Dynamic Scene Occlusion Culling

Article *in* IEEE Transactions on Visualization and Computer Graphics · February 1999

DOI: 10.1109/2945.764866 · Source: IEEE Xplore

Craig Gotsman
Technion - Israel Institute of Technology
176 PUBLICATIONS **7,376** CITATIONS

Dynamic Scene Occlusion Culling

Oded Sudarsky and Craig Gotsman, *Member*, *IEEE*

Abstract—Large, complex 3D scenes are best rendered in an output-sensitive way, i.e., in time largely independent of the entire scene model's complexity. Occlusion culling is one of the key techniques for output-sensitive rendering. We generalize existing occlusion culling algorithms, intended for static scenes, to handle dynamic scenes having numerous moving objects. The data structure used by an occlusion culling method is updated to reflect the objects' possible positions. To avoid updating the structure for every dynamic object at each frame, a temporal bounding volume (TBV) is created for each occluded dynamic object, using some known constraints on the object's motion. The TBV is inserted into the structure instead of the object. Subsequently, the object is ignored as long as the TBV is occluded and guaranteed to contain the object. The generalized algorithms' rendering time is linearly affected only by the scene's visible parts, not by hidden parts or by occluded dynamic objects. Our techniques also save communications in distributed graphic systems, e.g., multiuser virtual environments, by eliminating update messages for hidden dynamic objects. We demonstrate the adaptation of two occlusion culling algorithms to dynamic scenes: hierarchical Z-buffering and BSP tree projection.

Index Terms—Occlusion culling, dynamic scenes, distributed multiuser virtual environments, hierarchical Z-buffer, octrees, BSP tree.

1 Introduction

INTERACTIVE three-dimensional computer graphics and animation have gained popularity over the last few years. However, their further acceptance is inhibited by the amount of computation they involve. To achieve realistic-looking images, scenes must be modeled in fine detail. The large models that result can take a long time to render, even when hardware support such as Z-buffering is available. If the scenes are dynamic, containing moving objects, then additional time is needed to update the model to reflect these objects' motions. This makes it hard to attain the goal of interactivity, which requires images to be rendered at a rate of about 10 frames or more each second.

Additional difficulties arise in distributed, client-server graphics systems. In such environments, a central server maintains a scene model, e.g., in VRML format [1]. A client that wishes to access the scene downloads the model into its local memory, then interacts with this local copy. If the model is large, it can take a very long time to download over a slow communication link, such as the Internet. In fact, the model may even be too big to fit entirely into the client, which is typically much weaker than the server. In dynamic scenes, if some of the dynamic objects are not controlled by the client but by another machine, more communications are needed to update these objects' positions. This can happen, for example, in scenes containing multiple robots on a factory floor, as in Tecnomatix's ROBCAD robotics simulator [2]. Another example is a multiuser virtual environment [3], [4], [5], [6], [7], where some of the dynamic objects are avatars, i.e., graphic representations of other users in the environment. Messages are sent between the client machines, possibly through a central server, to update each user as to the others' whereabouts.

Higher-performance hardware alone may be insufficient to solve these problems because, as we have witnessed over the past several years, enhanced performance is countered by higher software demands and user expectations. When hardware becomes available that allows, say, a furnished room to be modeled at sufficient detail and rendered with adequate speed, users will wish to interact with a whole apartment instead of a single room. If this becomes feasible, then entire buildings, neighborhoods, cities, or countries will be wanted. Hardware improvements typically provide only a constant-factor speedup, which may not suffice to meet this ever-increasing demand.

The solution lies in output sensitivity. An *output-sensitive* calculation is one whose runtime depends (usually linearly) only on the size of its output, not on the input. The number of samples required to render the image depends only on its resolution (and on any multisampling scheme in effect), not on the model's complexity. Therefore, the number of scene primitives that can contribute to the image is limited and output-sensitive rendering can (at least theoretically) deal with arbitrarily large scenes. Similarly, due to its fixed size, the image can only be affected by a limited portion of the model and a limited number of dynamic object motions. Therefore, output-sensitivity can reduce the amount of information that needs to be communicated from server to client.

Output-sensitive rendering is achieved by two complementary techniques: occlusion culling [8], [9], [10], [11], [12], [13] and level-of-detail control [14], [15], [16]. *Occlusion culling* (also called visibility culling or output-sensitive visibility calculation) finds the visible parts of the scene without wasting time on the occluded parts, not even to determine that they are occluded. Occlusion culling techniques usually perform view frustum culling, too: Parts of the scene that are outside the field of view are also discarded quickly. *Level-of-detail control* replaces parts of the scene model that subtend a very small screen-space area with simpler objects that are faster to render.

• *O. Sudarsky and C. Gotsman are with the Department of Computer Science, Technion—Israel Institute of Technology, Haifa 32000, Israel. E-mail: {sudar, gotsman}@cs.technion.ac.il.*

For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number 108186.

Occlusion culling is generally effective in densely occluded scenes, for example, indoors, where a lot of the model is hidden. On the other hand, level-of-detail control contributes mostly in wide-open, sparsely occluded situations, such as outdoor scenes, in which many of the objects are seen at a distance. In practice, general scenes require a combination of both techniques. Outdoors, one might see the outsides of some complex buildings, most of whose inner structure is occluded; indoors, open landscape may be seen through windows.

Most research into level-of-detail control deals with the automatic generation of multiple representations of an object at different resolutions. Given a high-resolution polygonal representation or a freeform description of an object, one approach is to create less detailed versions of the object as a preprocessing stage. Once these versions have been generated, the appropriate one is chosen, at runtime, based on the distance to the viewer. To avoid abrupt transitions between consecutive levels of detail, interpolation is sometimes performed between the levels. An alternative approach is to construct the required simplified object at runtime, based on the viewer's current position. Thus, the object's farther parts are rendered more roughly than the nearer parts. See the surveys of polygonal simplification techniques by Luebke [17] and by Heckbert and Garland [18].

Occlusion culling algorithms typically use some spatial hierarchical data structure, e.g., an octree or a binary space partitioning tree (BSP tree), to subdivide space into a hierarchy of volumes. This structure is built as a preprocessing stage, and each scene object is associated with the volumes it lies in. At runtime, the visible parts of the scene are discovered by traversing this structure in a top-down, near-to-far order, discarding each volume that is completely obscured by objects that have been rendered so far during the traversal. If this volume is the root of a subtree that contains many scene objects, then all these objects are eliminated wholesale, without specifically testing the visibility of each and every one. Various occlusion culling algorithms differ mostly by the type of spatial data structure they use and by their method of finding whether a volume is obscured.

Current occlusion culling algorithms are inappropriate for dynamic scenes. It usually takes much longer to construct the spatial hiearachical data structure than to render the scene from any single viewpoint. Therefore, the structure is built at preprocessing, under the assumption that all the objects are static. Thus, the only interaction or animation sequences that existing occlusion culling algorithms allow are walk-throughs or fly-throughs, where the entire scene is static and only the viewer moves through it.

It is possible, in principle, to update the data structure to reflect the dynamic objects' movements. However, if this update is not done carefully, it might take far too much time—longer than the naïve rendering of all the scene objects. In particular, the update should be avoided for dynamic objects which are currently hidden, otherwise the goal of output sensitivity will not be attained. On the other hand, one cannot totally ignore occluded dynamic objects, because they may become visible at some later time.

What is needed is a mechanism to ignore hidden dynamic objects most of the time, yet be notified when they might no longer be hidden. Such a mechanism can eliminate not only the time it would otherwise take to render hidden dynamic objects or to maintain a spatial data structure according to their motions: It can also save the time to update the hidden objects' positions and configurations. This can constitute a great saving in a distributed graphic environment, where dynamic objects are controlled by remote machines, because the eliminated update messages would have been sent through a relatively slow communication link. This will be discussed in Section 3.7.

We present a method to achieve this avoidance of computation for hidden dynamic objects. It is based on the observation that the possible movements of dynamic objects may be subject to some known constraints; such constraints may be imposed, for example, by physical simulation or by a user interface. Given these constraints, occlusion culling algorithms can be adapted to scenes containing multiple dynamic objects. This adaptation is shown for two algorithms: Greene et al.'s hierarchical Z-buffer [10] and Naylor's BSP tree projection [9]. Experimental results are given for both.

2 Related Work

2.1 Static Scene Occlusion Culling

The earliest known mention of occlusion culling was by Clark [19]. He proposed using a hierarchical object representation, with a bounding sphere and a simply-shaped bounded volume at each node of the hierarchy. If the bounded volume of one node obscures the bounding sphere of another, then the subtree of the obscured node need not be rendered. This utilizes only occlusions by a single node, not accumulated occlusions by several nodes. No implementation or empirical results were reported.

Meagher [20] developed an occlusion culling algorithm for volumetric data. In many ways, this algorithm resembles hierarchical Z-buffering, described below.

The computational geometry community has produced some occlusion culling algorithms for polygonal models, but they are too limited and complicated to be practical. For example, de Berg and Overmars' algorithm [21] is restricted to polyhedral scenes in which there is a finite number of possible orientations of the polyhedra, and these orientations are known a priori.

Teller, Séquin, and Funkhouser's technique [4], [8], [22], [23] is more suitable for implementation, but is intended specifically for indoor architectural scenes. Their approach is to construct a *k*-D tree for the scene (assuming it is composed of rectilinear surfaces), and to precompute the set of potentially visible cells (leaf nodes) from each cell of the tree. During rendering, one may safely ignore all geometry outside the cells which are visible from the one containing the viewpoint. Further improvement is attained by view frustum culling.

The restriction of the above technique to rectilinear planes can easily be lifted by using BSP trees instead of *k*-D trees. (This has indeed been done by Teller and Hanrahan [24], but their objective was efficient calculation of form factors for radiosity.) A more serious drawback of Teller et al.'s method is that the length of time required to calculate the cell intervisibility relationships, and the amount of memory needed to store the relationships, can be quadratic in the number of cells (in addition to any calculation time and storage required for the *k*-D or BSP tree itself). Luebke and Georges [25] avoid this drawback by conservatively computing overestimates of the intervisibility relationships at run time, using screen-space bounding rectangles of object-space portals between the cells. However, they require the cells and the portals for the scene to be defined manually. This manual intervention step does not permit models to be preprocessed for occlusion culling in an automatic batch process.

Two practical and general algorithms are Greene et al.'s hierarchical Z-buffer algorithm [10] and Naylor's BSP tree projection method [9]. We describe them in more detail in the following sections, because our techniques build on these methods. Like Teller and Séquin, Greene et al. and Naylor use hierarchical data structures to subdivide object space. This is a common property of many output-sensitive visibility algorithms: They employ a hierarchical spatial data structure to quickly cull large, occluded regions of space, without explicitly considering every object within those regions. However, the spatial data structure does not have to be a hierarchy in the strict sense of the word. For example, it may be a directed acyclic graph, and sibling nodes do not have to represent disjoint regions of space.

Another common characteristic of occlusion culling algorithms is that they also perform view-frustum culling: Subtrees of the data structure outside the field of view are also eliminated. This provides further acceleration, albeit by an expected constant factor. Through the rest of this paper, we refer to objects as hidden whether they are occluded by other objects or out of the view frustum.

