References

1. P. S. Heckbert and M. Garland, "Multiresolution modeling for fast rendering", in *Proceedings of Graphics Interface '94*, (Banff, Alberta), (May 1994).
2. J. Rohlf and J. Helman, "IRIS Performer: A high performance multiprocessing toolkit for real-time 3D graphics", in *SIGGRAPH '94 Conference Proceedings*, (Orlando, Florida), pp. 381–394, (July 1994). *ACM Computer Graphics*, **28**(4).
3. B.-O. Schneider, P. Borrel, J. Menon, J. Mittleman, and J. Rossignac, "Brush as a walkthrough system for architectural models", in *Proceedings of the Fifth Eurographics Workshop on Rendering*, (Darmstadt, Germany), pp. 389–399, (June 1994).
4. T. A. Funkhouser, "RING: A client-server system for multi-user virtual environments", in *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, (Monterey, California), pp. 85–92, ACM SIGGRAPH, (Apr. 1995).
5. Worlds Inc., "AlphaWorld." ⟨`http://www.worlds.net/products/alphaworld/`⟩.
6. R. A. Earnshaw, N. Chilton, and I. J. Palmer, "Visualization and virtual reality on the Internet", in *Proceedings of the Visualization Conference*, (Jerusalem, Israel), (Nov. 1995).
7. M. de Berg and M. Overmars, "Hidden surface removal for *c*-oriented polyhedra", *Computational Geometry Theory and Applications*, **1**(5), pp. 247–268 (1992).
8. T. A. Funkhouser, C. H. Séquin, and S. J. Teller, "Management of large amounts of data in iteractive building walkthroughs", in *Proceedings of the 1992 Symposium on Interactive 3D Graphics*, (Cambridge, Massachusetts), pp. 11–20, ACM SIGGRAPH, (Mar.–Apr. 1992).
9. S. J. Teller and C. H. Séquin, "Visibility preprocessing for interactive walkthroughs", in *SIGGRAPH '91 Conference Proceedings*, (Las Vegas), pp. 61–69, (July 1991). *ACM Computer Graphics*, **25**(4).
10. S. J. Teller and P. Hanrahan, "Global visibility algorithms for illumination computations", in *SIGGRAPH '93 Conference Proceedings*, (Anaheim, California), pp. 239–246, (Aug. 1993). *ACM Computer Graphics*, **27**(4).
11. N. Greene, M. Kass, and G. Miller, "Hierarchical Z-buffer visibility", in *SIGGRAPH '93 Conference Proceedings*, (Anaheim, California), pp. 231–238, (Aug. 1993). *ACM Computer Graphics*, **27**(4).
12. B. F. Naylor, "Partitioning tree image representation and generation from 3D geometric models", in *Proceedings of Graphics Interface '92*, (Vancouver), pp. 201–212, (May 1992).
13. N. Greene and M. Kass, "Error-bounded antialiased rendering of complex environments", in *SIGGRAPH '94 Conference Proceedings*, (Orlando, Florida), pp. 59–66, (July 1994). *ACM Computer Graphics*, **28**(4).
14. H. Fuchs, Z. M. Kedem, and B. F. Naylor, "On visible surface generation by *a priori* tree structures", in *SIGGRAPH '80 Conference Proceedings*, (Seattle), pp. 124–133, (July 1980). *ACM Computer Graphics*, **14**(3).
15. B. F. Naylor, J. Amanatides, and W. C. Thibault, "Merging BSP trees yields polyhedral set operations", in *SIGGRAPH '90 Conference Proceedings*, (Dallas), pp. 115–124, (Aug. 1990). *ACM Computer Graphics*, **24**(4).
16. Y. Chrysanthou and M. Slater, "Computing dynamic changes to BSP trees", in *Proceedings of Eurographics '92*, (Cambridge, U.K.), pp. 321–332, Blackwell Publishers, (Sept. 1992). *Computer Graphics Forum*, **11**(3).
17. Y. Chrysanthou and M. Slater, "Shadow volume BSP trees for computation of shadows in dynamic scenes", in *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, (Monterey, California), pp. 45–50, ACM SIGGRAPH, (Apr. 1995).
18. B. F. Naylor, "Interactive solid geometry via partitioning trees", in *Proceedings of Graphics Interface '92*, (Vancouver), pp. 11–18, (May 1992).
19. E. Torres, "Optimization of the binary space partition algorithm (BSP) for the visualization of dynamic scenes", in *Proceedings of Eurographics '90*, (Montreux, Switzerland), pp. 507–518, Elsevier Science Publishers B.V. (North-Holland), (Sept. 1990).
20. F. W. Burton and M. M. Huntbach, "Lazy evaluation of geometric objects", *IEEE Computer Graphics & Applications*, **4**(1), pp. 28–33 (1984).

