP because the geometry of the model in question is almost 2-D. The event-based approach to the walkthrough problem will also be practical for a 3-D VSP and any model as long as the number of objects and walls is limited. However, for given space and time limits, the efficiencies achieved will likely be lower.

#### RELATED WORK

Other researchers have also addressed the walkthrough problem. Brooks [3] uses the BSP-tree for walkthroughs. The BSP-tree makes it practical to work with much larger models when a Z-buffer is unavailable, but traditionally it has been used to find a back-to-front order of the polygons and not to reduce the number of polygons rendered at each frame. In fact, the number of polygons to be rendered at each frame can increase significantly. However, Gordon and Chen [4] have recently described a method for displaying the faces in a BSP tree in front-to-back order that is faster than the traditional back-to-front traversal when the number of polygons in the tree is large. Teller and Séquin [13] discuss visibility preprocessing for a model subdivided into rectangular cells and portals; their approach computes cell-to-cell visibility by finding sight-lines between cells. Airey *et al.* [1] address the walkthrough problem by precomputing the potentially visible sets of the model for sets of associated viewpoints or cells. They describe two approximation approaches to computing visibility from cells: point sampling and partial computation of shadows. They express the hope that a better algorithm can be developed in the future. Plantinga [11] gives an algorithm for computing the set of weakly visible faces of a scene from a polygon, but the time required to do so is  $O(n^4 \log n)$  in the worst case for a scene with  $n$  vertices.

Bern *et al.* [2] describe methods for computing visibility along a straight-line walk path over three different types of terrain. They describe a method for finding all topology changes along a line segment that requires worst-case time  $O((n^2 + p) \log n)$ , where  $n$  is the number of edges in the scene and  $p$  is the number of "transparent topology changes." (Here  $p$  is  $O(n^3)$ .) Mulmuley [8] gives a different algorithm for the same problem. These methods cannot be used when the walk path is not known in advance or is not a straight line. In that paper Mulmuley also defines an interesting data structure—a cylindrical partition  $H(A)$  for a scene  $A$  such that the view from any viewpoint can be computed in time proportional to the size of the complex  $\text{View}(v) \cap H(A)$  to within a log factor.  $\text{View}(v)$  is the star-shaped polyhedron consisting of the visible points from  $v$ . The size of  $H(A)$  is  $\Theta(n^2)$ , the amount of preprocessing time required  $\Theta(n^2 \log n)$ , and the view constructed  $\Theta(n^2)$  in size in the worst case. Although Mulmuley says that the typical sizes and runtimes are much closer to linear, this approach may still not prove practical for large scenes. However, some of the tech-

niques for handling large scenes introduced here may also be applicable to Mulmuley's approach.

## CONCLUSIONS

In this paper I present an approach to the walkthrough problem: group the primitives of the model into objects, select a subset of faces as walls, and find a partition of viewpoint space into cells for which the visibility status of each object is known (conservatively). At cell boundaries store lists of objects that become visible and invisible upon crossing that boundary; these objects are posted or unposted from the display structure when the viewpoint crosses a boundary. This approach admits a tradeoff between preprocessing time and space and on-line time: smaller, more numerous objects and more walls generate more cells and require more precomputation and more storage, but since the display list is potentially a smaller superset of the visible objects for any viewpoint, the on-line phase of the algorithm is faster.

In this approach it is possible to handle large scenes and curved surfaces. The partition of viewpoint space into cells is automatically generated for a choice of objects and walls. The amount of storage required is "tunable" for the specific characteristics of the application; more preprocessing time and more storage allow a more efficient on-line walkthrough. I believe that this approach or other approaches to conservative visibility computations will be widely applicable for situations in which precomputation time is available and interactive walkthroughs of large scenes are desired.

Finally, because processor speeds and memory capacities are increasing faster than bandwidth into frame buffers, I believe that object-space techniques for reducing the number of bits that must be sent to a frame buffer or Z-buffer will increase in importance. As the processor speed to frame-buffer bandwidth ratio increases, techniques such as the one presented here will yield a greater increase in efficiency per unit time invested.

## REFERENCES

1. Airey, J. M., J. H. Rholf, and F. P. Brooks Jr., "Towards image realism with interactive update rates in complex virtual building environments," *Computer Graphics* **24** (2), March 1990, pp. 41-50.
2. Bern, Dobkin, Eppstein, and Grossman, "Visibility with a moving point of view," SODA 1990.
3. Brooks, F., "Walkthrough—A dynamic graphics system for simulating virtual buildings," *Proc. Workshop in Interactive 3D Graphics*, 1986, pp. 9-21.
4. Gordon, D. and S. Chen, "Front-to-back display of BSP trees," *Computer Graphics and Applications* **11**(5), 1991, pp. 79-85.
5. Miller, J. R. and R. N. Goldman, "Using tangent balls to find plane sections of natural quadrics," *Computer Graphics and Applications*, **12**(2), March 1992, pp. 68-82.
6. Mulmuley, K., "An efficient algorithm for hidden surface removal," *Computer Graphics* **23** (3), 1989a, pp. 379-388.
7. Mulmuley, K., "A fast planar partition algorithm, II," *Proc. Fifth Ann. Symp. on Computational Geometry*, 1989b, pp. 33-43.
8. Mulmuley, K., "Hidden surface removal with respect to a moving view point (Extended abstract)," *Proceedings of the 23rd. Annual Symp. on Theory of Computing*, 1991, pp. 512-522.
9. Plantinga, H., and C. R. Dyer, "Visibility, occlusion, and the aspect graph," *International Journal of Computer Vision* **5**(2), November 1990, pp. 137-160.
10. Plantinga, H., W. B. Seales, and C. R. Dyer, "Interactive viewing of polyhedral scenes using viewpath coherence and the asp representation," *Graphics Interface '90*, pp. 9-16.
11. Plantinga, H., "An algorithm for finding the weakly visible faces from a polygon in 3-D," *Canadian Conference on Computational Geometry*, 1992, pp. 45-51.
12. Preparata, F. P., and M. I. Shamos, *Computational Geometry*, Springer-Verlag, 1985.
13. Teller, S. J., and Carlo H. Séquin, "Visibility preprocessing for interactive walkthroughs," *Computer Graphics* **25** (4), July 1991, pp. 61-70.