2.1.1 Algorithms Using Auxiliary Spatial Hierarchical Data Structures

The hierarchical Z-buffer algorithm [10] is based on the ordinary Z-buffer, but uses two hierarchical data structures: an octree and a Z-pyramid. The lowest level of the pyramid is a plain Z-buffer; in all other levels, there is a pixel for every 2 × 2 square of pixels in the next lower level, with a value equal to the greatest (farthest) z among these four pixels.

At the algorithm's initialization stage, an octree is constructed for the entire model. This operation is too time-consuming to be done every frame, and takes much longer than just calculating visibility from a single viewpoint; however, assuming the model is static, the same octree can be used to calculate visibility from many different viewpoints.

To calculate visibility from a viewpoint, the Z-pyramid is first initialized to ∞ at all pixels in all levels. Then, recursively from the octree's root, each encountered octree node is checked for occlusion by the current contents of the Z-pyramid. If a node is totally hidden, it can be ignored; otherwise, the primitives directly associated with the node are rendered, the Z-pyramid is updated accordingly, and the eight child nodes are traversed recursively, from near to far. Because of this front-to-back order, there is a good chance that farther nodes will be discovered to be occluded by primitives in nearer ones, thus saving the handling of all the subtrees associated with the farther nodes.

The pyramid is used for fast visibility checking of nodes and primitives: Find the lowest pyramid level where a single pixel still covers the entire projection of the primitive or node. If the z value registered at that pixel is closer than the closest z of the projection, then the entire primitive or node is invisible. Otherwise, the projection is divided into four, and checked against each of the four corresponding pixels in the next lower level. See Fig. 1.

A more recent version of the hierarchical Z-buffer algorithm [26] uses an image-space quadtree instead of a Z-pyramid. This worsens performance to some extent, but enables effective antialiasing. Later still, Greene [27] introduced another algorithm, that uses bitwise operations between precalculated coverage masks to find the image area occluded by a convex polygon. However, this algorithm is more complicated than hierarchical Z-buffering, because it requires a complete depth sort of the model's polygons; therefore a BSP tree must be constructed inside each octree leaf.

The main drawback of the hierarchical Z-buffer algorithm is its reliance on a Z-pyramid. Updating the pyramid in software is too inefficient; reading the Z-buffer from hardware (to build the pyramid on top of it) takes too long on most platforms. Even once the Z-pyramid is available, too much overhead is involved in testing whether it occludes a given octree node (answering the so-called "Z query"). Because of these overhead factors, Greene et al. report that the break-even point for the hierarchical Z-buffer algorithm, compared to ordinary Z-buffer rendering, is about 3 sec per animation frame for a 512 × 512 pixel

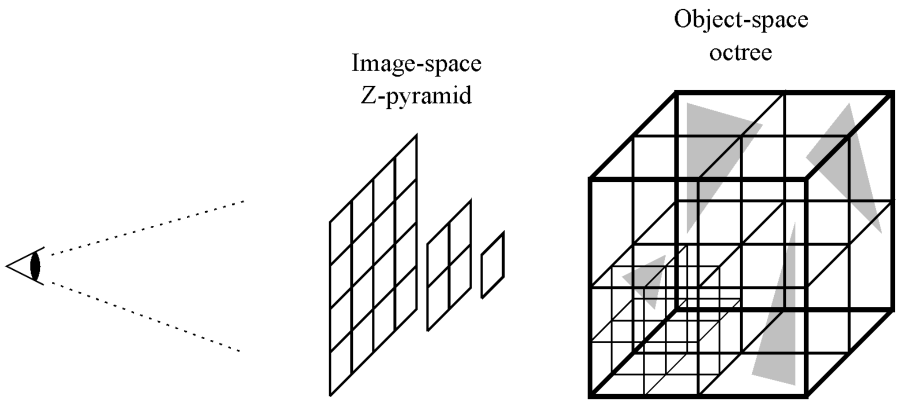


Fig. 1. The hierarchical Z-buffer algorithm.

image. This means that, given current hardware performance, the hierarchical Z-buffer algorithm does not allow interactive speeds; this has been verified by our own experiments. One possible long-term solution is to construct specialized graphics hardware, probably involving an internal Z-pyramid, that answers Z queries efficiently, in time which is constant and independent of the queried object's projected image size. However, such hardware has yet to become available. In fact, Z query support at *any* speed is rare in graphics hardware, presumably because answering such queries would require flushing the graphics pipeline, which would slow down rendering.

While the hierarchical Z-buffer algorithm is primarily intended for static scenes, Greene's PhD thesis [28] mentions a few ideas for handling dynamic objects. However, none of them has been implemented or seriously explored. The idea most closely related to our method (see Section 3.2) is to exploit limited range of object motion: If an object moves at most some fixed distance each frame, then, instead of associating each of the object's primitives with the smallest containing octree leaf, all the primitives can be associated with the smallest octree node that contains a bounding volume for the object for *k* frames (for some fixed *k*). This approach differs from ours because it does not distinguish visible dynamic objects from occluded ones: Both are updated in the octree once every *k* frames. Therefore, the octree update time is merely reduced by a constant factor of *k* (relative to updating the octree for every dynamic object at each frame). Furthermore, this approach causes dynamic objects to be associated with relatively big octree nodes; consequently, they are more likely to be needlessly rendered even when they are occluded.

When rendering a static scene, the cost of updating a data structure representing the occlusions (e.g., a Z-pyramid) for each rendered primitive can be mitigated by updating it only for a minority of the primitives—those that are expected to be good occluders from the current viewpoint. The set of potential occluders from each region of space, i.e., scene polygons that appear relatively large from viewpoints inside the region, can be found at preprocessing. This set can be further reduced at runtime by considering the actual position of the viewpoint within the region. The algorithms due to Coorg and Teller [11] and Hudson et al. [12], [13] are based on this idea. They are *conservative*, meaning they detect only some of the occlusions: They never report a visible node of the spatial data structure as invisible, but may erroneously classify some invisible nodes as visible. Consequently, some hidden regions of space are needlessly traversed. Unlike hierarchical Z-buffering, these algorithms might fail to take advantage of accumulated occlusions by multiple small primitives. For example, in a brick wall, each brick does not obscure much, but if all the bricks are disregarded as occluders then the occlusion by the entire wall is not utilized. Another disadvantage is that the preprocessing stage which finds the set of potential occluders takes some time and space, and it has to be repeated if any of the scenery changes, e.g., by users doing construction work in a shared virtual environment.

Coorg and Teller [11] estimate visibility by observing the position of the viewpoint relative to *supporting* and *separating* planes, i.e., planes which include an edge of one polyhedron and a vertex of another. This analysis is only guaranteed to correctly detect occlusions by a single convex polygon or by a mesh of edge-connected polygons with a convex silhouette.

Hudson et al. presented two different occlusion culling algorithms. The first [12] performs geometric analysis to eliminate spatial data structure nodes that are in the "shadow frustum" of a single occluder; it does not detect acummulated occlusions by multiple occluders. The second algorithm [13] takes advantage of graphics hardware capabilities to construct a hierarchical occlusion map (HOM): a set of gray-level raster images with different resolutions, in which the brightness of each pixel corresponds to the degree to which the pixel is covered by occluders. Spatial data structure nodes are tested for occlusion by comparing their screen-space bounding rectangles against the HOM and against a depth estimation buffer (a coarse Z-buffer of the occluders). For dynamic scenes, both the potential occluder set and the spatial data structure are omitted, and object bounding boxes are used instead. Due to the omission of the hierarchical data structure, this method is not expected to be output-sensitive for dynamic models of increasing size. Furthermore, it makes rather strong assumptions on the capabilities of the graphics hardware.

2.1.2 The BSP Tree Projection Algorithm

Naylor's projection algorithm performs output-sensitive visibility calculation using the same principle as the hierarchical Z-buffer algorithm: elimination of large parts of the model at an early stage of the calculation, using a data structure constructed at preprocessing time. However, Naylor uses more sophisticated data structures: BSP trees.

A BSP (binary space partitioning) tree [29], [30] can be defined in any number of dimensions. It is a binary tree in which each node represents some hyperplane; the left subtree of the node corresponds to the negative half-space of the hyperplane, while the right subtree corresponds to the positive half-space. For example, in the 2D case, each node represents a line, and each subtree represents a region in the plane. See Fig. 2.

In the 3D case, a BSP tree is a proper generalization of an octree: The planes dividing each node do not have to be in the middle of the node, and are not necessarily axis-parallel. In fact, if the model consists entirely of polyhedrons, then the

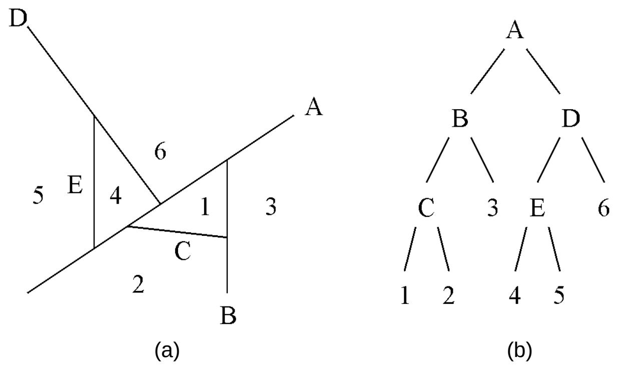


Fig. 2. (a) A binary partitioning of the plane, with lines denoted by letters and regions by numbers; (b) the corresponding 2D BSP tree.

BSP tree is general enough to accurately represent the scene itself, without need for any additional data structure; a Boolean "in/out" attribute is simply maintained with each leaf node, such that the value of the attribute is "in" if the corresponding region of space is inside a polyhedron, "out" otherwise. This is in contrast to an octree, which usually serves only as an auxiliary data structure in computer graphics, and not as a representation of the model itself.

Naylor [9] advocates the use of 2D BSP trees to represent and manipulate images; the trees are to be scan-converted into raster images only as a last stage, for actual display. He presents an algorithm to project a 3D BSP tree, representing a scene model, into a 2D BSP tree that describes the scene's image. The algorithm transforms the input tree into the output tree by traversing the tree recursively, from near to far. During the traversal, whenever the algorithm encounters a face of the model, it constructs a small BSP tree that represents the volume occluded by the face from the current viewpoint. It then unites this small tree into the overall BSP tree (see Fig. 3). The union operation [30] eliminates redundant nodes: Every subtree that has identical "in/out" attributes at all the leaves is collapsed to a single leaf. Thus, the entire region of space occluded by the model face is discarded and replaced by one "in" leaf, regardless of the complexity of the BSP tree in that region prior to the face's projection.

The construction of BSP trees from boundary-represented (B-rep) polygonal models involves a heuristic choice of partitioning planes. Naylor [31] presents heuristics that construct the tree as a series of approximations to the modeled object, yielding minimum expected cost for various operations on the resulting trees (e.g., Boolean set operations). The BSP tree projection process is output-sensitive if its input tree was constructed according to these heuristics. Output sensitivity is achieved for the same reason it is attained in the hierarchical Z-buffer algorithm: wholesale elimination of large, hidden parts of space, without specific examination of each object in these parts.

