Helsinki University of Technology 4.5.2003
Telecommunications Software and Multimedia Laboratory
Tik-111.500 Computer Graphics Seminar
Spring 2003 varvana.myllarniemi@hut.fi

Dynamic Scene Occlusion Culling

Varvana Myllarniemi
50127F

Dynamic Scene Occlusion Culling

Varvana Myllarniemi
Helsinki University of Technology
vmyllarn@cc.hut.fi

Abstract

As the need for real-time 3D graphical applications increases, so does increase the need for effective visibility culling algorithms. Complex, dense models require effective occlusion culling; this requires output-sensitivity, which means that that the runtime depends on the amount of the visible primitives. This paper discusses several dynamic occlusion culling algorithms, remembering that the word dynamic has many meanings in this case. To achieve output sensitivity, most algorithms use some hierarchical data structure. But there are other solutions available too.

1 INTRODUCTION

The means to produce real-time 3D graphical applications have improved over the last few years tremendously. Hardware is becoming more and more powerful as new processors, chipsets and memories are introduced. But the needs and expectations of the users have grown even more rapidly. Scenes must be modeled with fine detail, which leads to large models with many polygons. Users often want interaction with the model. And all this has to be rendered at speeds that enable real-time animation - which means at least 10 frames per second.

The hardware improvements alone do not suffice. The size of the models that meet the requirements grow more rapidly than the size of the models that can be rendered. The solution to this situation is *output sensitivity*. Output sensitivity means that the runtime depends on the size of the output, not on the size of the input. Put it another way, an algorithm is output sensitive if it has a runtime per frame that is $O(n+f(N))$, where n is number of visible primitives, N is total number of all primitives and $f(N) \ll N$ is the overhead imposed by the model. (Sudarsky & Gotsman, 1996)

This can be achieved several ways. One technique is level-of-detail control, which means that objects that are farther away are rendered with less detail. Another technique is to render nothing that is not visible to the viewer. The process of leaving out the non-visible parts from the model is called *visibility culling*. Visibility culling actually contains three different mechanisms: *back-face culling*, *view-frustrum culling* and *occlusion culling*. Back-face culling means leaving out the back-faces of the polygons, view-frustrum culling means ignoring all objects that are not contained in the view frustrum. Occlusion culling means leaving out all the polygons that are *occluded* by other polygons. An object is occluded by another object if it is totally hidden behind it. Thus the *occluder* conceals the occluded object. Usually occlusion culling techniques include view-frustrum culling as a part of the algorithm. Occlusion culling is especially important in dense models, like cities or buildings, where a large portion of the objects are occluded.

This paper concentrates on dynamic occlusion culling algorithms. Word *dynamic* has unfortunately many meanings. Sometimes dynamic means that the culling computation is done on-the-fly, depending on the viewpoint, and the result is a real-time animated model in which the user can walk or fly around freely. But often dynamic means that the objects in the model are not static but can move just like the camera moves. This paper discusses both cases.

Chapter 2 discusses different algorithms that are based on hierarchical representations of the model. In chapter 3, one tries to adopt two of these techniques to handle models with moving objects. Chapter 4 introduces different approaches to handle the moving objects. In chapter 5, conclusions are drawn based upon the contents of the previous chapters.

2 HIERARCHICAL REAL-TIME ALGORITHMS

For algorithms to be output sensitive, they cannot iterate through all objects and decide the ones that are visible. Instead, they must reject large portions of the object-space with one query, if these large portions are hidden. This usually requires the use of some hierarchical data structures.

The algorithms in this section do the visibility culling on-the-fly, enabling (at least almost) real-time rendering of models. They all use some kind of hierarchical data structure that divides the object-space into regions. This means that the model must be preprocessed to obtain this hierarchical representation. If an object in the model moved, the whole structure should be preprocessed again. Since the preprocessing is very time-consuming, these algorithms cannot be used in models where objects are moving.

