

Output-Sensitive Rendering and Communication in Dynamic Virtual Environments

Oded Sudarsky and Craig Gotsman
Computer Science Department
Technion—Israel Institute of Technology
32 000 Haifa, Israel

ABSTRACT

The efficient rendering of large dynamic scenes is an important open problem. Optimization techniques for static scenes, such as output-sensitive visibility calculation, must be carefully adapted to dynamic models in order to remain effective. Distributed virtual environments pose a particular difficulty, because communication between the users must be minimized in addition to each user's rendering time. We show how output-sensitive visibility calculation algorithms can be adapted to dynamic scenes, and used to reduce the communication requirements between workstations in a distributed virtual environment. The solution is based on temporal bounding volumes, guaranteed to contain the dynamic objects for some period of time. These volumes are inserted into a visibility algorithm's main data structure instead of hidden dynamic objects. Subsequently a dynamic object is ignored until its bounding volume becomes visible or is no longer guaranteed to contain the object. In a distributed virtual environment, this saves not only the rendering of the object, but its update through a communication network too. We show an algorithm which combines this method with BSP tree based output-sensitive visibility calculation, and report on the implementation of our system.

1 Introduction

Typical virtual reality (VR) applications feature large dynamic models, i.e. scenes containing moving objects. This trend manifests in VRML 2.0 browsers and authoring tools, such as Silicon Graphics' Cosmo software [18] and Sony's Community Place [19]. While such systems are highly desired by the user community, their future

success largely depends on their performance. Currently used software and hardware architectures are only capable of displaying models of limited complexity at reasonable screen refresh rates. Developing techniques to improve graphics performance will provide better response times and allow more complex, realistic models to be displayed.

The issue of rendering optimization is quite well understood for *static* scenes. Among the techniques used for such scenes are view frustum culling, visibility culling and level-of-detail switching [13]. However, little work has been done so far concerning the generalization of these methods for dynamic models, which are the bread and butter of VR applications.

Consider visibility culling techniques [6, 10, 11, 15], also known as occlusion culling or output-sensitive visibility algorithms. These techniques utilize the occlusions in a scene to render it in time proportional to the number of *visible* objects, rather than the *total* number of objects, which can be greater by orders of magnitude. However, these algorithms are designed for static models: as a preprocessing stage, they create a data structure based on the scene, which is assumed to be unchanging. To efficiently render dynamic scenes, these algorithms have to be modified properly; doing so in a naïve way may result in performance which is even worse than if no visibility culling technique was used at all.

One class of applications that naturally give rise to complex scenes with multiple dynamic objects are distributed virtual environments [7, 8, 19, 23]. Such environments are gaining greater popularity as cyberspace continues to evolve. In these systems, numerous users can travel through a shared scene. Each user is virtually represented as a graphic character called an *avatar*. The avatar's position and orientation in the virtual world are controlled by the user through his or her workstation. The user usually does not see his or her own avatar, but rather sees the world through the avatar's eyes. This view may, of course, include avatars of other users. See Figure 1.

Typically, most of the virtual world through which the avatars move is static scenery. While some of it

Permission to make digital/hard copies of all or part of this material for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copy-right notice, the title of the publication and its date appear, and notice is given that copyright is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires specific permission and/or fee