Contrary to algorithms based on auxiliary spatial data structures, e.g., hierarchical Z-buffering, Naylor's projection algorithm needs no further data structures beyond those representing the model and the image. Again, the construction of the hierarchical spatial data structure (in this case, the 3D BSP tree) is very time-consuming; but it is only constructed once, as a preprocessing stage, and subsequently used for visibility calculation from many different viewpoints.

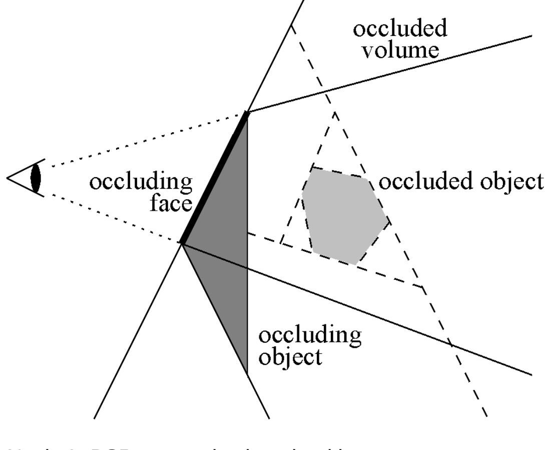


Fig. 3. Naylor's BSP tree projection algorithm.

2.2 Spatial Hierarchical Data Structures of Dynamic Scenes

Several techniques have been proposed to maintain spatial hierarchical data structures of dynamic scenes. These scenes can subsequently be displayed by an occlusion culling algorithm. However, this combination will not be output-sensitive with respect to the number of dynamic objects: Each such object will be updated every time the scene is displayed, even if the object is hidden. That is, the runtime of these techniques, followed by occlusion culling, is $\Omega(v + i)$ per frame, where v is the number of visible objects, whether static or dynamic, and i is the number of invisible dynamic objects. We would like to achieve $O(v + f(i))$, where f is a sublinear function such that $f(x) \ll x$ for sufficiently large x, e.g., $f(x) = \log x$. For such performance, the data structure update must be closely coupled with occlusion culling.

Octree update methods have been presented by Ahuja and Nash [32] and by Weng and Ahuja [33]. Chrysanthou and Slater [34], [35] and Agarwal et al. [36] have proposed algorithms to maintain dynamic BSP trees. However, their trees are of the wrong kind for occlusion culling purposes: They keep the objects' surfaces as B-rep polygons at the trees' inner nodes, rather than keeping the objects' interiors as leaf "in/out" attributes. The scheme proposed by Torres [37] can be used with either brand of BSP trees: He suggests associating the higher levels of the tree with planes that separate between objects, thus allowing more efficient updates.

A simple method, based on work by Naylor et al. [30], [38], is to keep each object's BSP tree separately, and to construct the tree for the entire scene by uniting these trees, transformed to their proper positions and orientations. This can be done quickly by copying pointers to BSP tree nodes rather than entire subtrees, and avoids the need to update a BSP tree due to dynamic objects' motions. While this method may not be optimal, it can be combined relatively easily with the output-sensitive BSP tree projection algorithm to achieve dynamic scene occlusion culling. This is discussed in Section 5.

2.3 Message Reduction in Virtual Environment Systems

As mentioned in the introduction, and as Section 3.7 elaborates, a major advantage of our dynamic scene occlusion culling techniques is that they can also eliminate a significant number of unnecessary messages—those messages that would otherwise be transmitted to update hidden objects. These techniques may be used instead of, or in addition to, other methods to reduce the required number of update messages in shared virtual environments. The other methods include decomposition into cells, multicasting, dead reckoning and visibility precalculation.

The virtual environment may be decomposed into separate regions or cells; each user receives messages only from users which are in his cell (and possibly in immediately neighboring cells). NPSNET [5] uses this method with 4 km hexagonal cells, Active Worlds [39] uses 10 × 10 m cells, and Spline [6] allows arbitrary polyhedral cells given as BSP trees. The disadvantage of this method is that the display is correct only in the cell that contains the viewer and its immediate neighborhood; if the cells are relatively small, as in Active Worlds, this results in an incorrect image. If the cells are enlarged to minimize this effect, as in NPSNET, then the user's station is flooded with messages from many other users. In Spline, the model is constrained such that there is no direct line of sight between non-neighboring cells. This limitation requires the model to be distorted to match the constraint, e.g., by using winding corridors or airlock-style doors.

Spline, NPSNET, and DIVE [3], [7] use a message multicasting protocol, so when a user updates other users of his movement, he sends only one message, regardless of how many users receive it. In Spline and NSPNET, each cell has a different multicast channel associated with it; each user sends update messages on the channel associated with the cell he is currently in. The drawback is that each user receives and processes numerous update messages from other users. For example, in NPSNET, a process participating in a multiuser simulation spends most of its time filtering irrelevant messages.

NPSNET and its predecessor DIS/SIMNET use dead reckoning to further reduce the number of required messages: Instead of a user updating the other users of his position at every moment, the other users perform second-order extrapolation of his position, based on his prior behavior. The user himself performs the same approximation, and sends his updated position when the extrapolated position differs from the real one by more than some threshold (or after some fixed time limit).

In the RING system [4], a *k*-D tree of the model is constructed as a preprocessing stage; the model is usually of a building interior, and the leaves of the *k*-D tree generally represent the rooms in the building. Also at preprocessing time, the intervisibility relationship between the leaves of the tree is calculated, and stored at the leaves. At runtime, the system only transmits messages between users that potentially see each other because they are located in intervisible rooms. This method is effective mainly in densely occluded indoor scenes, and it does not utilize occlusions by dynamic objects. For example, a user in an office may be notified of the movements of many users in other offices down the hall, even though none of them are visible to him because his view of the office door is currently blocked by another user's avatar.

3 Our Approach

3.1 Problem: Dynamics

Existing occlusion culling algorithms are unsuitable for dynamic scenes because they rely on a large, complex spatial data structure, usually hierarchical. This structure is built in a compute-intensive preprocessing stage, under the assumption that the scene objects are stationary. While the existing algorithms allow the exploitation of temporal coherence in animation sequences, these are restricted to walkthrough animations, in which the scene is static and only the viewpoint moves through it. If anything other than the viewpoint moves in the scene, then the data structure used by the occlusion culling algorithm becomes outdated and incorrect images may result.

Obviously, it is out of the question to reconstruct the data structure each time an object moves in the scene, as this would be much slower than just displaying everything by the plain Z-buffer algorithm. The existing data structure should be updated for the dynamic objects' motions, rather than being initialized from scratch. However, if this update is performed for every object movement, then the overall algorithm will not be output-sensitive: It will waste time on updating the structure for occluded dynamic objects as well as for visible ones.

To preserve output-sensitivity, the update should only be performed in the visible parts of the data structure. Objects moving in other regions should somehow be ignored. The occlusion culling algorithm that runs on the data structure can report which of the objects it encountered in the last frame displayed; all the other objects were hidden. However, the update of the structure for these unseen objects cannot be simply omitted for the next frame, because they might become visible in the interim. Furthermore, if a dynamic object is naïvely ignored from the moment it is found to be hidden, then it might not be displayed again when it should be. This can happen because the culling algorithm does not traverse the entire data structure, but only its observable part; since the object's position in the structure would remain outdated, the algorithm might miss it. In fact, if the object's last known position remains occluded, then the object will never be seen again.

3.2 Solution: Temporal Bounding Volumes

Our solution is a variation on the *lazy evaluation* principle: Do not compute anything until it is necessary. We avoid wasting time on updating the spatial data structure for hidden dynamic objects, yet circumvent the above-mentioned problems, by employing temporal bounding volumes (TBVs). A TBV is a volume guaranteed to contain a dynamic object from the moment of the TBV's creation until some later time. This time is called the TBV's "expiration date," and the length of time from the TBV's creation until the expiration date is its "validity period."

TBVs are based on some known constraints on the objects' behaviors. For example, such constraints can be physically-based, e.g., nonpenetration of solids, or they may be imposed by the user interface of an interactive virtual environment. For objects moving along preset trajectories, the TBVs can be sweep surfaces (see Fig. 4); if only maximum velocities or maximum accelerations are known, then spheres may be used. Generally, we assume some bounding volume can be found for each dynamic object until some moment in the future.

TBVs are inserted into the data structure in lieu of dynamic objects which the occlusion culling algorithm deems invisible. Subsequently, a hidden dynamic object is ignored until such time as its TBV expires or until the culling algorithm determines that the TBV is visible, whichever occurs first. The former event means the TBV is no longer guaranteed to contain the object; the latter implies that the TBV is visible, therefore, the object itself

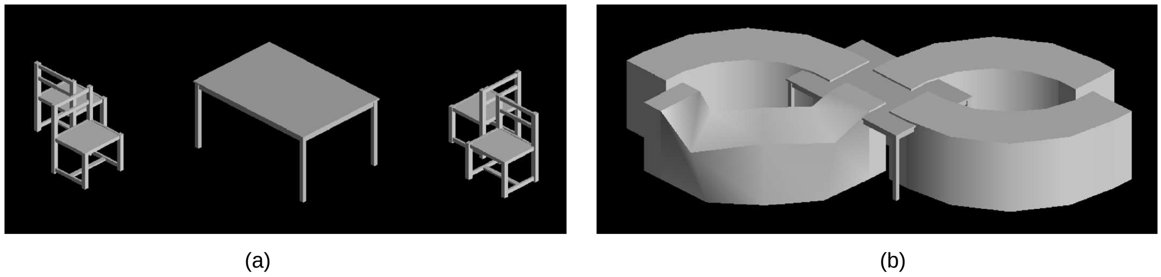


Fig. 4. (a) A scene with dynamic objects (chairs) moving along preset trajectories; (b) sweep surface temporal bounding volumes for the chairs moving towards the table. Note that the closer chair on the left wobbles during its movement.

might be visible too. In either case, the object should be considered again, because it is no longer guaranteed to be occluded. A priority queue of TBV expirations, similar to event queues used in simulation, notifies when TBVs cease to be valid. As long as expiration dates are chosen with sufficient care (see Section 3.4), most volumes remain invisible throughout most of their validity periods. Thus, in every frame, the majority of the hidden dynamic objects is ignored, and output sensitivity is maintained.

This approach is specifically tailored to visibility calculation. It involves more than merely plugging an updatable spatial data structure into an occlusion culling algorithm, as discussed in Section 2.2. In our method, data structure updates and occlusion culling are tightly interleaved: The culling algorithm determines which parts of the structure need to be updated. This enables output sensitivity to be achieved with respect to the number of dynamic objects.

Note that the TBVs are calculated on the fly, and that prior knowledge of the objects' trajectories is not essential. Thus, compatibility is assured with interactive applications in which this information is not available in advance, such as simulations, games, and virtual reality.

3.3 Algorithm

The exact formulation of the dynamic scene occlusion culling algorithm depends on the underlying (static-scene) occlusion culling technique $\mathcal{A}$. In particular, it is influenced by the spatial data structure employed. Nevertheless, the general algorithm can be described as follows.