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. TBV eliminates many of these octree updates, thus improving performance considerably.

Figure 6(c) presents the results obtained by the three techniques for increasing numbers of dynamic objects within a static model of fixed size (125 rooms and tables). This demonstrates that the performance gain achieved by TBV increases with the proportion of dynamic objects in the model.

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

5. TBVs in Distributed Virtual Environments

The TBV technique eliminates the need to update the visibility algorithm's underlying data structure (octree or BSP tree) for most dynamic objects, namely those which are invisible and with a TBV that is still valid. However, the technique has another, very important advantage: in addition to the auxiliary data structure, the scene model itself does not have to be updated for these hidden objects, but can be left with their old configurations and properties. This may save a significant amount of calculation if these objects exhibit complex behaviours, deformations etc. In the model, the data about these objects will be incorrect, but this will not matter since they are invisible anyway.

Consider a multi-user virtual environment, e.g. Funkhouser's RING system[4]. Many users can roam simultaneously through a shared 3D virtual building, seeing each other as graphic representations in the appropriate places in the building. In a client-server configuration, each user's workstation acts as a client, and a central server (or a network of servers) updates each client as to the geometry it might see; some of this geometry may depend on other clients. The server itself is constantly updated of every user's geometry. Using the TBV technique, some communication can be saved: the server can keep TBVs for those users which are currently not seen by any other user; it will only request geometry updates from those clients that become potentially visible, or whose TBV expires.

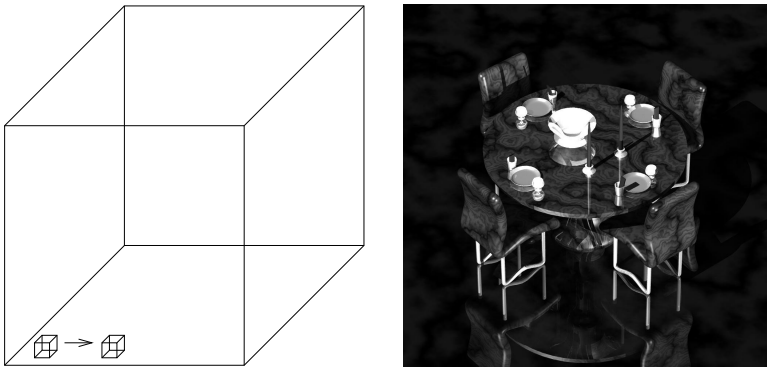
TBVs can eliminate still more communication overhead in a distributed VR system with no central server, such as the VRMUD environment described by Earnshaw et al.[6], or with a server that holds only the static parts of the model. Instead of having every station broadcast its user's movements and deformations, each station can specifically keep track of just those other users which may be visible to it. For all the other users it will keep TBVs, and request a geometry update once a TBV expires or becomes potentially visible.

6. Discussion and Future Work

The known visibility culling algorithms are unsuitable for dynamic scenes, because they rely on a heavy preprocessing stage, in which the objects' positions are used to build a hierarchical spatial data structure. The update technique we presented allows the underlying hierarchical structure to be updated efficiently by restricting the changes to a small part of the structure, the so-called *least common ancestor* of a dynamic object at its old and new configurations. *Temporal bounding volumes* (TBVs) were introduced to maintain output sensitivity in scenes with numerous dynamic objects: using these volumes, an occluded dynamic object is ignored most of the time, and need only be considered if its bounding volume is visible, or if the volume's expirations date has arrived. Since TBVs need not be known in advance, but are calculated on the fly during the rendering process, they are compatible with interactive applications such as virtual reality (VR). Moreover, TBVs allows the data of the hidden objects themselves to remain outdated, enabling significant savings if these data would take a long time to obtain, e.g. over a slow communications link in a distributed multi-user VR application.