The algorithms presented are Hierarchical Z-buffer (Greene *et al.* , 1993), BSP tree projection algorithm (Naylor, 1992) and cPLP (Klosowski & Silva, 2001). Other similar algorithms use for example hierarchical occlusion maps (Zhang *et al.* , 1997) and kD-trees (Coorg & Teller, 1997).

2.1 Hierarchical Z-buffer

Hierarchical Z-buffer algorithm (HZB), first introduced in (Greene *et al.* , 1993), is an improvement of traditional Z-buffer algorithm. It exploits two hierarchical data structures: an octree in object-space and a Z-pyramid in image-space (see figure 1).

The octree contains the whole model inside a cube. The cube is then divided into eight subcubes (see figure 1), which are then divided again. Each primitive in the model is associated with the smallest possible enclosing cube. When the construction is ready, each octree cube knows its direct children and the polygons associated to it. This construction of the model into an octree is done at a preprocessing stage.

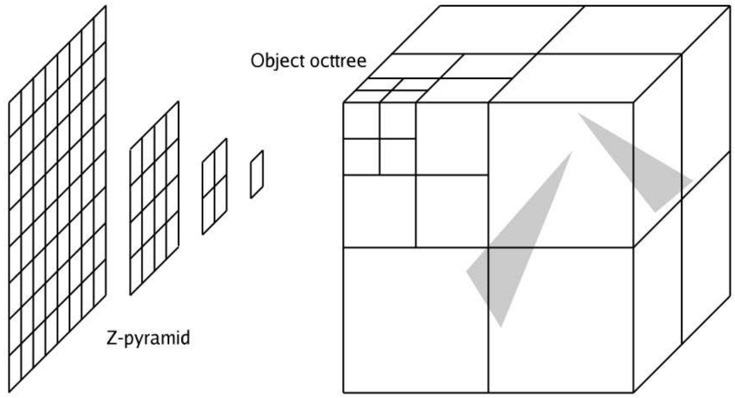

Figure 1: The hierarchical Z-buffer algorithm uses two data structures: an octree contains the objects in the model, and Z-pyramid contains the values of Z-buffer.

Using the octree, the visibility algorithm is intuitively clear. We start with the root node in the octree (the largest cube) the following procedure. (1) Check if the cube intersects with the viewing frustrum. If not, quit. (2) Scan convert the faces of the cube to find out whether any part of the cube is visible. If not, quit. (3) Scan convert the polygons associated with the cube, recursively process the child nodes of the cube in front-to-back order.

However, step 2 in this procedure might take quite long time using traditional Z-buffer scan conversion. To reduce the cost of determining the cube visibility, HZB uses a Z-pyramid. The lowest level of this pyramid contains the values from the original Z-buffer. Each group of 2x2 values on one level is then combined into one value on the next level by choosing the farthest value of Z from these four values. Thus the value on top of the Z-pyramid represents the farthest Z-value in the whole image (see figure 1).

In order to use the Z pyramid to test the visibility, we do the following for the cube faces and the polygons of the mode. (1) Find the finest-level value of the pyramid whose corresponding image region covers the projection of the polygon. (2) If this Z-value is nearer than the closest Z-value of the polygon, the polygon is totally hidden and we can quit. (3) Otherwise divide the projection of polygon into four and move to finer level of the pyramid. (4) If we reach the lowest level of the pyramid, we have found a visible pixel.

In addition, HZB uses a temporal coherence list that contains all visible cubes from the previous frame. Before starting the algorithm, all nodes in the list are rendered. This forms an initial Z-buffer, which can be used as a basis for the Z-pyramid. If there is enough coherence in the frames, it is quite likely that all visible geometry has been rendered. This makes the Z-pyramid test very effective. After rendering the frame, we remove the non-visible cubes from the list by using the Z-pyramid test on them.

The main weakness of HZB is its reliance on Z-pyramid. Z-pyramid is too slow to be constructed in software, and current hardware is either not capable of or too slow in answering the so called Z-query: given the current Z-buffer, are there any visible pixels in a polygon? This means that HZB cannot really come up to interactive speeds. Some newest hardware chipsets are now becoming available that support the needed Z-buffer query. But it seems that there are better and faster algorithms that exploit this Z-query (see for example chapter 2.3).