ACM VRST '97 Lausanne Switzerland
Copyright 1997 ACM 0-89791-953-x/97/9..\$3.50

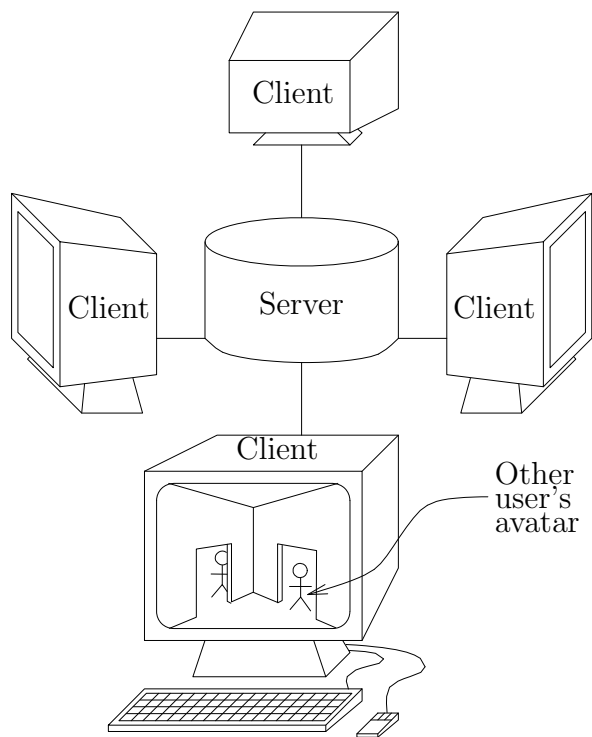


Figure 1: A typical distributed virtual environment

may be modified by the users (e.g. constructing a new building), the fraction of the scenery that is changing, as well as the rate at which it is changing, are usually far smaller than the motions of the avatars.

Distributed virtual environments pose particularly challenging optimization problems. As the avatars are controlled interactively by human users, and do not have *a priori* scripts, their behaviors might be unpredictable. Therefore 4D methods, such as Glassner’s spacetime technique [9], cannot be used. Additionally, since the users interact via a communication network of limited bandwidth, care must be taken to minimize communication requirements between the users.

Most current distributed virtual environments are implemented as client-server systems. Each user’s workstation acts as client; a central server maintains the scene, keeping track of each user’s position and sending model update messages to the clients (see Figure 1). The drawback of such a system is its lack of scalability: as the number of users grows, the server becomes congested, degrading the system’s performance. Alternatives to the client-server configuration are a network of communicating servers and a completely decentralized system where all workstations may communicate with each other.

Whatever system configuration is chosen, the overall communication requirements generally grow as the square of the number of users in the environment, be-

cause each user is informed of the other users’ whereabouts. Therefore, this problem will not be solved by faster communication alone. For example, quadrupling communication throughput will only allow twice as many users to share a virtual environment at a given response time.

A potential solution to this problem is to avoid transmitting unnecessary update messages. In most shared virtual environments of sufficient size and complexity, each user will usually see only a small proportion of the model and of the other users’ avatars. The rest of the users will normally be hidden by relatively static parts of the model, e.g. by walls of a building model, that change very little (if at all) and constitute static scenery. There is no need to waste communication resources on transmitting update messages between such non-intervisible users. A visibility calculation algorithm can be used to determine user-to-user visibility, and to eliminate unneeded messages.

In the following sections, we show how visibility culling algorithms can be adapted to dynamic scenes, and used to avoid unnecessary messages between non-intervisible users in shared environments. Occlusions will thus be utilized both to optimize the rendering at each station and to reduce communication between the stations. The extension to dynamic scene models has been implemented for octree based and BSP tree based visibility culling algorithms.

2 Related work

2.1 Visibility culling for static scenes

A few visibility culling algorithms have been presented in recent years. All these algorithms construct a spatial hierarchical data structure as a preprocessing stage: a binary space partitioning tree (BSP tree) in Naylor’s partitioning tree visible surface algorithm [15]; an octree or a k -D tree in Greene et al.’s hierarchical Z-buffer algorithm [11] and in Coorg and Teller’s object-space visibility algorithm [6]. Given a viewpoint, these algorithms proceed by traversing the spatial hierarchical data structure in a top-down, near-to-far order, terminating the recursion as soon as an entire subtree is occluded by the previously projected objects. The algorithms differ mainly in the manner in which these occlusions are detected: Naylor uses BSP tree containment, Greene et al. utilize a Z-pyramid, and Coorg and Teller perform object-space analysis. Incidentally, all three algorithms perform view frustum culling as well as visibility culling, by terminating recursion at tree nodes which are outside the field of view.