As shown in Section 4, the speedup that the data structure update technique yields, relative to the naïve algorithm **N**, is proportional to the data structure's depth; in our test scenes, speedup of up to about $\times 3$ was achieved. The runtime of the TBV method is proportional to the number of visible objects (whether static or dynamic), whereas the hierarchical Z-buffer's runtime linearly depends on the number of object which are dynamic and/or visible, and the plain Z-buffer's runtime is proportional to the total number of objects—static and dynamic, hidden and visible.

We are currently modifying our implementation to use the more sophisticated BSP trees (rather than octrees). Future work includes experimentation with larger, more diverse models to get a better understanding of validity period selection, and experimentation with different rendering schemes. For example, in ray tracing, a dynamic object can be ignored unless its TBV is visible, but the volume may be visible either directly or indirectly, for example through reflection by another object.



(a) (b)

Figure 5: *(a) Test scene 1 (not shown to scale); (b) Test scene 2.*

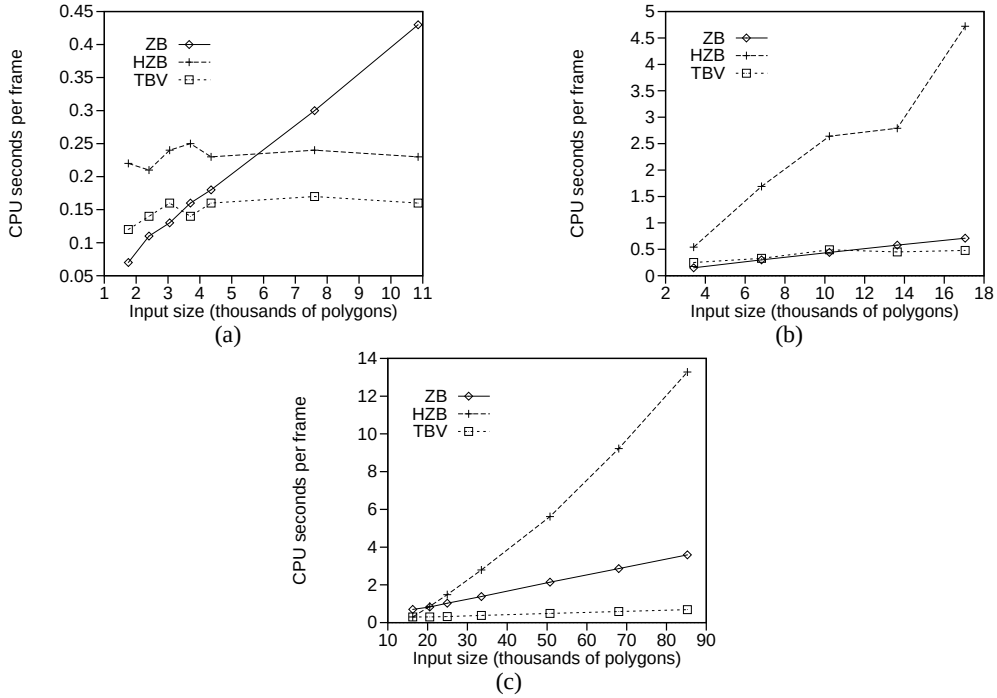
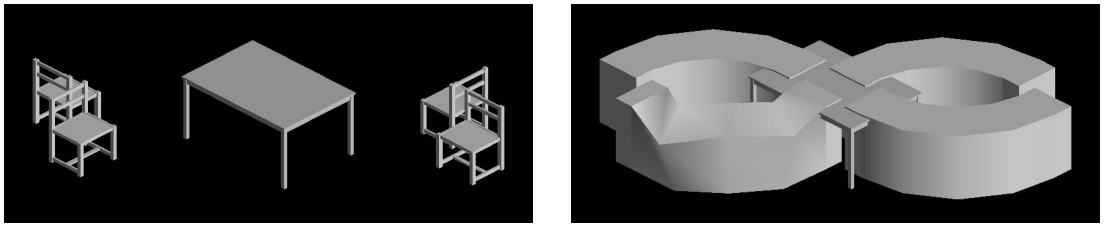


Figure 6: *Performance of algorithms on a test scene: (a) with a varying number of static polygons, and the number of dynamic polygons fixed at 1100; (b) with the number of dynamic polygons approximately 80% of the entire scene; (c) with 16,250 static polygons and a varying number of dynamic polygons.*

the tables. See Figure 4(a). To test the net improvement gained by the TBV technique, the octree was updated naïvely by both TBV and HZB. (For the models we used in these tests, 7 octree levels were sufficient; the total runtime saved by the octree update method was up to about 8%.)