Define these data structures:

T: the spatial data structure used by $\mathcal{A}$, with a set of dynamic object IDs in each node.

D: a set of all the dynamic objects, each having a unique identifier (ID), a time of last observation, and a flag stating whether the object is known to be hidden or is (potentially) visible. The hidden dynamic objects also have a TBV, an expiration time for the TBV, and a set of pointers to nodes of **T** associated with the TBV.

Q: an event queue of TBV expirations.

V: a set of IDs of visible dynamic objects.

A node of **T** has an object's ID if the object is visible and associated with the node, or if the object is hidden and its TBV is associated with the node. This association depends on $\mathcal{A}$. For example, objects can be associated with the nodes whose spatial region they intersect. Note that visible dynamic objects do not have node pointers, although nodes of **T** do have IDs of both hidden and visible objects.

All the dynamic objects in **D** are initially marked as visible, and have a time of last observation earlier than the first frame. **T** is initialized to represent all the static objects, **Q** is empty, and **V** initially contains the IDs of all the dynamic objects. If the algorithm is used in a multiuser virtual environment, then these data structures are maintained at each user's workstation. Each of the dynamic objects can be either an avatar of some other user or an autonomously moving object, controlled by a program, e.g., a Java-controlled object in a VRML 2.0 model.

The algorithm uses two subroutines: del_TBV deletes a hidden dynamic object's temporal bounding volume, and changes the object's status from hidden to (potentially) visible; upd_vis updates the scene for a visible dynamic object.

```
del_TBV(ID):
1) Delete ID from the nodes of T that contain ID;
2) Delete the dynamic object identified by ID from T;
3) Empty the object's set of node pointers;
4) Mark the object as potentially visible;
5) $\mathbf{V} \leftarrow \mathbf{V} \cup \{\mathrm{ID}\}$.
```

In Step 1 of del_TBV, the nodes of **T** which contain ID are given by the set of node pointers maintained with the corresponding dynamic object in **D**. Steps 1 and 2 may actually be performed simultaneously, because the deletion of Step 2 can start at the nodes from which ID is deleted in Step 1.

```
upd_vis(ID):
1) Obtain the current configuration of the dynamic object identified by ID;
2) Update T to reflect the object's current configuration;
3) Remove ID from the object identifier sets of those nodes of T that $\mathcal{A}$ no longer associates with the object;
4) Add ID to the object identifier sets of the nodes of T that are now associated with the object.
```

In Step 1 of upd_vis, the object's up-to-date configuration can be obtained, for example, through a communication link. The exact content of the message depends on the dynamic object's nature. For instance, if the object is rigid, then all that needs to be sent are translation and rotation parameters, e.g., as a 4×4 homogeneous transformation matrix. For an articulated object, a transformation is needed for each rigid segment. For a deformable object, the communication can include some deformation parameters, or a complete description of the object's current form. Steps 3 and 4 of upd_vis may, in fact, be done in parallel with Step 2, because the nodes from which ID needs to be removed or added are encountered during the update of Step 2. If the object was previously hidden, then Step 2 amounts to insertion of the object into **T**, and Step 3 can actually be omitted.

At every frame, the following steps are performed:

1) For each object ID of an expired TBV, as determined by **Q**, do del_TBV(ID).
2) For each ID in **V** do upd_vis(ID).
3) Operate $\mathcal{A}$ on **T**, displaying visible objects. At each node of **T** that $\mathcal{A}$ traverses, for each object ID in the node's dynamic object identifier set, if the object is marked as hidden and its TBV is visible, then do:
   a) del_TBV(ID);
   b) upd_vis(ID).

   For each dynamic object ID in the node's object identifier set, if the object is visible then update its time of last observation to be the current frame's time.
4) For each dynamic object in **V** whose time of last observation is earlier than the current frame, do:
   a) Delete the object's ID from **V**;
   b) Mark the object as hidden;
   c) Obtain a TBV for the object until some time in the future;
   d) Insert the TBV into **T**, add the object's ID to the nodes of **T** associated with the TBV, and add pointers to these nodes into the object's set of node pointers;
   e) Insert the TBV's expiration event into **Q**.

In a multiuser virtual environment, these steps are performed by each user's workstation.

Step 1 of the algorithm handles hidden dynamic objects whose TBVs have expired by deleting their TBVs and moving them to the set **V**. Note that **V** contains *potentially* visible dynamic objects, rather than objects which are definitely visible; at this stage it is too early to determine certain visibility, so the objects whose TBVs have expired are simply bundled with those that are potentially visible because they were visible in the previous frame.

Step 2 updates **T**, which is used in Step 3, the heart of the algorithm.

Step 3 displays the scene, and handles exposed TBVs by treating them the same way they would have been handled had they expired rather than becoming visible. This step requires special care, because it traverses a hierarchical data structure and modifies it at the same time. For example, if the actions del_TBV and upd_vis can cause changes in the structure of **T**, then these changes may have to be buffered and postponed until the end of the step. Step 3 can then be repeated for any new subtrees created by these changes. Alternatively, if the step is not repeated for the new subtrees, some of the dynamic objects which have just become visible might not be shown until the next frame. This inaccurate display may be quite acceptable, as it only lasts for one frame. If it is unacceptable, slightly bigger TBVs can guarantee accurate results by ensuring that a TBV is exposed at least one frame before its dynamic object.

Finally, Step 4 handles dynamic objects which cease being visible. In Step 4, item c), an object's TBV is provided by whoever controls the object (e.g., another workstation), upon request from the algorithm; the request should specify the expiration date, i.e., the time in the future until which the provided TBV should be valid. The choice of expiration dates is discussed in Section 3.4.

Note that the algorithm performs fewer calculations due to the TBV of a hidden dynamic object than it would perform due to the object itself, both because the TBV does not have to be updated at every frame and because it can be geometrically simpler than the object itself (e.g., it can be just a sphere or an axis-aligned box). While the size of the priority queue **Q** is linear in the number of invisible dynamic objects i, the time to perform each operation on **Q** is sublinear. For example, if **Q** is implemented as a skip-list [40], then the expected time for each TBV insertion is $O(\log i)$, and the time for each TBV deletion is $O(1)$ if the deletion is due to the TBV's expiration, $O(\log i)$ if it is due to exposure.

Output-sensitivity with respect to the number of dynamic objects is achieved by ignoring such objects unless they are potentially visible. While the algorithm involves some overhead, its amount does not depend linearly on the size of the entire scene. For most frames, no time is wasted on updating and displaying obscured dynamic objects, not even to discover that they are obscured; they are simply not reached during the traversal of **T**. Hidden dynamic objects require processing only if their TBVs expire or become exposed. If TBV validity periods are chosen with sufficient care (e.g., if adaptive expiration, discussed in the next section, is used), then TBV exposures occur only for a minority of the invisible objects and TBV expirations become less frequent with time. Thus, performance is sublinear in the number of hidden dynamic objects. See Section 3.5 of Sudarsky's PhD thesis [41] for a more formal analysis.

As for space complexity, in addition to the storage required by the underlying occlusion culling technique $\mathcal{A}$, most of the storage is used to maintain the associations between dynamic objects and nodes of the spatial data structure **T**. In the worst case, the known constraints about the dynamic objects' movements are so weak that most TBVs occupy most of the scene space, therefore, the space complexity equals the number of dynamic objects times the number of nodes in **T**. However, under more usual circumstances, the motion constraints are expected to be more useful, yielding tighter TBVs which lead to much smaller memory requirements. The exact space complexity is application- and scene-dependent.

Static objects may, in principle, be regarded as dynamic objects with zero velocity. However, visible dynamic objects are updated at every frame, and it would be undesirable to "update" visible static objects in the same way, especially if this update takes place over a slow network (even though relatively few of the static objects may be visible). Therefore, the algorithm treats static objects differently from dynamic ones.

3.4 TBV Validity Periods

Once the culling algorithm has determined that a dynamic object is occluded, there remains the problem of choosing the right validity period for its TBV. If the chosen expiration date is too soon, the dynamic object will have to be considered again before long, thus decreasing efficiency. On the other hand, if the date is too far in the future, then the bounding volume is too big and loose around the object, and, therefore, it might become visible after a short time, again hindering performance. A TBV with an optimal validity period would remain occluded for a maximum length of time.

If the viewpoint is stationary, and most of the occlusions in the scene are by static objects (e.g., walls in a building), then the optimal validity periods for TBVs can be calculated exactly: Starting with an initial validity period of one frame, the period is repeatedly doubled until the bounding volume is no longer hidden by static objects, then binary search is used to find the exact moment at which the volume starts to become visible. If the viewpoint is not stationary, this will not necessarily find optimal expiration dates. Since it looks for TBVs which are "almost visible," i.e., will become visible in just one frame's time, even the slightest movement of the viewpoint might reveal a part of the volume to the viewer, requiring reference to the dynamic object. Therefore, if the viewpoint is movable, it would be better not to use such long validity periods. A better choice might be to use shorter periods (e.g., by half), to get smaller bounding volumes which will take longer to be revealed.

In general situations, a better choice is *adaptive* validity periods. If a TBV expires before it is revealed, then its validity period was too short, because it would have been possible to postpone the costly reference to the dynamic object by choosing a later expiration date. Therefore, a longer period is selected for the object's next TBV. In the opposite case—if the TBV is seen before its expiration—the period was too long, because a shorter period would have produced a smaller TBV that might have remained occluded for a longer time. Therefore, if the object itself is still hidden, a shorter validity period is chosen for its next TBV. This strategy makes validity periods adapt to their object's behavior and visibility status. Dynamic objects that are fast-moving or stay near visible regions of space have relatively short validity periods; objects in obscured regions have longer periods, and are sampled less frequently as time progresses.

3.5 Fuzzy TBVs

The knowledge of hard-and-fast motion constraints by which TBVs can be calculated may be an excessive requirement. Even if such contraints are known, they may be very loose, yielding relatively big TBVs that do not contribute much to output sensitivity. If this is the case, fuzzy TBVs can be used. A *fuzzy* TBV is a volume that is not necessarily guaranteed to contain its dynamic object throughout its validity period, but is only assumed to do so with some probability. For example, an object may move at a velocity of no more than v_{MAX} with probability 99 percent; this maximum velocity v_{MAX} can be used to calculate a TBV for the object. If the agent that controls the object (e.g., a simulator or another user's workstation) detects that a dynamic object violates an assumption that was used to construct a fuzzy TBV for the object, the agent reports this violation to the algorithm.[1] The algorithm then treats this TBV as an expired TBV: It calls del_TBV(ID) in step 1, where ID is the identifier of the corresponding dynamic object. Consequently the algorithm no longer ignores the object, but inserts it into the spatial data structure and tests it for visibility.

Alternatively, lower performance may be acceptable for objects that exceed their assumed constraints. For example, the normal mode of interaction in a virtual environment is smooth motion, simulating walking or flying through the environment. The speed limit for such motion is used to calculate TBVs. If the system allows users to "teleport" to a remote location, this limit is violated; however, in such a case, it is quite acceptable for the display to take longer to update than during ordinary, smooth motion. This is true both for the teleporting user's display and for the displays of other users into whose vicinity this user teleports.