Each of these three visibility culling algorithms has its drawbacks. Naylor requires that the scene model itself be represented as a BSP tree, with boolean “in/out” attributes at the leaves and with a color associated with each “in” leaf. This tree of the scene is transformed into

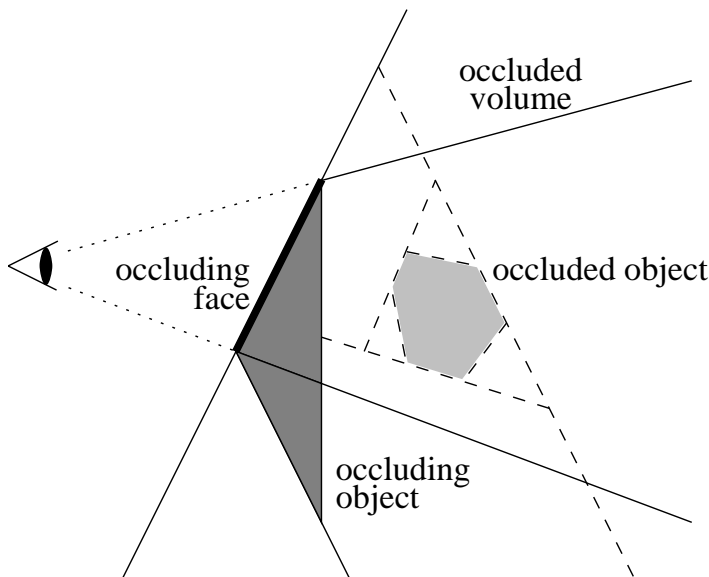


Figure 2: Naylor's BSP tree projection algorithm

a tree describing its image by uniting it with the volume occluded by each face encountered during the traversal of the tree (see Figure 2); this projection process is output-sensitive if the initial BSP tree is constructed as a series of approximations to the modeled objects [16]. The requirement for a BSP tree restricts the geometry to planar polygons, and makes the use of surface attributes such as normal directions and texture maps more complicated. Furthermore, the overwhelming majority of the models available worldwide (e.g. VRML models on the Web) are not given as BSP trees, but as boundary representations (B-reps). Such models have to be converted to BSP trees in a separate process. The conversion process [21] assumes that there are polygons on all of the objects' boundaries, only on the objects' boundaries, and that all of the polygons' normals consistently point out of the objects. Many of the available models fail to meet these requirements—in fact, very few do. On the other hand, Naylor's algorithm is elegant because it uses the same type of data structure—a BSP tree—to represent both the scene itself and its image. Since it is an object-space algorithm, it is not prone to aliasing artifacts, and its runtime is independent of image size. It does not depend on any hardware capabilities such as Z-buffering; any display which draws filled convex polygons can be used, e.g. X11 terminals and low-end PCs.

Greene et al. rely on a Z-pyramid constructed on top of the Z-buffer. Updating these structures 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 seconds per animation frame for a 512×512 pixel 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 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.

Coorg and Teller estimate a node's 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 conservatively detects only some of the occlusions: it never reports a visible octree node as invisible, but it may erroneously classify some invisible nodes as visible. It is only guaranteed to correctly detect occlusions by a single convex polygon, or by a mesh of edge-connected polygons with a convex silhouette. Consequently, the visibility algorithm needlessly traverses some hidden regions of the octree. Another disadvantage of the algorithm is that it involves a preprocessing stage which finds, for each octree leaf node, the set of scene polygons that appear relatively large from viewpoints inside the leaf. This takes some time and space, and has to be repeated if any of the scenery changes, e.g. by users doing construction work in the shared virtual environment.

2.2 Octree-based visibility culling for dynamic scenes

A disadvantage common to all of the above visibility culling algorithms is that they are unsuitable for dynamic scenes. While they 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 else but the viewpoint moves in the scene, then the spatial hierarchical data structure used by the visibility culling algorithm becomes outdated, possibly resulting in incorrect images. The solution is, of course, to update the data structure; initializing it again from scratch would be too wasteful—much slower than just displaying everything by the plain Z-buffer algorithm. To preserve output sensitivity, the update should be performed only on the visible regions of the data structure. No time should be wasted on updating it for invisible dynamic objects, e.g. for avatars of other users which are moving behind occluding walls.