Figure 6(a) shows the performance of the three 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 and TBV are nearly constant since the same number of objects are visible regardless of the model's size. (TBV does slightly better because only half of the dynamic objects are actually visible.)

In Figure 6(b), the performance of the three 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



(a) (b)

Figure 4: *(a) A typical room in the building model used in our experiments. (b) Temporal bounding volumes for four chairs moving towards the table. Note that the closer chair on the left wobbles during its movement.*

3.2.1. Implementation

Our current implementation of the TBV technique works in conjunction with the hierarchical Z-buffer visibility algorithm: TBVs optimize the updating of an octree; the visibility algorithm draws the scene using this octree, and detects which TBVs have become visible.

Dynamic objects should either have their trajectories available in advance as an animation script, or just have maximum known velocities. In the first case, bounding volumes are constructed as the sweep surfaces of the objects' bounding boxes along the trajectory curves. See Figure 4(b). In the second case, the bounding volume for a dynamic object is a sphere centered around the object's last known position; the sphere's radius is its validity period times the maximum velocity (plus the object's radius). Validity periods can be either constant, adaptive, or maximal (extending to the end of the animation sequence, if it is known).

4. Experimental Results

4.1. Data Structure Update for a Dynamic Object

The octree update algorithm was implemented for dynamic primitives associated with multiple octree nodes, without grouping. It was tested on two different scenes. Test scene 1 is very simple, consisting of a small cube moving inside a larger one (see Figure 5(a)). The small cube was repeatedly moved parallel to the x axis, in steps equal to half its side length, such that its distance from the nearest edge of the large cube remained equal to the small cube's side length. The total octree update time was measured at all steps. The octree updating algorithm consistently outperformed the naïve algorithm. The exact speedup achieved depended on the ratio of cube sizes; generally, it was proportional to the logarithm of the ratio of cube sides. This is due to the fact that the depth of the octree depends on the proximity between objects in the model. As the small cube becomes smaller and closer to the large cube's edge, the octree must become deeper to separate the two objects; this directly affects the run time of the naïve algorithm, but has less effect on the bottom-up octree updating algorithm, in which 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 Figure 5(b). The model contains 5745 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° increments. Our octree update algorithm achieved a speedup of 2.7 for test scene 2.

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.

4.2. Temporal Bounding Volumes

The performance of the temporal bounding volume technique (TBV) 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 Z-buffer (ZB). All tests were carried out on an SGI IRIS Indigo XS24 4000 workstation with a hardware Z-buffer, using the GL library for display. 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

number of dynamic objects, even if most of these objects are invisible; output sensitivity has been lost, with respect to the number of dynamic objects.

Our ultimate goal is to efficiently (output-sensitively) calculate visibility in a scene with numerous dynamic objects, utilizing occlusions. This goal seems to be unachievable if no constraints are known on the possible behaviours of dynamic objects. Although the position of each object might be expected to be close to its position at the previous frame (due to temporal coherence), it is not guaranteed to be so; it could have moved a very long distance between frames. Thus, any dynamic object which was invisible at the previous frame might become partly visible at the current frame, forcing all dynamic objects to be examined—no output sensitivity.

Suppose we know in advance, for each dynamic object, a bounding volume: some region of space which completely contains the object for the entire duration of the animation sequence. Then these volumes can be inserted into the model's spatial data structure; a dynamic object may be ignored unless the visibility cullling algorithm finds its bounding volume to be visible. Such bounding volumes can be found, for instance, for doors revolving on hinges, railroad cars moving on tracks etc. Performance in such cases depends on the tightness of the bounding volumes. In one extreme case, nothing is known about the dynamic objects' possible locations, and the bounding volumes are equal to the entire model space; since these volumes are (almost) always visible, each dynamic object must be examined at every frame, so visibility calculation is not output-sensitive. In the other extreme case, the bounding volume is very tight and is identical to the object itself; in fact, the dynamic object becomes static, since it has nowhere to move, and visibility calculation becomes as output-sensitive as it is with a static model.

It is not always possible to find bounding volumes for the entire period of the animation sequence; it is particularly difficult in interactive applications such as virtual reality (VR), simulators and games, where the dynamic objects' courses may not be known in advance and the animation period might be unlimited. Even if bounding volumes can be found, they might be too big and loose, and not contribute much toward output sensitivity.