3.6 Implicit Data Structure Update

Step 2 of the algorithm presented in Section 3.3 may, in fact, be omitted altogether. This might cause some visible dynamic objects to be associated with the wrong nodes of the spatial data structure **T**, but has few ill effects on the final rendered images. To see why this is the case, consider a dynamic object which the algorithm currently assumes to be visible. The object may or may not be, in fact, visible, but it is considered visible either because it was visible in the previous frame, or because its TBV has expired, or because the object violated a motion constraint which was used to construct a fuzzy TBV (see the previous section). Assume the object is associated with the wrong nodes of the data structure, i.e., it is not associated with some nodes it intersects at its current position, or it is associated with some nodes it does not intersect.

If the dynamic object is actually visible, and any of the data structure nodes it is associated with (either rightly or wrongly) is also visible, then this node will be encountered during the traversal in Step 3 of the algorithm, and the object will be properly displayed. However, if none of the nodes that the object is associated with are encountered during the traversal, then Step 4 creates a TBV for the object and inserts it into the data structure **T**. This TBV contains the object's current updated position, as well as some predicted future positions. In the next frame, if the object is still visible, its TBV will be visible, too, and will be associated with the right nodes, since it has just been created and inserted into **T**. Therefore, the TBV will be encountered during the Step 3 traversal. The algorithm will then proceed to obtain the object's current position, insert it into the data structure, and display it. That is, a visible dynamic object can be wrongly omitted from the display for at most one frame, not for extended periods of time.

Next, assume that the object is not actually visible, but the algorithm initially assumes, wrongly, that it is. If all the nodes of the data structure that the object is associated with are hidden, too, then the object will be handled properly. If it is associated with a visible node, then it will be needlessly rendered during Step 3. However, since visibility calculation

1. This bears some similarity to dead reckoning (see Section 2.3), where the agent that controls a dynamic object reports to other agents when the object deviates from its extrapolated behavior by more than some tolerance.

is performed, this unnecessary display does not affect the final image, because the object will be overdrawn by nearer objects; the only drawback is the time this unrequired rendering takes. The unnecessary rendering will not be repeated in the following frames, because Step 4 of the current frame detects that the object is invisible and creates a TBV for it.

This method avoids the explicit update of the spatial data structure for the movements of the visible dynamic objects, yet implicitly updates the structure through the use of TBVs. It may be simpler to implement than the complete algorithm. The biggest disadvantage of this method is that the time it takes to generate TBVs of visible dynamic objects, insert them into the octree, and immediately delete them again in the next frame may be prohibitive. Whether this is indeed the case, or whether this cost is offset by the time saved by not updating the data structure for every single motion of a dynamic object, is application-dependent.

3.7 Reduction of Communication Requirements

The dynamic scene occlusion culling algorithm eliminates the rendering of occluded objects, and the update of the underlying spatial data structure for most dynamic objects (namely those which are invisible and whose TBV is valid and hidden). However, the algorithm also has another, very important advantage: In addition to the auxiliary data structure, the scene model itself is not updated for these hidden objects, but is left with their old configurations and properties. In the model, the data about these objects is outdated, but this does not matter since they are invisible anyway. This saves a significant amount of calculation if these objects exhibit complex behaviors, deformations etc., involving intensive computation; or if their up-to-date configurations would have been obtained through a relatively slow communication network, such as the Internet.

This is particularly beneficial in applications that would otherwise have high communication requirements, such as multiuser virtual environments. In such systems, many users roam simultaneously through a shared 3D scene, seeing each other as avatars at the appropriate places in the virtual world. Consider a distributed system with no central server, for example, the VRMUD environment described by Earnshaw et al. [42]. Rather than having every workstation broadcast its user's current configuration, each station can execute our algorithm to specifically keep track of just those other users that it may observe. For all the other users it keeps TBVs, and requests a geometry update only when a TBV expires or becomes visible.

In a client-server configuration, e.g., Funkhouser's RING system [4], each user's station acts as a client; a central server (or a network of servers) maintains the model, and updates each client as to the geometry it might see. Some of this geometry may depend on other clients, for example, other users' avatars. Instead of the server constantly checking which clients need to be notified of other clients' movements and deformations, it can have each pair of clients establish a peer-to-peer communications channel between them when they first see each other. Every client runs the dynamic scene occlusion culling algorithm; it requests updates from other clients, over the peer-to-peer links, only when the algorithm requires it. This approach is more scalable, i.e., it allows more clients to connect to the same server, because it utilizes the computational resources of the clients to reduce the server's load.

Compared to the message reduction schemes mentioned in Section 2.3, this method is more accurate than decomposition of the environment into disconnected cells, because it does not omit messages between nonneighboring, but intervisible, cells. It is more general than cell intervisibility precalculation, which is primarily applicable to indoor environments; and it can be more efficient than multicasting, which only reduces the overhead involved in sending messages, not in receiving them. It can be combined with dead reckoning to eliminate still more messages: Update messages are not sent for dynamic objects which are either hidden or whose extrapolated position is near enough to the actual one.

4 OCTREE-BASED DYNAMIC SCENE OCCLUSION CULLING[2]

The octree-based version of the dynamic scene occlusion culling algorithm is founded on hierarchical Z-buffering. This technique, described in Section 2.1, was originally developed for static scenes: Its preprocessing stage builds the octree once, and its runtime stage then repeatedly traverses this tree without changing it. In the octree, each node is split into eight octants if it intersects too many graphic primitives and if it is not too deep in the tree. Every primitive is associated with some nodes; if it intersects more than one, it is associated with multiple nodes, rather than being split between them.

For dynamic scenes, the octree needs to be modified at runtime. To insert an object (or a TBV) into the tree, the object's primitives are associated with the nodes they intersect, and the octree nodes which are associated with too many primitives are then split. To delete an object or a TBV, its primitives are disassociated from the nodes they were associated with; then, sibling nodes associated with the same primitives, or with few enough primitives, are merged.

To update the octree for a dynamic object's motion, the object can be deleted from the tree, and, then, inserted at its new position (see Fig. 5). The resulting octree is equivalent to the tree that would have been constructed with the object at its postmovement position. If the update is performed for relatively few dynamic objects, then it is much more efficient than rebuilding the entire tree from scratch. However, it is not optimal: It needlessly wastes time by merging nodes which are immediately split again. This is illustrated in Fig. 6. (If the update is performed by inserting the moving primitives at the new positions before deleting them from their old positions, then it wastes time for the opposite reason: the unnecessary creation of nodes which are immediately deleted again.)

An improvement is to not update the entire octree for a dynamic object's motion, but only the subtree whose root is the least common ancestor (LCA) of the object's old and new positions. See the pseudocode in Fig. 7 and the illustration in Fig. 8. This method can be shown to yield the same results as updating the entire octree, and to eliminate all unnecessary deletions and recreations of octree nodes [43].

2. The work described in this section was presented at the Eurographics '96 conference [43].

```
procedure update(octree, primitive, new_config);
begin
    delete(octree, primitive);
    primitive.config ← new_config;
    insert(octreee, primitive)
end
```

Fig. 5. Octree update by primitive deletion and insertion.

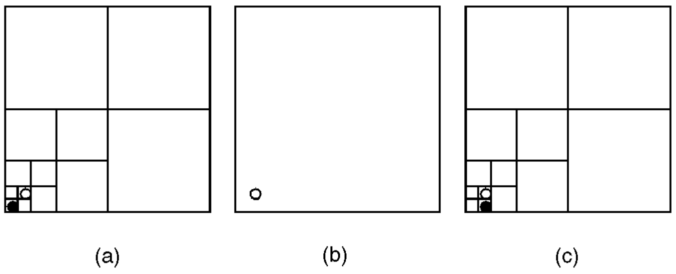


Fig. 6. Octree nodes might be needlessly deleted and recreated during update. (a) A vertical view of an octree with an initial model (two primitives); (b) after deletion of dynamic primitive; (c) after insertion of dynamic primitive at new configuration.

```
procedure update(octree, primitive, new_config);
begin
    v ← LCA(primitive, new_config);
    delete (v, primitive);
    primitive.config ← new_config;
    insert(v, primitive)
end
```

Fig. 7. Octree update under the LCA.

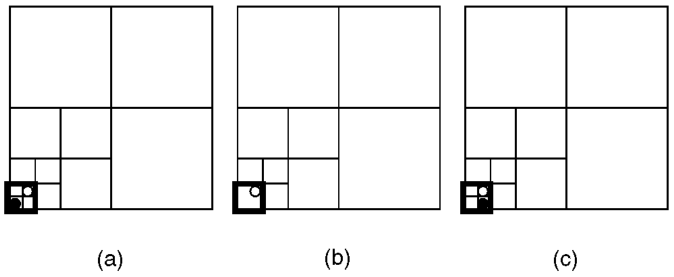


Fig. 8. Updating under the least common ancestor (LCA) of a dynamic primitive's old and new configurations eliminates unnecessary node deletions and creations. (a) The bold square is the LCA; (b) after deletion of primitive; (c) after insertion of primitive at new configuration.

The search for the LCA can be combined with the node merge, as both are performed during a bottom-up traversal of the octree. Fig. 9 shows this combined search-and-deletion for primitives associated with a single octree node; Fig. 10 is the pseudocode for primitives associated with multiple nodes, as in the hierarchical Z-buffer algorithm.

This method is an improvement over updating the entire octree because, for relatively deep trees representing big, complex models, the LCA is expected to be closer to the leaves than to the root. This is due to temporal coherence [44]—each dynamic object's location is expected to be close to its place at the previous frame; and because the planes separating high (large) octree nodes are few and far between compared to the planes separating smaller nodes.

```
procedure update(octree, primitive, new_config);
begin
    v ← primitive.node;
    dissociate(v, primitive);
    primitive.config ← new_config;
    while v.parent can be collapsed and not
      (v contains primitive ) do begin
        v ← v.parent;
        collapse(v)
    end;
    while not (v contains primitive) do
        v ← v.parent;
    insert(v, primitive)
end
```

Fig. 9. The bottom-up octree update algorithm for single primitive references.

```
procedure update(octree, primitive, new_config);
begin
    σ ← primitive.nodes;
    primitive.config ← new_config;
    ℓ ← lowest level in σ;
    s ← octree nodes at level ℓ in σ;
    while |s| > 1 or ℓ < highest level in σ do begin
        t ← ∅;
        for each node v ∈ s do begin
            t ← t ∪ {v.parent};
            if (v ∈ σ) and not
                    ((v intersects primitive)
                    and (primitive is small compared to
                      v.parent)
                    and not (primitive is small compared
                      to v)) then
                dissociate(v, primitive);
            if v.parent can be collapsed then
                collapse(v.parent)
        end;
        ℓ ← ℓ – 1;
        s ← t ∪ nodes at level ℓ in σ
    end;
    v ← single octree node in s;
    if v ∈ σ then
        dissociate(v, primitive);
    while v.parent can be collapsed and not
      (v contains primitive) do begin
        v ← v.parent;
        collapse(v)
    end;
    while not (v contains primitive) do
        v ← v.parent;
    insert_multiple(v, primitive)
end
```

Fig. 10. The octree update algorithm for multiple primitive references.