We have introduced [20] a technique to update an

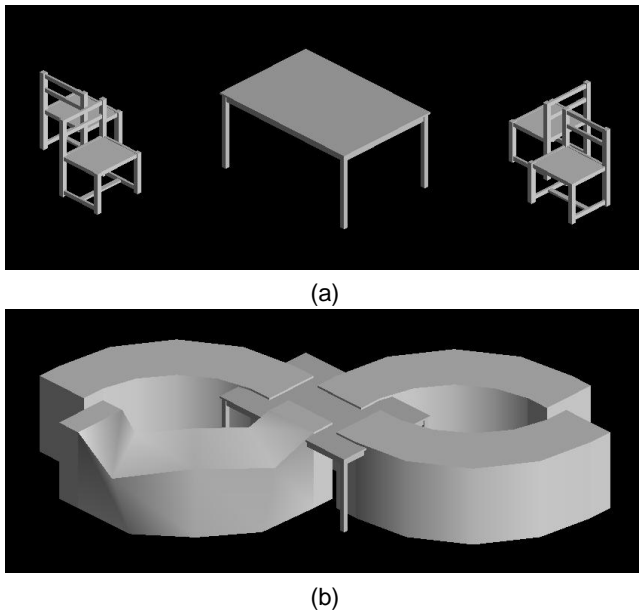


Figure 3: (a) A scene with dynamic objects (chairs) moving along preset trajectories. (b) Sweep surface temporal bounding volumes for the chairs moving towards the table. Note that the closer chair on the left wobbles during its movement.

octree for the movements of dynamic objects. This technique avoids wasting time on hidden dynamic objects by utilizing *temporal bounding volumes* (TBVs). These are volumes guaranteed to contain a given dynamic object from the moment of their construction until some later time, based on some known constraints on the object’s behavior. For example, for objects moving along preset trajectories, sweep surfaces can be used (see Figure 3); if only maximum velocities or maximum accelerations are known, then spheres may be employed as TBVs. Generally, we assume some bounding volume can be found for each dynamic object until any desired moment in the future.

The TBVs are inserted into the octree in lieu of dynamic objects which the visibility culling algorithm deems to be invisible. Subsequently, a hidden dynamic object is ignored until such time as its TBV expires, i.e. is no longer guaranteed to contain the object, or until the visibility culling algorithm determines that the TBV is visible, whichever comes first. Several strategies to choose expiration times for TBVs were discussed; the most advanced of these is adaptive expiry, in which a TBV’s validity period is shortened if the previous TBV’s period was too long (i.e. the previous TBV became visible before it expired), lengthened if too short (in the opposite case).

To test the performance of this technique compared to other rendering techniques, we conducted experiments on the test scene shown in Figure 4, using a Silicon

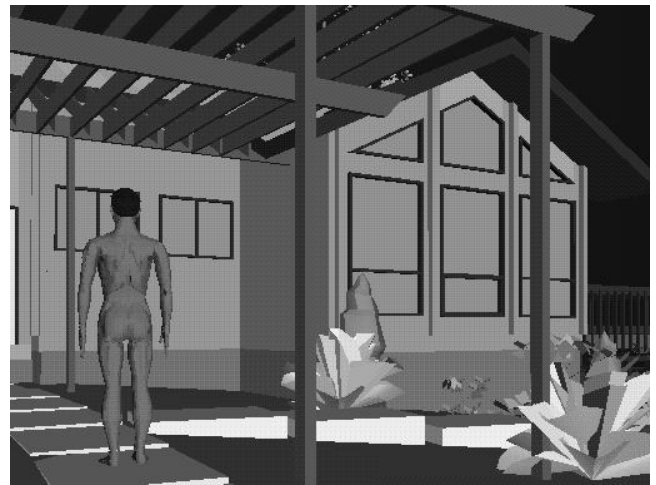


Figure 4: A test scene used in our experiments

Graphics Indy R5000. 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. As Figure 5 shows, the runtime of the plain Z-buffer algorithm (ZB) is linearly proportional to the total number of objects in the scene. The hierarchical Z-buffer algorithm (HZB) requires that the octree be updated for every dynamic object, and therefore does even worse than ZB. Our TBV technique updates the octree only for the visible dynamic objects, and its runtime is almost constant in comparison to ZB and HZB. 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.