Another approach is to calculate *temporal bounding volumes* (TBVs) for shorter periods of time, rather than for the whole animation sequence. For example, if a maximum velocity is known for each dynamic object, then, given an object's position at a certain time, it is possible to compute a bounding sphere for its location at any future time. In more general cases, TBVs do not have to be spheres—for instance, a dynamic object may have a maximum velocity as well as preset tracks, or the maximum velocity or resistance to movement might be different in different directions. Generally, we assume there is some way to find a bounding volume for each dynamic object from the time of the current frame until any desired moment in the future. This future moment constitutes an "expiration date" for the TBV; the period of time until that date is the bounding volume's validity period. A dynamic object which has been hidden should only be considered if its bounding volume becomes visible, or if the volume's expiration date arrives.

The TBV technique is a variation of the *lazy evaluation* principle: don't compute anything until it's absolutely necessary (see, e.g., Burton and Huntbach's use of this principle for 2D geometric calculations[20]). Note that the volumes are calculated on the fly, and that no prior knowledge of the objects' trajectories is necessary. Thus we assure compatibility with interactive applications in which this information is not available in advance, such as simulations, games and VR.

Once a dynamic object has been checked and found not visible, there remains the problem of choosing the right expiration date for its TBV. If the chosen date is too soon, then 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, the bounding volume might be too loose, and become visible after a short time.

If we assume that the viewpoint is stationary, and that 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, repeatedly double the period until the bounding volume is no longer hidden by static objects, then use binary search to find the exact moment at which the volume starts to become visible. The rendering process must keep an event queue of expiration dates, similar to priority queues used in simulation. As long as the bounding volumes are tight enough, output sensitivity will be maintained, since most volumes will remain invisible throughout their validity period.

If the viewpoint is not stationary, the method described above will not necessarily find optimal expiration dates. Since it finds 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, thus necessitating 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. Alternatively, one may choose *adaptive* validity periods: if the last period for a dynamic object's TBV was too short (the TBV expired before it was revealed), then the next TBV for the same object will have a longer validity period; in the opposite case, it will have a shorter period.

Proof If a node *u* which is not an empty leaf is deleted and then created again, then *u* is either *v* itself or an ancestor of *v*. Since the entire deletion and insertion process takes place under *v*, and *v* itself is neither deleted nor created in the process, this constitutes a contradiction. □

If the octree is relatively deep, representing a big, complex model, then *v* is expected to be closer to the leaves than to the root. This is because of temporal coherence—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. 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.

Further improvement may be attained by combining the *LCA* finding stage with the deletion of the primitive from the octree, since both operations take place during a bottom-up search of the octree.

If every primitive may be associated with several octree nodes (as in the hierarchical Z-buffer algorithm[11]), rather than with a single node, then this search up the octree begins at the set of lowest-level nodes with which the object is associated. Each iteration in the search refers to a certain level in the tree; the set of relevant nodes in that level includes those with which the primitive is explicitly associated, as well as ancestors of lower-level associated nodes. The search continues until the set reduces to a single node, then continues as for the case of primitives associated with a single node.

If there is a dynamic object *o* composed of several primitives, they can all be grouped together for the purpose of finding the least node, *v = LCA(o, new_config)*, containing all the primitives in *o* at their old and new configurations; the primitives are deleted from the subtree under *v* and then inserted at the new locations. (Conversely, under the naïve algorithm **N**, it would be better to move primitives to their new positions one by one: the primitives which have yet to be moved can act as place holders, preventing the unnecessary deletion of octree nodes.)

If there are two dynamic objects, *a* and *b*, then it is worthwhile to group them if the nodes LCA_a and LCA_b coincide, or if one of them is an ancestor of the other and not too high above it. If there are more than two dynamic objects, they should be grouped so in every group the height difference between the *LCA*'s of the different objects is not too great, and one of these *LCA*'s is an ancestor of all the rest.

3.1.2. BSP Tree Update

Several algorithms have been proposed for maintaining dynamic BSP trees. Chrysanthou and Slater's algorithm[16] updates a BSP tree to generate shadow volumes[17]. However, their tree maintains the faces of the objects, rather than their interiors (leaf "in/out" attributes). It is therefore unsuitable for operations such as collision detection[18] and visibility culling[12].