2.2 BSP tree projection algorithm

BSP tree projection algorithm (Naylor, 1992) uses the same ideas as HZB: eliminate large portions of model with hierarchical structures. Instead of an octree, it uses a more complex structure: binary space partitioning tree (BSP tree).

A BSP tree is a binary tree in which each node represents a plane (or a line in 2D). The left subtree of a node corresponds to the negative half-space of the plane and the right subtree to the positive half-space. For example, figure 2 shows a 2D BSP tree.

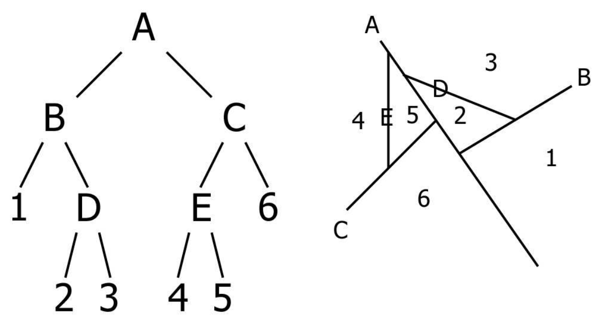

Figure 2: A BSP tree and the corresponding partitioning of a plane. Lines are denoted by letters and regions by numbers.

If the model consists entirely of polyhedrons, then the whole model could be represented as a BSP tree. If we attach a simple `in/out` variable with the leaf nodes, then the `in` region corresponds to a region inside a polyhedron. This means that no additional data structures are needed: BSP tree *is* the model itself.

The algorithm starts to transform the BSP tree recursively from near to far. Whenever the algorithm encounters a face of the model (a plane that has a region `in` as its direct child), it constructs a small BSP tree that represents the volume occluded by that face. (See figure 3.) The formed BSP tree is united with the model tree, thus eliminating redundant nodes. In the union, every subtree that has identical `in/out` regions is collapsed into a single leaf. This means that the whole volume that is occluded by the face is collapsed into a single leaf.

The reliance on the BSP tree as a model is the main weakness of the BSP projection algorithm. Majority of models are given in other formats, such as boundary representations (B-reps). These models must be converted to BSP trees, and it is a bit complicated process. First of all it requires the use of heuristic choices presented by Naylor, otherwise the resulting algorithm might not be output sensitive. Second, it assumes that there are polygons on all object boundaries, only on object boundaries and that all polygon normals point out of the objects. As Sudarsky and Gotsman (Sudarsky & Gotsman, 1997) point out, actually only a few models meet these requirements.

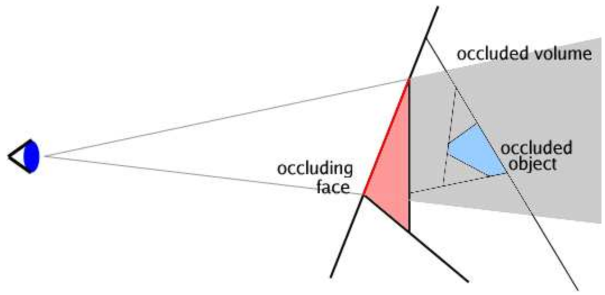

Figure 3: An occluding object and its occluding face (red) creates an occluded volume (gray) that hides the occluded object (blue).

2.3 cPLP

Algorithms in chapters 2.1 and 2.2 approached the visibility problem quite similarly. Another approach is presented in (Klosowski & Silva, 2001), which introduces a conservative occlusion culling algorithm, cPLP.

A *conservative* algorithm produces an image that looks just like the image rendered with all polygons (it contains no errors). This kind of algorithm can overestimate the output set, but never underestimate it. Examples of these algorithms are HZB (see chapter 2.1) and cPLP.