A significant advantage of our dynamic scene visibility culling method is that it not only avoids unnecessary updates of the spatial data structure used by the visibility algorithm; it also avoids needless updates of the invisible objects themselves. In other words, an invisible dynamic object (say, some other user’s avatar, hidden by a wall) can be completely ignored most of the time—not just eliminated from the rendering process, but disregarded altogether. This can save a lot of time if this object would be very costly to update, e.g. over a busy communication line. Thus our method applies to distributed virtual environments: it uses a visibility culling algorithm to cut down on the number of unnecessary transmissions between users that do not see each other.

While it may appear that the *a priori* knowledge of some motion constraints by which TBVs can be calculated is an excessive requirement, this is not the case. Interactive virtual environments generally have user in-

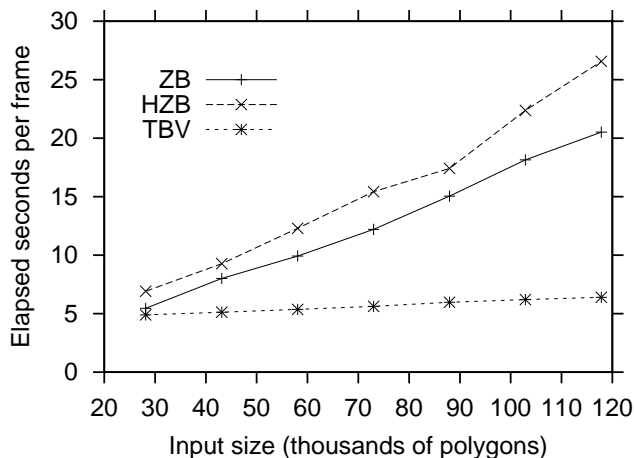


Figure 5: Performance of algorithms on a test scene with a fixed number of static and visible dynamic objects and a varying number of hidden dynamic objects

interfaces that impose such constraints: a user moving through the environment cannot exceed a certain speed. If the system allows users to “teleport” to a remote location, this speed 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 (both the teleporting user’s display and the displays of other users into whose vicinity this user teleports). Furthermore, one may use *probabilistic* bounding volumes, i.e. volumes that are not guaranteed to contain the dynamic object throughout their validity period, but are only assumed to do so. A TBV for which this assumption fails is treated as an expired TBV: its dynamic object is no longer ignored, but is inserted into the octree and tested for visibility.

2.3 BSP trees of dynamic scenes

Since the BSP tree visibility culling algorithm has several advantages over the octree-based algorithms, namely independence of graphics hardware and image size, it would be advantageous to use the same ideas in Naylor’s algorithm. It should be noted that this is not a straightforward generalization of the same technique, because BSP trees are inherently different from octrees and k -D tree: BSP trees, as used by the visibility algorithm, represent the objects themselves (with leaf “in/out” attributes), whereas octrees and k -D trees are merely auxiliary data structures, supplementing a B-rep.

Chrysanthou and Slater [4, 5] and Agarwal et al. [1] have proposed algorithms to maintain dynamic BSP trees. However, their trees are of the wrong kind for visibility culling purposes: they keep the objects’ boundaries 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 [22] 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, proposed by us [20], 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. In Section 3, this method will be used as a basis for a new, output-sensitive visibility algorithm for dynamic scenes.

These techniques can be used to dynamically update a BSP tree which is subsequently displayed by Naylor’s visibility 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 algorithms, followed by visibility 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), whereas we would like to achieve $O(v)$. For such performance, the BSP tree update must be closely coupled with BSP tree visibility culling.

2.4 Message reduction in virtual environment systems

As mentioned in Section 2.2, a major advantage of our dynamic scenes visibility culling technique is that it can also eliminate a significant number of unnecessary messages—those messages that would otherwise be transmitted to update hidden objects. This technique may be used instead of or in addition to other methods to reduce the required number of update messages in shared virtual environments. These 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 [14] uses this method with 4 km hexagonal cells, Spline [2] allows arbitrary polyhedral cells given as BSP trees, and Worlds Chat [23] uses cells just big enough to contain six other users and restricted to one room. 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 Worlds Chat, this results in an incorrect image; if the cells are made very large to minimize this effect, as in NPSNET, then the user 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, 12] use a message multicasting protocol, so when a user updates other users of his movement, he only sends 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 multi-user 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 approximation of his position, based on his prior behavior. The user himself performs the same approximation, and sends his updated position when the approximated position differs from the real one by more than some threshold (or after some fixed time limit).