A 3D BSP tree of the entire model, with leaf "in/out" attributes, may be obtained by calculating the union[15] of the trees of its constituent objects. At preprocessing, a single BSP tree (a "scenery") is constructed for all the static objects, and a separate tree is built for each dynamic object. At every animation frame, each dynamic object is transformed to its current configuration (e.g. by multiplying all its coordinates by a 4×4 matrix and all its plane equations by the inverse of that matrix), and then united into the scenery.

Torres[19] proposes to associate the BSP tree's higher levels with planes that separate between objects, without cutting through the objects themselves (if possible); the trees for the individual objects are maintained under these separating planes. This allows the tree to be updated more efficiently: the objects' BSP trees usually do not interfere with each other, because they are separated by the higher-level planes.

3.2. Temporal Bounding Volumes

Better performance can be achieved by considering the purpose for which the spatial data structure is being updated. Since our objective is visibility calculation, we can allow a structure which is incorrect and outdated, provided this error does not manifest itself in the end product—the rendered image. It is possible to associate each hidden object with the highest (largest) data structure node which contains it and is still hidden itself, rather than associating every object with the smallest containing node, as usual. This would eliminate several transfers of the dynamic object between nodes, and allow the object to be discarded at an earlier stage of the visibility culling.

The problem with the above method is that each dynamic object must be checked at every frame, to find whether it is still entirely contained in the node it is associated with. Hence, the runtime per frame is (at least) linearly dependent on the total

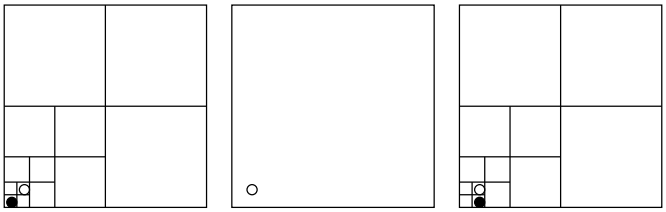


Figure 2: *A vertical view of an octree during the stages of the naïve update method* ***N****: (a) Initial model (two primitives); (b) After deletion of dynamic primitive; (c) After insertion of dynamic primitive at new configuration.*

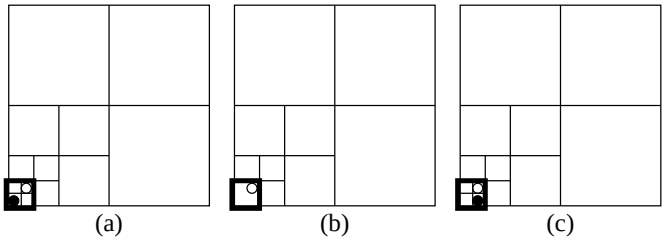


Figure 3: *A vertical view of an octree during the stages of the basic update method* ***B****: (a) The bold square is* v = LCA(primitive, new_config)*; (b) After deletion of dynamic primitive in* v*; (c) After insertion of dynamic primitive at new configuration in* v.

3.1. Data Structure Update for a Dynamic Object

3.1.1. Octree Update

The "naïve" way to update an octree for an object's motion is to delete the object from the octree, then insert it back at its new position. We denote this method as **N**. This technique is much better than rebuilding the entire octree, but it is still not optimal: octree nodes are needlessly deleted and created; furthermore, the search for the node in which to insert an object takes place over an unnecessarily long path. The update can be optimized by utilizing the *temporal coherence* inherent in animation sequences: consecutive images in the sequence are expected to be very similar to each other. An analogous correspondence exists between the model's states at consecutive frame times: dynamic objects usually do not "jump" randomly from place to place around the model; the position and shape of a dynamic object at a certain frame are expected to be close to its position and shape at the previous frame. Therefore, if a primitive is deleted and inserted as per the naïve method **N**, it is likely that octree nodes will be needlessly deleted, only to be immediately created again. See Figure 2.

It should be noted that the octree's depth is not necessarily dependent on the number of objects in it. By bringing objects closer together it is possible to obtain an arbitrarily deep octree.

The octree can be updated more efficiently by utilizing the fact that the change in the tree is restricted to the minimal subtree containing the dynamic primitive at both its old and new locations. We denote the node at the root of this subtree as *v = LCA(primitive, new_config),* as it is the Least Common Ancestor of all the nodes which contain the primitive's old and new configurations. The octree update method now becomes: find *v*, delete the dynamic primitive (still at its old position) from the sub-octree under *v*, and insert it at its new location into the same sub-octree. This basic update procedure is denoted **B**. See Figure 3.