An *approximate* algorithm produces an image that is mostly correct; it makes a tradeoff between correctness and speed. On some applications, fast rendering is more important than correct output. The Prioritized-Layered Projection (PLP) algorithm is one such example (Klosowski & Silva, 2000). It is an aggressive culling algorithm that renders only those polygons that are most likely to be visible from the given viewpoint. The algorithm works on a user-given budget, and stops when the budget has been reached.

Just like HZB and BSP, PLP divides the object-space into nodes, or cells. These cells need not to be in a hierarchical structure, so the structure might be a simple spatial partitioning, or it might be an octree that contains only leaf nodes. The division of the object-space is done at preprocessing stage, and the density of primitives in each cell is kept roughly uniform.

When rendering a frame, PLP starts from the initial cell that contains the current viewpoint. It constructs a collection $\mathcal{F}$, called *front*, that contains the cells that are least likely to be occluded. Initially $\mathcal{F}$ is empty. Each cell in $\mathcal{F}$ contains a solidicity value ρ, which tells how unlikely it is to be occluded. When algorithm starts, it takes the initial cell and inserts it into $\mathcal{F}$. Then the following steps are repeated:

(1) Take the cell in $\mathcal{F}$ that is least likely to be occluded (has largest ρ value), schedule it to be rendered. (2) Take its neighbouring cells, update their solidicity values ρ based upon the current viewpoint, view direction and normal of the cells. Insert these cells into $\mathcal{F}$. If $\mathcal{F}$ already contains one of these cells, ρ is accumulated to its previous value. (3) Repeat steps until the user-given budget runs out (or all primitives are rendered).

cPLP is a conservative culling algorithm that is built on top of PLP. cPLP first runs an initial, approximate run of PLP, and then fills in the holes, i.e. renders the visible objects that PLP left out. When the initial PLP has been run, two data structures have been created as a side effect: Z-buffer corresponding to the approximate image, and the front $\mathcal{F}$ that contains the cells that would have been rendered if there had been any budget left. cPLP can be summed as follows:

(1) Run PLP using a small budget of primitives. (2) Using the resulting front $\mathcal{F}$, determine which cells in $\mathcal{F}$ are occluded by doing visibility queries with resulting Z-buffer. Remove the occluded cells from $\mathcal{F}$. (3) Continue running PLP until $\mathcal{F}$ is empty (all cells in $\mathcal{F}$ are now known to be visible). During this step, new neighbouring cells are inserted into $\mathcal{F}$ but are not processed as candidates for rendering during this step. (4) If $\mathcal{F}$ is now empty, there are no more potentially visible primitives and we can quit. Otherwise, go back to step 2.

Clearly the most challenging part of this algorithm is doing the visibility queries. In (Klosowski & Silva, 2001), three different techniques are presented to perform these queries.

The first technique relies on a special HP graphic accelerator that is able to check if changes would have been made to the Z-buffer when scan-converting the primitives. (This same occlusion query is nowadays provided by some other manufacturers too, like NVidia and ATI.) This offers a very simple solution to finding the visible cells in $\mathcal{F}$: given the current Z-buffer, check whether the bounding volume of each cell is visible or not. However, this test is not cheap, so the naĄïve approach of checking each and every polygon with the hardware test is slower than using cPLP (0.157 seconds per frame compared to 0.06 seconds per frame).

The second technique, called item-buffer technique, doesn't rely on any special hardware but uses the color buffer to determine visibility. The idea is to store the contents of the color buffer and clear it, disable changes to the Z-buffer, and then render the queried cells with known colors. If that color appears during the scan of color buffer, we know the cell is visible. Since the cost of transferring the color buffer takes time, it is preferable to query many cells at once. This presents new challenges: some cells might be occluded by other cells in the front. This requires the use of multipass algorithm.

The third technique uses an OpenGL 1.2 histogram extension. It is almost like item-buffering described above, but instead of transferring the contents of the color buffer to CPU and scanning for the visible cells, it is transferred to texture memory (on the same accelerator). During this transfer, the number of color occurrences is counted. This gives the same effect as scanning the color buffer.