In the RING system [8], 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 movements of users in other offices down the hall, even though none of them are visible to him because his office door (a dynamic object) is closed at the moment.

3 BSP tree visibility culling for dynamic scenes

We hereby introduce an algorithm that performs output-sensitive visibility calculation of a BSP tree representing a scene with multiple dynamic objects. It is based on the method of uniting separate, transformed BSP trees of individual objects into a tree of the entire scene (as described in Section 2.3), then rendering this tree using Naylor's visibility technique [15]. However, it avoids wasting time on hidden dynamic objects. Like the octree-based method [20], this algorithm not only optimizes the rendering time at each user's workstation, but also reduces communication overhead in multi-user distributed virtual environments by utilizing occlusions and known constraints on objects' motions.

At each user's station, the following data structures are maintained:

D a set of all the dynamic objects, each having a unique identifier (ID) and a time of last observation. Those

of the dynamic objects that are hidden also have a TBV, specified by a BSP tree; an expiration time for the TBV; and a set of pointers to the leaves of **S** (see below) that intersect the TBV.

S a BSP tree that is the union of the static scenery and the TBVs of hidden dynamic objects in **D**. A set of dynamic object IDs is associated with each leaf of **S**; an object's ID is in the set if it is invisible and its TBV intersects the leaf.

T a BSP tree that represents the entire scene, including the static scenery, visible dynamic objects and TBVs of hidden dynamic objects. Each leaf of **T** has a set of dynamic object IDs; an ID is in the set if the corresponding dynamic object is either visible and intersects the leaf, or hidden and its TBV intersects the leaf.

Q an event queue of TBV expiration events for hidden dynamic objects.

V a set of IDs of visible dynamic objects.

Each of the "dynamic objects" referred to above can be either an avatar of some other user or an autonomously moving object, controlled by a program, e.g. Java-controlled objects in VRML 2.0 models.

The algorithm uses two subroutines: `del_TBV` (delete temporal bounding volume) accepts an ID of a hidden dynamic object, and changes the object's status from hidden to (potentially) visible; `uni_vis` (unite a visible object into the scene) handles a visible object.

`del_TBV(ID)`:

1. Delete ID from the leaves of **S** that contain ID.
2. Merge leaves of **S**, if possible.
3. $\mathbf{V} \leftarrow \mathbf{V} \cup \{\text{ID}\}$.

In step 1 of `del_TBV`, the leaves of **S** which contain ID are given by the set of leaves maintained with the corresponding dynamic object in **D**. In step 2, the leaves that should be merged are siblings that contain the same set of IDs after the deletion of step 1. The merge proceeds bottom-up, terminating at siblings that are either not both leaves or have different ID sets.

`uni_vis(ID)`:

1. Obtain a BSP tree **B** representing the current configuration of the dynamic object corresponding to ID.
2. $\mathbf{T} \leftarrow \mathbf{T} \cup \mathbf{B}$. During the union, insert ID into every leaf of **T** that **B** intersects.

In step 1 of `uni_vis`, **B** can be obtained, e.g., through a communication link. The exact content of the communication may depend on the nature of the dynamic

object. For example, if the object is rigid, then all that needs to be sent (after the first frame) are translation and rotation parameters, possibly as a 4×4 homogeneous transformation matrix A ; the tree \mathbf{B} is obtained by multiplying every coordinate in the object's BSP tree by A , and every plane equation by A^{-1} . 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 BSP tree representing the object's current form.

The union operation in step 2 can be performed as discussed by Naylor et al. [17].

At each user's workstation, all the dynamic objects in \mathbf{D} are initially marked as visible, and have a time of last observation earlier than the first frame. \mathbf{S} is initialized to the union of all the static objects, \mathbf{T} and \mathbf{Q} are empty, and \mathbf{V} initially contains the IDs of all the dynamic objects.