Claim 1 Method **B** is *correct,* i.e. it delivers the same results as the naïve method **N**.

Proof Since **B** acts in a subtree of the entire octree, obviously it cannot add or delete nodes which aren't added or deleted by **N**. If **N** deletes some node *u* which **B** does not delete, then *u* is either *v* itself or an ancestor of *v*, and is therefore created again by **N** when the primitive is re-inserted at the new configuration. If **N** creates some node *u* not created by **B**, again *u* is either *v* or an ancestor thereof, and therefore must have been previously deleted by **N** when the primitive was removed from its old position. □

Claim 2 Method **B** is *optimal,* in the sense that it does not needlessly delete or create any nodes in the octree (with the possible exception of empty leaf nodes).

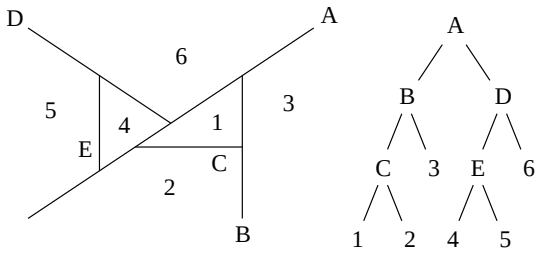


Figure 1: *A 2D BSP Tree*

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.

A more recent version of the hierarchical Z-buffer algorithm[13] uses an image-space quadtree instead of a Z-pyramid. This worsens performance to some extent, but enables effective antialiasing.

2.2. 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[14, 15] 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 Figure 1.

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 planar, polygonal faces, then the 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. 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[12] suggests using 2D BSP trees to represent images, and scan-converting them 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 representing its image. This algorithm traverses the input BSP tree recursively, from near to far, discarding all regions of space occluded by model faces. 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 specifically examining each object in these parts. Contrary to the hierarchical Z-buffer algorithm, 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.

3. Adaptation of Output-Sensitive Visibility Algorithms to Dynamic Scenes

Both output-sensitive visibility algorithms—hierarchical Z-buffer and BSP tree projection—were originally developed for static scenes. While Greene et al.[11] suggest a certain optimization for animation sequences (yielding about $\times 2$ speedup after rather significant overhead), these sequences are restricted to "walk-throughs," where the whole model is static and only the viewpoint may change between frames. For visibility culling algorithms to produce correct results, an up-to-date spatial data structure of the model has to be used. If any objects in the model move or deform then the underlying data strucure may become incorrect, and must be updated. It is not acceptable to construct it again from scratch, because, as mentioned above, this is a very expensive operation—usually more expensive than rendering a single frame by the plain Z-buffer algorithm. We discuss two ways to update the data structure efficiently: (1) minimize the time required to update the structure for a dynamic object; (2) minimize the number of dynamic objects for which the structure has to be updated.

and has yet to reach commercial systems. For example, both SGI's IRIS Performer high-performance graphics package[2] and IBM's Brush architectural and mechanical model visualizer[3] incorporate view frustum culling and multiple-resolution representations (level-of-detail switching) to speed up rendering, but neither does visibility culling.

If significant parts of the model are dynamic, then its complexity becomes even more of a problem. The known output-sensitive visibility algorithms become ineffective in such cases. Furthermore, in addition to the time it takes to render the model's visible parts, considerable time is spent just keeping it up-to-date. An example of a large model with numerous dynamic objects is an environment which multiple users roam simultaneously, such as Funkhouser's RING system[4] and Worlds Inc.'s AlphaWorld[5]; such systems have been called VRMUD (virtual reality multi-user dungeon)[6]. With existing visibility algorithms, the model in each user's workstation must reflect the other users' current whereabouts. In a distributed environment, it might take much time to update this model, and even more time to transmit the other users' movements over communication lines. The scheme we propose here not only adapts visibility culling algorithms to dynamic scenes, but also utilizes them to minimize the update overhead to those parts of the model that may be potentially visible to the user.

This rest of this paper is organized as follows: the known visibility culling algorithms for static scenes are described in the next section. We introduce our modifications of these algorithms for dynamic scenes in Section 3, and present experimental results in Section 4. In Section 5, we discuss how our techniques may be used in distributed virtual environments. Finally, Section 6 includes a discussion of the results and our plans for future work.