Experimental results show that cPLP can run complex, densely occluded models real-time with all these different implementations. Naturally the hardware implementation is the fastest one.

3 HIERARCHICAL DYNAMIC ALGORITHMS

All algorithms in the previous chapter had one thing in common: they required a pre-processing stage to build up a data structure of the scene. Clearly these algorithms cannot handle the moving of the objects, since this would require the expensive preprocessing be done again.

Sudarsky and Gotsman present in (Sudarsky & Gotsman, 1999) how these algorithms can be modified to handle moving objects. First of all, since building the data structure takes so long, it must be merely updated when objects are moving. But if this update is done for every object movement, including the hidden ones, the algorithm will not be output-sensitive. This means that the objects that are not visible should be ignored in the update. But they cannot be forgotten completely either, because they might become visible at some moment.

The solution to this problem lies in the *lazy evaluation* of the hidden object movements. This is achieved by using a special structure called Temporary Bounding Volume, TBV. A TBV for a dynamic object is guaranteed to contain that object during some period of time. This period of time is called *validity period* of the TBV, and the last moment of that period is called *expiration date*. To construct a TBV for an object, we need some prior knowledge of its movements. Fortunately, most objects have some constraints that can be used. If only maximum velocity or maximum acceleration is known, a TBV can be constructed upon it. Note that TBVs are constructed on-the-fly, so they can be used in applications where the flow of events is unknown beforehand.

TBVs are used in conjuction with the hierarchical structures that represent the moving objects. A moving object that is hidden is ignored until either of the events happen: (1) its TBV expires, which means that the TBV no longer is guaranteed to contain the object or (2) visibility algorithm finds out that the TBV that represents the object is visible again, thus implying that the object might be visible too. In either case, the object must be considered again.

How to set the validity periods for TBVs? In general situations, it is best to use an adaptive algorithm to choose the validity periods. If a TBV expires, it means that the period was too short, and the next TBV will be assigned a longer period. If a TBV becomes visible, it means that the TBV was too big and/or the validity period was too long. Thus the next TBV will be assigned a shorter period.

How to apply the use of TBVs into hierarchical algorithms like HZB (chapter 2.1) or BSP (chapter 2.2)? The key idea is to attach a TBV to each hidden moving object. We need a priority queue of these TBVs so we can find the expiring ones. In addition, the algorithm must be modified so that it handles TBVs and objects instead of objects. To handle the Potentially Visible Set (PSV), we need a list of all currently visible moving objects. To update this list of visible objects, each object is associated with a time stamp that tells when it was last seen. And of course we need the data structure (an octree or BSP tree) that represents the model. When these data structures are constructed, the general TBV algorithm proceeds as in figure 4.

Chapter 2.1 presented the hierarchical Z-buffer algorithm. The modification of it into a dynamic one is presented in (Sudarsky & Gotsman, 1999). To update HZB to handle moving objects like in the general algorithm (see figure 4), the octree needs to be modified at runtime. When inserting an object or a TBV into the octree, the primitives of the object are associated with the smallest enclosing cubes as usual. But this presents some new problems. If some nodes contain too many primitives, they must be split further. Also deletion might cause changes on the octree structure: to delete an object or a TBV from the octree, the primitives are removed and sibling

Needed datastructures:

- Priority queue for TBVs
- List of moving visible objects
- Time stamps for all objects
- Data structure for the model

Following steps are executed for each frame:

1. If there are any expiring TBVs in the TBV priority queue, they are removed from the queue and their objects are added to the list of moving visible objects.
2. All objects in the list of moving visible objects are calculated for their new positions, and these new positions are updated in the data structure that represents the model.
3. Run the visibility algorithm on the data structure as usual. If one encounters a node that contains a TBV, these TBVs are removed from the TBV priority queue and their objects are added to the list of visible objects. Also their new positions are calculated. For each visible object in the node, update its time stamp to the current time.
4. When the algorithm ends, the list of visible objects is updated so that each object that has an old time stamp is removed from the list. Since these objects are now becoming invisible, they are associated with a new TBV that is inserted into the TBV priority queue.