At every frame, the following steps are performed at each station:

1. For each object ID of an expired TBV, as determined by \mathbf{Q} , do `del_TBV(ID)`.
2. $\mathbf{T} \leftarrow \mathbf{S}$.
3. For each ID in \mathbf{V} do `uni_vis(ID)`.
4. Operate Naylor's visibility algorithm on \mathbf{T} , displaying visible faces. At each leaf of \mathbf{T} encountered during this traversal, for each ID of an invisible object whose TBV intersects the leaf, do:
 - (a) `del_TBV(ID)`;
 - (b) `uni_vis(ID)`.

For each ID of a visible object intersecting the leaf, update the time of last observation associated with the object in \mathbf{D} .
5. For each dynamic object in \mathbf{V} whose time of last observation is earlier than the current frame, do:
 - (a) Obtain a TBV for the object until some time in the future.
 - (b) Unite the TBV into \mathbf{S} ; for every leaf of \mathbf{S} that intersects the TBV, add the object's ID to the leaf, and add a pointer to the leaf into the object's set of leaf pointers.
 - (c) Insert the TBV's expiration event into \mathbf{Q} .
 - (d) Delete the object's ID from \mathbf{V} .

Step 1 of the algorithm handles hidden dynamic objects whose TBVs have expired by deleting their TBVs and moving them to the set \mathbf{V} . Note that \mathbf{V} contains *potentially* visible dynamic objects, rather than objects which are certainly visible; at this stage it is too early

to determine certain visibility, so we simply bundle the objects whose TBVs have expired with those that are potentially visible because they were visible in the previous frame.

In steps 2 and 3, the BSP tree \mathbf{T} representing the entire scene is constructed. It is used in step 4, the heart of the algorithm.

Step 4 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 an hierarchical data structure and modifies it at the same time: although the union operation in step 4(b) preserves the overall structure of the BSP tree, it might replace leaf nodes with subtrees; if this happens, the new subtree should be traversed too.

Finally, step 5 handles dynamic objects which cease being visible. If such an object is controlled by another workstation, then that station should provide a BSP tree representing the TBV for the object, upon request; the request should also specify the time in the future until which the provided TBV should be valid. Note that the algorithm performs less 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 box).

The algorithm achieves the goal of being output-sensitive with respect to the number of dynamic objects by ignoring such objects unless they are 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 invisible dynamic objects, not even to discover that they are invisible; they are simply not reached during the traversal of the BSP tree. Hidden dynamic objects require processing only if their TBVs expire or become exposed. If TBVs are chosen with sufficient care (e.g. if adaptive expiry, discussed in Section 2.2, is used), then TBV exposures only occur for a minority of the invisible objects, and TBV expiries become less frequent with time.

Static objects may 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 (although relatively few of the static objects may be visible). Therefore the algorithm treats static objects differently from dynamic ones.

4 Current status and ongoing work

We have implemented the extension of the hierarchical Z-buffer algorithm to dynamic scenes, discussed in Section 2.2. The experimental results reported in that section were obtained using this implementation. Currently the system executes on a single workstation: the viewpoint moves through the scene at the user's control or along a preset trajectory, and all the other users are simulated by the program.

We have also implemented the BSP tree based algorithm described in Section 3. Generally, it performs slower than the octree based algorithm, due to the high overhead incurred by the numerical calculations involved in BSP tree operations. However, it does exhibit the same output-sensitivity with respect to the number of dynamic objects as the octree based algorithm. Its performance is almost independent of the number of hidden dynamic objects, whereas the more naïve algorithm of uniting all the objects into a BSP tree of the whole scene, described in Section 2.3, needlessly spends time on these objects.

Ongoing work includes the implementation a distributed, multi-user virtual environment that uses similar concepts to optimize both rendering and communication. A working prototype is expected shortly.

Acknowledgements

We are grateful to Gershon Elber for his kind assistance with the *IRIT* solid modeler, which was used in the extension of the hierarchical Z-buffer algorithm to dynamic scenes.