Therefore, the probability of an object crossing a dividing plane decreases exponentially with the height of the nodes separated by that plane. This stands in marked contrast to the case where there is no temporal coherence, and dynamic objects jump randomly between frames: in that case, the probability of an object crossing a separating plane *increases* exponentially with the height of the nodes.

This method optimizes the octree update for individual dynamic objects in the upd_vis subroutine. The dynamic scene occlusion culling algorithm reduces the number of objects for which the octree is updated. In Step 3 of the algorithm, a conservative estimate of the visible objects and TBVs is those associated with octree nodes traversed by the hierarchical Z-buffer algorithm.

Implementation and results of octree updates and of octree-based dynamic scene occlusion culling are presented in Section 6.1.

5 BSP-Tree-Based Dynamic Scene Occlusion Culling

To demonstrate that dynamic scene occlusion culling can be based on various occlusion culling algorithms, we show how it can be founded on a different algorithm: Naylor's BSP tree projection technique. This particular technique was chosen because it has several advantages over hierarchical Z-buffering, namely independence of graphics hardware and image size. Furthermore, it is sufficiently different from hierarchical Z-buffering to highlight the essence of our approach.

BSP trees with leaf "in/out" attributes, as used by the projection algorithm, are inherently different from octrees and *k*-D trees: They represent the objects themselves, whereas octrees and *k*-D trees are merely auxiliary data structures, supplementing a B-rep of the objects. Therefore, the BSP tree projection algorithm is generalized to dynamic scenes in a somewhat different way from the octree-based hierarchical Z-buffer algorithm.

The specifics of the BSP-tree-based version of the dynamic scene occlusion culling algorithm were given elsewhere [45]. Implementation and results are presented in Section 6.2.

6 Implementation and Results

6.1 Octree-Based Dynamic Scene Occlusion Culling

6.1.1 Octree Updates

Octree update under the least common ancestor (LCA) of a dynamic object's old and new positions, as described in Section 4, was tested on two different scenes. Test scene 1 is very simple, consisting of a small cube moving in a larger one. The small cube was repeatedly moved in steps equal to half its side length, such that its distance from the large cube's edge remained equal to the small cube's side length. The total octree update time was measured at all steps. Octree update under the LCA was consistently more efficient than the update of the entire octree. As expected, the exact speedup achieved depended on the ratio r of cube sizes; generally, it was proportional to log r (specifically, it approached $0.145 + 0.385 \log_{10} r$). This is because the depth of the octree depends on the proximity between objects. As the cubes become closer, the octree must become deeper to separate them. This directly affects the update time of the entire octree, but has less effect on the bottom-up update under the LCA, where most update operations are not dependent on the octree's height.

Test scene 2 is more complicated, and more realistic, than test scene 1. It models a table for four, complete with chairs, plates, glasses, and candles, as shown in Fig. 11. The model contains 5,745 polygons, resulting in an octree of depth 11. The dynamic object for test scene 2 is again a small cube, this time circling around one of the candlesticks in 1^o increments. Our octree update algorithm achieved a speedup of 2.7 for this test scene. In contrast to test scene 1, where the entire octree was built just for the dynamic object, in test scene 2, almost all of the octree is constructed for the static parts of the model. Most of the speedup in test scene 2 is not the result of eliminated node deletions, but of a shortened search down the octree each time the dynamic object is inserted into the octree at a new position.

In these particular test scenes, it might be quicker to construct an octree only for the static objects, and to render all the dynamic objects at each frame. However, in scenes containing many moving objects, this alternative is slower than updating the octree and using it in an occlusion culling algorithm (as the next section illustrates). The results shown above demonstrate the advantage of performing this update under the LCA, rather than under the root.

6.1.2 Temporal Bounding Volumes

The performance of the octree-based dynamic scene occlusion culling technique was compared to that of the hierarchical Z-buffer algorithm, updating the octree for each dynamic object at every frame (HZB), and to simply rendering all the objects at every frame by an ordinary hardware Z-buffer (ZB). Two variants of dynamic scene occlusion culling were tested: the full version of the algorithm, as presented in Section 3.3, with the octree updates in Step 2 performed under the LCA of each

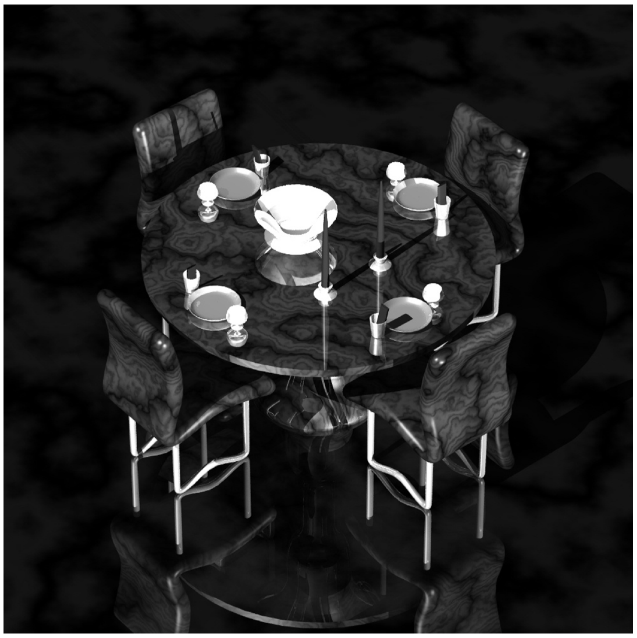


Fig. 11. Test scene 2 (image courtesy of Gershon Elber).

visible dynamic object's old and new positions as in Section 4 (TBVe); and the version with implicit octree updates, as discussed in Section 3.6 (TBVi). Performance was tested on models of the IRIT solid modeler [46]. The tests were carried out on an SGI O2 workstation with a hardware Z-buffer, using the GL library for display. Display lists were employed to reduce rendering time.

The scene models used in the tests were buildings consisting of interconnected rooms, each furnished with a table; the dynamic objects were chairs which followed trajectories approaching the tables. See Fig. 4a. The TBVs were spheres, calculated based on the chairs' maximum velocity and radius. (Sweep-surface TBVs were also tried, but were found to give lower performance because their more complex geometry yields longer octree update times.)

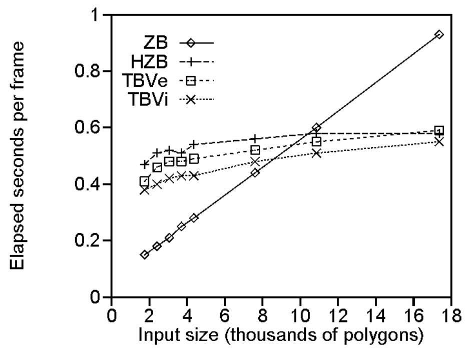
Fig. 12a shows the performance of the various techniques for models of increasing size, where the number of dynamic objects was kept constant. As can be expected, ZB's runtime is linear in the size of the model, while those of HZB, TBVe, and TBVi are nearly constant since the same number of objects are visible regardless of the model's size. (TBVe and TBVi do slightly better because only half of the dynamic objects are actually visible.)

In Fig. 12b, the performance of the four techniques is compared for models of increasing size, where the ratio of the number of dynamic objects to the total number of objects in the model is fixed, i.e., there are always four moving chairs in each room. In this case, both ZB and HZB have a linear runtime with respect to the size of the model; HZB does much worse than ZB because of the need to keep the octree up-to-date, incurring considerable overhead. TBVe and TBVi eliminate many of these octree updates, thus improving performance considerably.

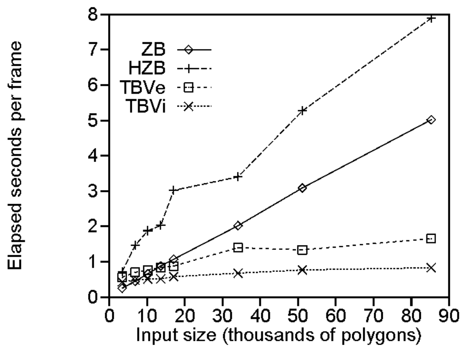
Fig. 12c presents the results obtained by the four techniques for increasing numbers of dynamic objects within a static model of fixed size (125 rooms and tables). This demonstrates that the performance gains achieved by TBVe and TBVi increase with the proportion of dynamic objects in the model.

These experiments show that the runtime of TBV is indeed dependent mostly on the number of visible objects and not on the number of hidden objects, whether static or dynamic. In contrast, ZB depends linearly on the total number of objects. HZB depends heavily on the number of dynamic objects, and it also depends on the number of visible objects. TBVi does slightly better than TBVe because it saves some octree updates, at the cost of possibly showing some newly-visible dynamic objects one frame too late. This effect was not visually noticeable in the conducted experiments.

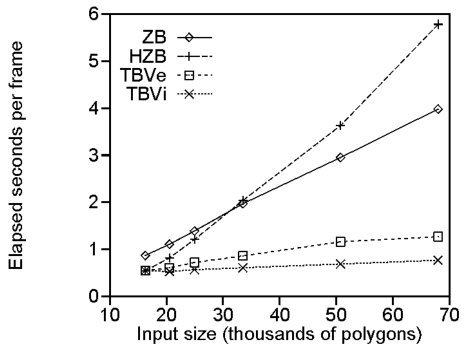
To demonstrate the performance of TBV on more realistic models, a second set of experiments was conducted on objects—a building and some moving men—found as VRML models on the WWW (see test scene in Fig. 13a). The



(a)

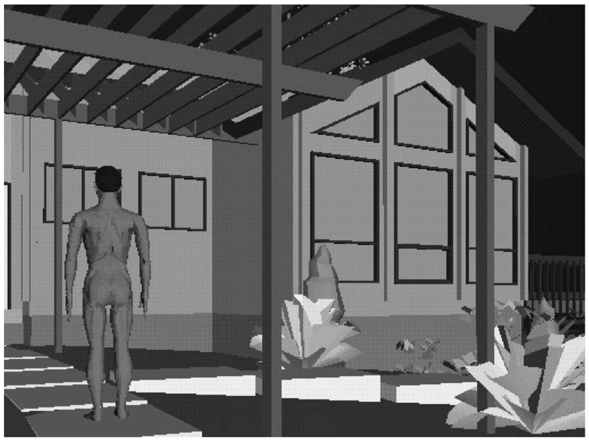


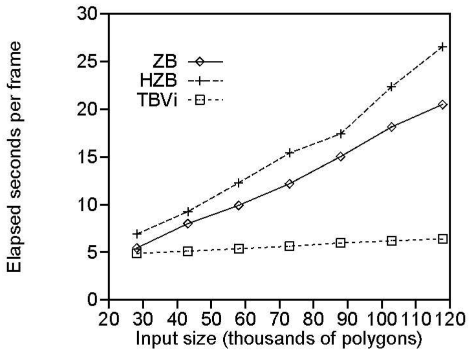
(b)



(c)

Fig. 12. Performance of algorithms on a test scene. (a) With a varying number of static polygons, and the number of dynamic polygons fixed at 1,100; (b) with the number of dynamic polygons approximately 80 percent of the entire scene; (c) with 16,250 static polygons and a varying number of dynamic polygons.