Figure 4: General hierarchical dynamic algorithm with TBVs.

nodes with too few primitives (or with the same primitives) are merged.

When objects are moving, they are updated accordingly. An update can contain a deletion from the tree followed by an insertion to a new position. The resulting octree is equivalent to the one that had been built from scratch. However, it is not optimal, as figure 5(a) shows.

To improve the situation, a technique called Least Common Ancestor (LCA) could be used. Instead of updating the whole octree for deletions and creations, only the least common ancestor is updated (see figure 5(b)). This method is improvement especially in relatively big, deep models, since the LCA is expected to be near the leaves.

The modification of the general TBV algorithm (figure 4) to handle HZB is now straightforward. We need the same auxiliary data structures (timestamps for objects, priority queue for expiring TBVs and list of visible objects). When adding or removing objects or TBVs in the octree, we use the technique of LCA. Step 3 is like the usual HZB traversal, except that one might encounter also TBVs that have now become visible. Other TBVs (hidden objects) remain untouched in the algorithm traversal.

Sudarsky and Gotsman also present (see Sudarsky & Gotsman, 1997) a dynamic

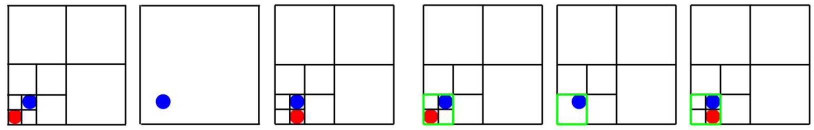

Figure 5: (a) When the red object moves into different location, the nodes are merged needlessly. (b) Using LCA eliminates unnecessary tree updates.

modification for BSP projection algorithm (see chapter 2.2). The particular algorithm was chosen because it has “several advantages over hierarchical Z-buffering” and “it is sufficiently different from hierarchical Z-buffering to highlight the essence of our approach” (Sudarsky & Gotsman, 1999). In addition to the data structures in the general TBV algorithm, dynamic BSP algorithm needs an auxiliary BSP tree that is used in the union process in the algorithm. This is used when adding or removing new TBVs or objects in the model data structure. The algorithm does roughly the same as in figure 4, the only difference is that it uses the small BSP tree to manipulate the model BSP tree.

This same approach of updating hierarchical structures can be applied to many hierarchical algorithms, for example HOM. But the modification must be done with care, because the update of complex structures is error-prone. In addition, the updates might even be quite expensive, resulting in a frame rate less than expected. This is why some have abandoned the use of hierarchical structures altogether and are approaching the problem from other directions.

4 OTHER DYNAMIC ALGORITHMS

Chapter 3 discussed how to convert an algorithm that contains hierarchical data structures into dynamic. There are, however, some other ways to approach the problem. Chenney and Forsyth (Chenney & Forsyth, 1997) present an output-sensitive way of statistically estimating the state of a moving system. They deduce the properties of the moving objects and derive how to calculate the state of the object when it becomes visible again.

4.1 Regular grid

Another approach, presented in (Batagelo & Wu, 2002), is a more traditional culling algorithm. The thing that makes it unusual is that is doesn’t use any hierarchical data structure to represent the model. Instead, it divides the model into regular voxels and keeps four matrices that represent different attributes of these voxels.

Occluder matrix $\mathcal{O}$ classifies each voxel as opaque or non-opaque. A voxel is opaque if it is totally inside in a potentially visible object. **Occlusion matrix** $\mathcal{H}$ classifies each voxel as occluded or non-occluded. A voxel is fully occluded if it is hidden behind opaque or occluded voxels. **Identifiers matrix** $\mathcal{I}$ associates a list of object IDs to each voxel. These objects span the spatial region of that voxel. **TBVs matrix** $\mathcal{T}$ associates a list of TBV IDs to each voxel. These TBVs span the spatial region of that voxel.