REFERENCES

1. P. K. Agarwal, J. Erickson, and L. J. Guibas. Kinetic binary space partitions for intersecting segments and disjoint triangles. (<http://www.cs.duke.edu/~pankaj/papers/kinetic-3d.ps.gz>). Submitted to FOCS 97, Apr. 1997.
2. 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-95-16, Mitsubishi Electric Research Laboratories Cambridge Research Center, 201 Broadway, Cambridge, MA 02139, Nov. 1995.
3. C. Carlsson and O. Hagsand. DIVE—a platform for multi-user virtual environments. *Computers & Graphics*, 17(6):663–669, 1993.
4. Y. Chrysanthou and M. Slater. Computing dynamic changes to BSP trees. In *Proceedings of Eurographics '92*, pages 321–332, Cambridge, U.K., Sept. 1992. Blackwell Publishers. *Computer Graphics Forum*, 11(3).
5. 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*, pages 45–50, Monterey, California, Apr. 1995. ACM SIGGRAPH.
6. S. Coorg and S. Teller. A spatially and temporally coherent object space visibility algorithm. Technical Report TM-546, MIT, Feb. 1996.
7. 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.
8. T. A. Funkhouser. RING: A client-server system for multi-user virtual environments. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, pages 85–92, Monterey, California, Apr. 1995. ACM SIGGRAPH.
9. A. S. Glassner. Spacetime ray tracing for animation. *IEEE Computer Graphics and Applications*, pages 60–70, Mar. 1988.
10. N. Greene. Hierarchical polygon tiling with coverage masks. In *SIGGRAPH '96 Conference Proceedings*, pages 65–74, New Orleans, Louisiana, Aug. 1996. *ACM Computer Graphics*, 30(4).
11. N. Greene, M. Kass, and G. Miller. Hierarchical Z-buffer visibility. In *SIGGRAPH '93 Conference Proceedings*, pages 231–238, Anaheim, California, Aug. 1993. *ACM Computer Graphics*, 27(4).
12. O. Hagsand. Interactive multiuser VEs in the DIVE system. *IEEE MultiMedia*, 3(1):30–39, Spring 1996.
13. P. S. Heckbert and M. Garland. Multiresolution modeling for fast rendering. In *Proceedings of Graphics Interface '94*, Banff, Alberta, May 1994.
14. 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. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, pages 93–94, Monterey, California, Apr. 1995. ACM SIGGRAPH.
15. B. F. Naylor. Partitioning tree image representation and generation from 3D geometric models. In *Proceedings of Graphics Interface '92*, pages 201–212, Vancouver, May 1992.
16. B. F. Naylor. Constructing good partitioning trees. In *Proceedings of Graphics Interface '93*, pages 181–191, Toronto, May 1993.
17. B. F. Naylor, J. Amanatides, and W. C. Thibault. Merging BSP trees yields polyhedral set operations. In *SIGGRAPH '90 Conference Proceedings*, pages 115–124, Dallas, Aug. 1990. *ACM Computer Graphics*, 24(4).
18. Silicon Graphics, Inc. Cosmo Software. (<http://cosmo.sgi.com/>).
19. Sony Corporation. Virtual Society on the Web. (<http://sonypic.com/vs/>).
20. O. Sudarsky and C. Gotsman. Output-sensitive visibility algorithms for dynamic scenes with applications to virtual reality. In *Proceedings of Eurographics '96*, Poitiers, France, Aug. 1996. Blackwell Publishers. *Computer Graphics Forum*, 15(3).
21. W. C. Thibault and B. F. Naylor. Set operations on polyhedra using binary space partitioning trees. In *SIGGRAPH '87 Conference Proceedings*, pages 153–162, July 1987. *ACM Computer Graphics*, 21(4).
22. E. Torres. Optimization of the binary space partition algorithm (BSP) for the visualization of dynamic scenes. In *Proceedings of Eurographics '90*, pages 507–518, Montreux, Switzerland, Sept. 1990. Elsevier Science Publishers B.V. (North-Holland).
23. Worlds Inc. Worlds Chat™. (<http://www.worlds.net/products/wchat/>).