2. Output-Sensitive Visibility Algorithms for Static Scenes

The computational geometry community has produced some output-sensitive visibility algorithms, but they are too limited and complicated to be practical. For example, de Berg and Overmars' algorithm[7] 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 and Séquin's technique[8, 9] is more suitable for actual implementation, but it is restricted to scenes composed of rectilinear surfaces, e.g. architectural models. Their approach is to construct a *k*-D tree for the model, 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[10], but their objective is efficient calculation of form factors for radiosity.) A more serious drawback of Teller and Séquin's method is the amount of memory it consumes, which is proportional of the number of cells squared in the worst case.

Two practical and general algorithms are Greene et al.'s hierarchical Z-buffer algorithm[11] and Naylor's BSP tree projection method[12]. They are described briefly in the following subsections. Like Teller and Séquin's algorithm, they use hierarchical data structures to subdivide object space. This appears to be an intrinsic property of all output-sensitive visibility algorithms: a hierarchical spatial data structure is needed 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 DAG, and sibling nodes do not have to represent disjoint regions of space.)

2.1. The Hierarchical Z-Buffer Algorithm

The hierarchical Z-buffer algorithm[11] 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 very time-consuming, 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.)

EUROGRAPHICS '96 / J. Rossignac and F. Sillion (Guest Editors), Blackwell Publishers

Volume 15, (1996), Number 3

Output-Sensitive Visibility Algorithms for Dynamic Scenes with Applications to Virtual Reality

Oded Sudarsky and Craig Gotsman

Computer Science Department, Technion—Israel Institute of Technology, 32 000 Haifa, Israel
E-mail: {sudar, gotsman}@cs.technion.ac.il

Abstract

An output-sensitive *visibility algorithm is one whose runtime is proportional to the number of* visible *graphic primitives in a scene model—not to the* total *number of primitives, which can be much greater. The known practical output-sensitive visibility algorithms are suitable only for static scenes, because they include a heavy preprocessing stage that constructs a spatial data structure which relies on the model objects' positions. Any changes to the scene geometry might cause significant modifications to this data structure. We show how these algorithms may be adapted to dynamic scenes. Two main ideas are used: first, update the spatial data structure to reflect the dynamic objects' current positions; make this update efficient by restricting it to a small part of the data structure. Second, use* temporal bounding volumes *(TBVs) to avoid having to consider every dynamic object in each frame. The combination of these techniques yields efficient, output-sensitive visibility algorithms for scenes with multiple dynamic objects. The performance of our methods is shown to be significantly better than previous output-sensitive algorithms, intended for static scenes.*

TBVs can be adapted to applications where no prior knowledge of the objects' trajectories is available, such as virtual reality (VR), simulations etc. Furthermore, they save updates of the scene model itself, not just of the auxiliary data structure used by the visibility algorithm. They can therefore be used to greatly reduce the communications overhead in client-server VR systems, as well as in general distributed virtual environments.

Keywords: visibility culling; output-sensitive hidden surface removal; virtual reality; distributed multi-user virtual environments; client-server design

1. Introduction

Visibility calculation is one of the most important tasks in computer graphics. Given a geometric model of a scene and a viewpoint, the goal of visibility calculation (also known as hidden surface removal) is to find which parts of the model are visible from the viewpoint. The performance of the visibility calculation stage can largely affect that of the entire rendering process.

The visibility calculation's runtime can become a problem with big, complex models featuring large numbers of graphic primitives. Consider, for example, a detailed model of a big building. It might include millions of polygons, but only a small fraction of them will be visible from any single viewpoint. In such scenes, it would be preferable if the visibility calculation algorithm's runtime were linearly proportional just to the number of visible primitives, rather than the total number of primitives in the model.

Definition A visibility algorithm is called *output-sensitive* if its runtime per frame (excluding any initializations) is $O(n+f(N))$, where N is the number of primitives in the entire model, n is the number of visible primitives and $f(N) \ll N$.

$f(N)$ is the (inevitable) overhead imposed by the model. An output-sensitive visibility calculation algorithm is also called a *visibility culling* algorithm[1].

Example The Z-buffer visibility algorithm is *not* output-sensitive, because it examines each polygon in the model. Even if every polygon is handled very quickly (e.g. in hardware), the runtime is still proportional to the total number of polygons.

In a recent survey on real-time 3D rendering[1], Heckbert and Garland observe that output-sensitive visibility algorithms are essential in future generation graphics systems. Research on output-sensitive visibility calculation has only began recently,