(a) (b)

Fig. 13. (a) A test scene used in our experiments; (b) performance of algorithms on the test scene with a fixed number of static and visible dynamic objects and a varying number of hidden dynamic objects.

experiments were performed on an SGI Indy R5000 computer. The number of static objects and the number of visible dynamic objects were kept constant at 13,220 and 14,946 polygons, respectively; the number of hidden dynamic objects was varied by adding men inside the building. The results, shown in Fig. 13b, demonstrate again that the runtime of ZB is linearly proportional to the total number of objects in the scene; HZB does even worse than ZB, because it requires that the octree be updated for every dynamic object; and the runtime of TBVi is almost constant in comparison to ZB and HZB, because it updates the octree only for the visible dynamic objects. Note that none of the techniques achieves real-time speed. As discussed in Section 2.1, given current hardware performance, such speed cannot be achieved using the hierarchical Z-buffer algorithm.

The experiments show that the dynamic scene occlusion culling algorithm is superior to existing visibility algorithms, most notably when a large fraction of the scene polygons are dynamic.

Fuzzy TBVs. To test the effect of fuzzy TBVs, discussed in Section 3.5, an experiment was conducted using a model consisting of 125 rooms, 125 tables, and 500 moving chairs (85,250 polygons altogether). Fuzzy TBVs were simulated by randomly selecting, at each frame, a fraction of the valid TBVs and invalidating them, i.e., treating them as if they have expired. This is equivalent to a corresponding fraction of fuzzy TBVs violating an assumed motion constraint by which they were constructed. Fig. 14 shows the runtime performance of TBVe for different values of the fraction. Notice that the performace penalty is not too great even if a significant proportion of the fuzzy TBVs (e.g., 20 percent) violate their assumptions every frame. As the fraction approaches 1 (that is, when the given motion constraints are useless), the performance is only slightly worse than HZB.

6.2 BSP-Tree-Based Dynamic Scene Occlusion Culling

The BSP tree projection algorithm has a high overhead relative to other visibility algorithms, due to the number of numerical calculations it involves. While it is output-sensitive, its break-even point, compared, e.g., to hardware Z-buffering, is beyond real-time refresh rates. It is also slower than hierarchical Z-buffering for models of comparable size, and harder to implement. Nevertheless, it is of interest to see how this algorithm, too, can be improved by our dynamic scene occlusion culling technique.

The BSP-tree-based version of the dynamic scene occlusion culling algorithm, described in Section 5, was implemented and tested on an SGI O2 computer. Again, the static scenery consisted of interconnected rooms furnished with tables, and the dynamic objects were chairs. The models were converted to BSP trees using Thibault and Naylor's BSP tree from B-rep construction algorithm [47], employing Naylor's minimum-expected-cost heuristics [31]. As for Fig. 12c and Fig. 13b, the number of static objects and the number of visible dynamic objects were kept constant, and the number of hidden dynamic objects was varied by adding moving chairs in unseen parts of the world. Adaptive expiration was used for the validity periods of TBVs.

Fig. 15a shows the elapsed time per frame in our dynamic scene occlusion culling algorithm (DSVC) vs. naïvely updating the scene for every dynamic object at each frame and displaying it using Naylor's BSP tree projection algorithm [9] (BSP Proj). Both methods have approximately the same overhead. However, DSVC's performance is almost

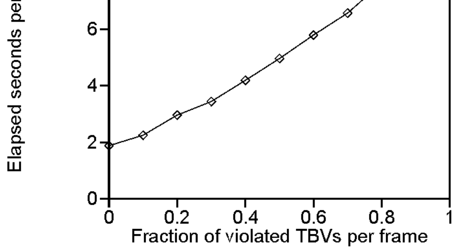


Fig. 14. Dynamic scene occlusion culling performance with fuzzy TBVs.

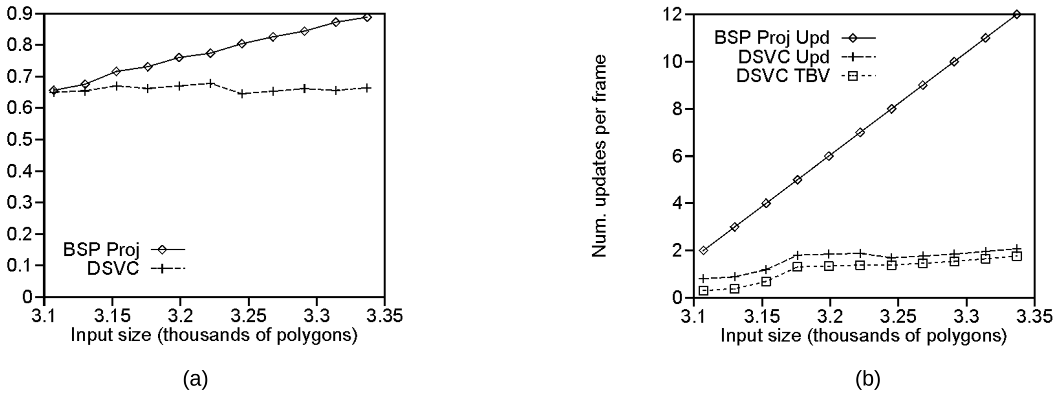


Fig. 15. Performance of BSP-tree-based visibility algorithms with two visible dynamic objects and a varying number of hidden dynamic objects. (a) Rendering; (b) communications.

constant compared to BSP Proj, whose runtime continues to increase linearly in addition to the overhead. The difference between the two methods is expected to increase with the number of dynamic objects.

In Fig. 15b, communications time over a 26-frame sequence was simulated by counting the number of times a dynamic object is updated (which would require a message to be transmitted in a distributed environment). Unsurprisingly, the number of update messages when every object is updated each frame (BSP Proj Upd) is exactly linear in the number of dynamic objects. Using our algorithm, the number of object messages (DSVC Upd), even when combined with the number of messages specifying TBVs of occluded dynamic objects (DSVC TBV), is again almost constant. This demonstrates the potentially huge benefit obtainable by our method for reducing network traffic.

7 Conclusions

We have presented a scheme for occlusion culling in dynamic scenes. The scheme modifies existing static-scene occlusion culling techniques by updating their underlying spatial data structure for the movements of dynamic objects. The structure is not updated for every object motion, but only for the visible objects and for a minotority of the hidden ones. This is accomplished by inserting temporal bounding volumes (TBVs) into the data structure in lieu of occluded dynamic objects.

Our dynamic scene occlusion culling technique is quite suitable to most real-world applications, because the assumptions on which it is based are not too demanding. The sole requirement is that some constraint be known on the dynamic objects' motions; full a priori knowledge of their behavior is not necessary. It is rare for the movements of the dynamic objects to be so random and haphazard that there is absolutely no knowledge about them.

Dynamic scene occlusion culling is most beneficial when most of the scene is occluded. This is usually the case in a number of applications, the most notable of which is architectural scenes. In other application areas, where there are no significant occlusions, the benefits of occlusion culling might be marginal. A combination of occlusion culling with complementary optimization techniques, such as level-of-detail control, is expected to yield a high performance in most circumstances.

The general algorithm uses some underlying occlusion culling technique. Implementations were presented for two distinct techniques: hierarchical Z-buffering and BSP tree projection. Other culling algorithms can be used in a similar way. Minor adjustments may be required for some algorithms. For example, if Teller et al.'s cell intervisibility precalculation is used [8], [22], then occlusions of dynamic objects by static scenery can easily be utilized, but occlusions by dynamic objects are harder to detect.

The dynamic scene occlusion culling algorithm ignores most unseen dynamic objects most of the time. It thus harnesses the power of occlusion culling to reduce potentially time-consuming object position updates, e.g., avatar movements transmitted over a slow communication link in a multiuser environment. The same general idea may be applied to level-of-detail control: Update messages for distant objects with small screen projections can be sent less frequently than for apparently large objects.

References

[1] R. Carey and G. Bell, *The Annotated VRML 2.0 Reference Manual*. Addison-Wesley Longman, 1997.

[2] Tecnomatix Technologies Ltd., "ROBCAD: Manufacturing Process Design Tools," http: //www.tecnomatix.com/robc.htm.

[3] C. Carlsson and O. Hagsand, "DIVE—A Platform for Multi-user Virtual Environments," *Computers & Graphics*, vol. 17, no. 6, pp. 663–669, 1993.

[4] T.A. Funkhouser, "RING: A Client-Server System for Multi-user Virtual Environments," *Proc. 1995 Symp. Interactive 3D Graphics*, pp. 85–92, Monterey, Calif., ACM SIGGRAPH, Apr. 1995, http://www.bell-labs.com/user/funk/symp95.ps.gz.

[5] M.R. Macedonia, D.P. Brutzman, M.J. Zyda, D.R. Pratt, P.T. Barham, J. Falby, and J. Locke, "NPSNET: A Multi-player 3D Virtual Environment Over the Internet," *Proc. 1995 Symp. Interactive 3D Graphics*, pp. 93–94, Monterey, Calif., ACM SIGGRAPH, Apr. 1995, ftp://taurus.cs.nps.navy.mil/pub/NPSNET_MOSAIC/macedonia.exploit.ps.Z.

[6] J.W. Barrus, R.C. Waters, and D.B. Anderson, "Locales: Supporting Large Multiuser Virtual Environments," *IEEE Computer Graphics and Applications*, vol. 16, pp. 50–57, Nov. 1996. Extracted from MERL TR95-16a [48].

[7] O. Hagsand, "Interactive Multiuser VEs in the DIVE System," *IEEE MultiMedia*, vol. 3, pp. 30–39, Spring 1996, http:// www.computer.org/multimedia/mu1996/u1030abs.htm.

[8] S.J. Teller and C.H. Séquin, "Visibility Preprocessing for Interactive Walkthroughs," *Proc. Conf. SIGGRAPH '91*, pp. 61–69, Las

Vegas, July 1991. *ACM Computer Graphics*, vol. 25, no. 4, http://graphics.lcs.mit.edu/~seth/pubs/siggraph91.ps.Z.

[9] B.F. Naylor, “Partitioning Tree Image Representation and Generation from 3D Geometric Models,” *Proc. Graphics Interface ‘92*, pp. 201–212, Vancouver, May 1992.

[10] N. Greene, M. Kass, and G. Miller, “Hierarchical Z-Buffer Visibility,” *Proc. Conf. SIGGRAPH ’93*, pp. 231–238, Anaheim, Calif., *ACM Computer Graphics Ann. Conf. Series*, Aug. 1993.

[11] S. Coorg and S. Teller, “Real-Time Occlusion Culling for Models with Large Occluders,” *Proc. 1997 Symp. Interactive 3D Graphics*, pp. 83–90, Providence, R.I., *ACM SIGGRAPH*, Apr. 1997, http://graphics.lcs.mit.edu/~seth/pubs/realtime-i3d97.ps.Z.