The algorithm itself contains four steps: (1) Scene discretization (2) View-frustrum traversal (3) Occluder extension (4) Occlusion computation.

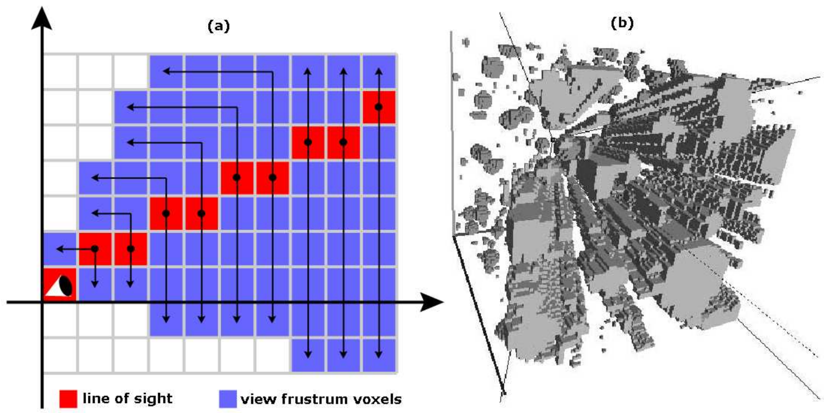

Figure 6: (a) View-frustrum traversal. (b) A visualisation of a scene and its hidden and opaque voxels.

Scene discretization starts by taking the Potentially Visible Set (PVS) of the last frame and discretizing these objects in occluders matrix $\mathcal{O}$ and identifiers matrix $\mathcal{I}$. When determining the opaque voxels in $\mathcal{O}$, the actual objects are underestimated (for example estimating a 3D object with a sphere that is fully contained in it). When determining the object IDs that span voxels, the objects are overestimated (for example estimating a 3D object with a sphere that fully contains the object). This *occlusion-preserving* principle ensures the conservatism in the algorithm. Also TBVs are updated in TBVs matrix $\mathcal{T}$ using the same principle as Sudarsky and Gostman: only nonvisible objects can have TBVs. If an object is not in the current PVS and has no TBV, it was potentially visible in the previous frame and is now becoming invisible. Thus it needs a new TBV, which is updated in $\mathcal{T}$. If an object is not in the current Potentially Visible Set but has a TBV, it has been and is continuing to be invisible. One has to only check the expiration and update if needed.

View frustrum traversal starts along the view vector that is dicretized using Bresenham algorithm. (See figure 6.) The traversal is done by sweeping subsuquent layers of voxels that have the same chess distance (chess distance between (x_1, y_1, z_1) and (x_2, y_2, z_2) is given by $max(|x_2 - x_1|, |y_2 - y_1|, |z_2 - z_1|)$). During the traversal, if a non-occluded voxel is reached, all objects in identifiers matrix $\mathcal{I}$ for that voxel are added to PVS of the current frame, since they are now visible. If this voxel is associated with any TBVs, it means that there are some TBVs that are becoming visible. Therefore these TBVs are removed from the TBVs matrix $\mathcal{T}$ and the corrensponding objects are discretized, added to identifiers matrix $\mathcal{I}$ and occluders matrix $\mathcal{O}$ and treated like other visible objects. The computation of Potentially Visible Set is done when the traversal finishes. It can also be terminated earlier if the algorithm detects only occluded voxels in all directions.

During the traversal, opaque non-occluded voxels are used to deduce the occluded voxels behind them. This step, called occluder extension phase, tries to maximally aggregate the opaque and occluded voxels. This set of aggregated voxels is called an *extended* or *virtual occluder*, since these voxels necessary don't contain an opaque or occluded voxel (they are not part of the scene model at all).

The last step, occlusion computation, computes the actual occlusion volume based on the extended occluders and the viewpoint. (See figure 6.) The occluded voxels can be calculated by rasterizing the occlusion volume into the data structure. The voxels that are rasterized are condidered occluded. To preserve conservatism, we should guarantee that only voxels totally inside the occlusion volume are classified as occluded. This can be achieved by shrinking the occlusion volume into a rasterized occlusion polygon with a simple algorithm.