[12] T. Hudson, D. Manocha, J. Cohen, M.C. Lin, K.E. Hoff III, and H. Zhang, “Accelerated Occlusion Culling Using Shadow Frusta,” Proc. *13th Ann. Symp. Computational Geometry*, pp. 1–10, Nice, France, *ACM SIGGRAPH* and *SIGACT*, June 1997, ftp://ftp.cs.unc.edu/pub/users/manocha/PAPERS/VISIBILITY/shadow.{ps.gz, pdf}.

[13] H. Zhang, D. Manocha, T. Hudson, and K.E. Hoff III, “Visibility Culling Using Hierarchical Occlusion Maps,” *Proc. Conf. SIGGRAPH ’97*, pp. 77–88, Los Angeles, *ACM Computer Graphics Ann. Conf. Series*, Aug. 1997, ftp://ftp.cs.unc.edu/pub/users/manocha/PAPERS/VISIBILITY/hom.{ps.gz, pdf}.

[14] W.J. Schroeder, J.A. Zarge, and W.E. Lorenson, “Decimation of Triangle Meshes,” *Proc. Conf. SIGGRAPH ’92*, pp. 65–70, Chicago, *ACM Computer Graphics*, vol. 26, no. 2, Aug. 1992.

[15] D.P. Luebke and C. Erikson, “View-Dependent Simplification of Arbitrary Polygonal Environments,” *Proc. Conf. SIGGRAPH ’97*, pp. 199–208, Los Angeles, *ACM Computer Graphics Ann. Conf. Series*, Aug. 1997, http://www.cs.virginia.edu/~luebke/publications/sig97.html.

[16] M. Garland and P.S. Heckbert, “Surface Simplification Using Quadric Error Metrics,” *Proc. Conf. SIGGRAPH ’97*, pp. 209–216, Los Angeles, *ACM Computer Graphics Ann. Conf. Series*, Aug. 1997, http://www.cs.cmu.edu/~garland/quadrics/quadrics.html.

[17] D.P. Luebke, “A Survey of Polygonal Simplification Algorithms,” Technical Report 97-045, Dept. of Computer Science, Univ. of North Carolina at Chapel Hill, 1997, ftp://ftp.cs.unc.edu/pub/publications/techreports/97-045.ps.Z.

[18] P.S. Heckbert and M. Garland, “Survey of Polygonal Surface Simplification Algorithms,” *SIGGRAPH ‘97 Course Notes*, Los Angeles, Aug. 1997, ftp://ftp.cs.cmu.edu/afs/cs/project/anim/ph/paper/multi97/release/heckbert/simp.pdf.

[19] J.H. Clark, “Hierarchical Geometric Models for Visible Surface Algorithms,” *Comm. ACM*, vol. 19, pp. 547–554, Oct. 1976.

[20] D.J. Meagher, “Efficient Synthetic Image Generation of Arbitrary 3-D Objects,” *Proc. Conf. Pattern Recognition and Image Processing*, pp. 473–478, IEEE Computer Society, June 1982.

[21] M. de Berg and M. Overmars, “Hidden Surface Removal for *c*-Oriented Polyhedra,” *Computational Geometry Theory and Applications*, vol. 1, no. 5, pp. 247–268, 1992.

[22] T.A. Funkhouser, C.H. Séquin, and S.J. Teller, “Management of Large Amounts of Data in Interactive Building Walkthroughs,” *Proc. 1992 Symp. Interactive 3D Graphics*, pp. 11–20, Cambridge, Mass., *ACM SIGGRAPH*, Mar.–Apr. 1992, http://www.bell-labs.com/user funk/symp92.ps.gz, http://graphics.lcs.mit.edu/~seth/pubs/FunkST-i3d92.ps.Z.

[23] T.A. Funkhouser, “Database Management for Interactive Display of Large Architectural Models,” *Proc. Graphics Interface ‘96*, pp. 1–8, Toronto, Ontario, May 1996, http://www.bell-labs.com/user/funk/gi96.ps.gz.

[24] S.J. Teller and P. Hanrahan, “Global Visibility Algorithms for Illumination Computations,” *Proc. Conf. SIGGRAPH ’93*, pp. 239–246, Anaheim, Calif., *ACM Computer Graphics Ann. Conf. Series,* Aug. 1993, http://graphics.lcs.mit.edu/~seth/pubs/visglobillum.ps.Z.

[25] D.P. Luebke and C. Georges, “Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets,” *Proc. 1995 Symp. Interactive 3D* Graphics, pp. 105–106, Monterey, Calif., *ACM SIGGRAPH*, Apr. 1995, http:/www.cs.virginia.edu/~luebke/publications/portals.html.

[26] N. Greene and M. Kass, “Error-Bounded Antialiased Rendering of Complex Environments,” *Proc. Conf. SIGGRAPH ’94*, pp. 59–66, Orlando, Fla., *ACM Computer Graphics Ann. Conf. Series*, July 1994.

[27] N. Greene, “Hierarchical Polygon Tiling with Coverage Masks,” *Proc. Conf. SIGGRAPH ’96*, pp. 65–74, New Orleans, *ACM Computer Graphics Ann. Conf. Series*, Aug. 1996, http://wwwx.cs.unc.edu/~hoff/projects/comp390/papers/greene.pdf.

[28] N. Greene, “Hierarchical Rendering of Complex Environments,” PhD thesis, Univ. of California at Santa Cruz, June 1995, ftp://ftp.cse.ucsc.edu/pub/tr/ucsc-crl-95-27.ps.Z.

[29] H. Fuchs, Z.M. Kedem, and B.F. Naylor, “On Visible Surface Generation by a priori Tree Structures,” *Proc. Conf. SIGGRAPH ’80*, pp. 124–133, Seattle, *ACM Computer Graphics*, vol. 14, no. 3, July 1980.

[30] B.F. Naylor, J. Amanatides, and W.C. Thibault, “Merging BSP Trees Yields Polyhedral Set Operations,” *Proc. Conf. SIGGRAPH ’90,* pp. 115–124, Dallas, *ACM Computer Graphics*, vol. 24, no. 4, Aug. 1990.

[31] B.F. Naylor, “Constructing Good Partitioning Trees,” *Proc. Graphics Interface ‘93*, pp. 181–191, Toronto, May 1993.

[32] N. Ahuja and C. Nash, “Octree Representations of Moving Objects,” *Computer Vision, Graphics, and Image Processing*, vol. 26, pp. 207–216, May 1984.

[33] J. Weng and N. Ahuja, “Octrees of Objects in Arbitrary Motion: Representation and Efficiency,” *Computer Vision, Graphics, and Image Processing*, vol. 39, pp. 167–185, Aug. 1987.

[34] Y. Chrysanthou and M. Slater, “Computing Dynamic Changes to BSP Trees,” *Proc. Eurographics ‘92*, pp. 321–332, Cambridge, U.K., *Computer Graphics Forum*, vol. 11, no. 3, Sept. 1992.

[35] Y. Chrysanthou and M. Slater, “Shadow Volume BSP Trees for Computation of Shadows in Dynamic Scenes,” *Proc. 1995 Symp. Interactive 3D* Graphics, pp. 45–50, Monterey, Calif., *ACM SIGGRAPH,* Apr. 1995.

[36] P.K. Agarwal, J. Erickson, and L.J. Guibas, “Kinetic Binary Space Partitions for Intersecting Segments and Disjoint Triangles,” *Proc. Symp. Discrete Algorithms*, San Francisco, Calif., Jan. 1998, http://www.cs.duke.edu/~pankaj/papers/kinetic-3d.ps.gz.

[37] E. Torres, “Optimization of the Binary Space Partition Algorithm (BSP) for the Visualization of Dynamic Scenes,” *Proc. Eurographics ‘90*, pp. 507–518, Montreux, Switzerland, Sept. 1990.

[38] B.F. Naylor, “Interactive Solid Geometry via Partitioning Trees,” *Proc. Graphics Interface ‘92*, pp. 11–18, Vancouver, May 1992.

[39] Circle of Fire Studios Inc., “Active Worlds,” http:/www.activeworlds.com/.

[40] W. Pugh, “Skip Lists: A Probabilistic Alternative to Balanced Trees,” *Comm. ACM*, vol. 33, pp. 668–676, June 1990, ftp://ftp.cs.umd.edu/pub/skipLists/skiplists.ps.

[41] O. Sudarsky, “Dynamic Scene Occlusion Culling,” PhD thesis, Technion—Israel Inst. of Technology, Jan. 1998, ftp://ftp.cs.technion.ac.il/pub/thesis/Oded_Sudarsky.thesis.ps.gz.

[42] R.A. Earnshaw, N. Chilton, and I.J. Palmer, “Visualization and Virtual Reality on the Internet,” *Proc. Conf. Visualization,* Jerusalem, Israel, Nov. 1995.

[43] O. Sudarsky and C. Gotsman, “Output-Sensitive Visibility Algorithms for Dynamic Scenes with Applications to Virtual Reality,” *Proc. Eurographics ‘96*, Poitiers, France, *Computer Graphics Forum*, vol. 15, no. 3, Aug. 1996, http://www.cs.technion.ac.il/ ~sudar/eg96.ps.gz.

[44] O. Sudarsky, “Exploiting Temporal Coherence in Animation Rendering: A Survey,” Technical Report CIS 9326, Computer Science Dept., Technion—Israel Inst. of Technology, Nov. 1993, http://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-info.cgi?1993/CIS/CIS9326 .

[45] O. Sudarsky and C. Gotsman, “Output-Sensitive Rendering and Communication in Dynamic Virtual Environments,” *Proc. Symp. Virtual Reality Software and Technology*, pp. 217–223, Lausanne, Switzerland, Sept. 1997, http://www.cs.technion.ac.il/~sudar/vrst97.ps.gz.

[46] G. Elber, “The IRIT Modeling Environment,” http://www.cs.technion.ac.il/~gershon/irit/.

[47] W.C. Thibault and B.F. Naylor, “Set Operations on Polyhedra Using Binary Space Partitioning Trees,” *Proc. Conf. SIGGRAPH ‘87* pp. 153–162, *ACM Computer Graphics*, vol. 21, no. 4, July 1987.

[48] J.W. Barrus, R.C. Waters, and D.B. Anderson, “Locales and Beacons: Efficient and Precise Support for Large Multi-user Virtual Environments,” Technical Report TR-95-16a, MERL—A Mitsubishi Electric Research Lab., Cambridge, Mass., Aug. 1996, http://www.merl.com/reports/TR95-16a/.

Oded Sudarsky received a BA degree in 1985, an MSc degree in 1988, and a DSc degree in 1998, all in computer science from the Technion—Israel Institute of Technology. His research interests include 3D graphics optimization techniques and active vision.

Craig Gotsman received a BSc degree in mathematics, physics, and computer science in 1983, an MSc degree in computer science in 1985, and a PhD degree in computer science in 1991, all from the Hebrew University of Jerusalem. Dr. Gotsman is a senior lecturer with the Department of Computer Science at the Technion—Israel Institute of Technology, currently on sabbatical leave at Virtue Ltd. His research interests include computer graphics, image rendering, and geometric modeling. Dr. Gotsman is a member of the IEEE and the IEEE Computer Society.