Regular grids obtain interactive speeds even when there are hundreds of moving objects. Thus it is truly dynamic, output-sensitive, interactive algorithm that requires no preprocessing at all. But there are some weaknesses in the algorithm. First of all, regular grids handle dense and sparse areas with the same subdivision. This means that it can't cull large parts of the model in a high level of hierarchy. In addition, it is a bit too conservative, and requires the model to consist of closed objects.

But these drawbacks are quite minor compared to the good aspects of this algorithm.

5 CONCLUSIONS

As we saw, there are many ways to achieve dynamic occlusion culling algorithms. Dynamic might mean that the culling is done on-the-fly, or it might mean that the objects in the model can move. The previous case is much more common, and a multitude of algorithms have been developed for this purpose. Most of them rely on a hierarchical data structure, like octree or BSP tree. These hierarchical structures are used to achieve output sensitivity: the runtime of the culling algorithm doesn't depend on input but on output. Hierarchical structures enable culling large portions of the model high on hierarchy. Example of these algorithms are HZB (Greene *et al.* , 1993) and cPLP (Klosowski & Silva, 2001).

But these hierarchical structures run into problems when objects start to move. Since these structures are constructed at a time-consuming preprocessing stage, they cannot be built from the scratch when objects in the model move. Instead, the data structures must be updated. This updating is quite a complicated task. Sudarsky and Gotsman solved the problem using temporary bounding volumes (Sudarsky & Gotsman, 1999). But there are some solutions that solve the problem of hierarchical structures by leaving them out altogether.

In the future, it would be interesting to see how algorithms with moving objects are developed. Will the hierarchical structures be dominant, or will we see more non-hierarchical solutions?

REFERENCES

Batagelo Harlen Costa, & Wu Shin-Ting. 2002. Dynamic scene occlusion culling using a regular grid. *Pages 43–50 of: Proceedings of the XV Brazilian Symposium on Computer Graphics and Image Processing*.

Chenney Stephen, & Forsyth David. 1997. View-dependent culling of dynamic systems in virtual environments. *Pages 55–58 of: Proceedings of the 1997 symposium on Interactive 3D graphics*.

Coorg Satyan, & Teller Seth. 1997. Real-time occlusion culling for models with large occluders. *Pages 83–90 of: Proceedings of the 1997 symposium on Interactive 3D graphics*.

Greene Ned, Kass Michael, & Miller Gavin. 1993. Hierarchical Z-buffer Visibility. *Pages 231–238 of: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*.

Klosowski J.T., & Silva C.T. 2000. The prioritized-layered projection algorithm for visible set estimation. *IEEE Transactions on Visualization and Computer Graphics*, **6**(2), 108–123.

Klosowski J.T., & Silva C.T. 2001. Efficient conservative visibility culling using the prioritized-layered projection algorithm. *IEEE Transactions on Visualization and Computer Graphics*, **7**(4), 365–379.

Naylor B.F. 1992. Partitioning tree image representation and generation from 3D geometric models. *Pages 201–212 of: Graphics Interface, Proceedings 1992*.

Sudarsky Oded, & Gotsman Craig. 1996. Output-sensitive visibility algorithms for dynamic scenes with applications to virtual reality. *Computer Graphics Forum*, **15**(3), 249–258.

Sudarsky Oded, & Gotsman Craig. 1997. Output-sensititive rendering and communication in dynamic virtual environments. *Pages 217–233 of: Proceedings of the ACM symposium on Virtual reality software and technology*.

Sudarsky Oded, & Gotsman Craig. 1999. Dynamic scene occlusion culling. *IEEE Transactions on Visualization and Computer Graphics*, **5**(1), 13–29.

Zhang Hansong, Manocha Dinesh, Hudson Tom, & Hoff Kenneth E. III. 1997. Visibility culling using hierarchical occlusion maps. *Pages 77–88 of: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*.