Volume xx (*200y*), *Number z, pp. 1–10*

Dynamic Scene Visibility Culling using a Regular Grid

Harlen Costa Batagelo and Wu, Shin-Ting

Department of Computer Engineering and Industrial Automation
School of Electrical and Computer Engineering
State University of Campinas, Campinas, São Paulo, Brazil

Abstract

We present an occlusion culling algorithm for large and complex scenes where both the viewpoint and objects can move arbitrarily on-the-fly. Our method uses a regular grid as spatial subdivision of the scene instead of the hierarchical data structures commonly used for static scenes. Our hypothesis is that this approach can lead to a faster evaluation of dynamic occluders because the overhead of maintaining the grid data structure is smaller. We also introduce new techniques of traversal of voxels, object discretization and occlusion computation that strengthen the benefits of using regular grids in dynamic scenes. We discuss the results of an implementation.
Note: *This is the revised version of a paper presented at SIBGRAPI 2002, Fortaleza, Brazil, Oct 2002. The proceedings were published by IEEE Computer Press.*

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Hidden line/surface removal

1. Introduction

The efficient visualization of large scenes composed by several millions of polygons is one of the most challenging problems in today's computer graphics. Usually, these scenes are *densely occluded*, which means that the fraction of visible geometry with respect to any viewpoint is only a fraction of the whole model[2]. The exhibition of densely occluded scenes can be accelerated by *occlusion culling* algorithms - a class of visibility algorithms that quickly detect trivially occluded portions of the scene and avoid sending them to the rendering pipeline.

The efficient visibility determination of scenes composed by objects of arbitrary motion - the so-called dynamic scenes - is still an under-explored area in computer graphics[3]. In general, occlusion culling algorithms rely on expensive preprocessing stages in order to build hierarchical data structures to accelerate the discard of trivially occluded geometry in runtime, as large parts of the scene can be early classified as hidden in high levels of the hierarchy. Nevertheless, the handling of changes of hierarchical relations for multiple dynamic objects may be prohibitively slow to be done on-the-fly.

Although considerable research effort has been devoted to the acceleration of updates in hierarchical spatial databases (see section 2), in this work we propose to use a simple (but fast) regular grid that combines efficient procedures of grid traversal and occlusion computation for fast evaluation of dynamic occluders. The visibility algorithm proposed to work with this data structure is based on previous works of occlusion culling, mainly on the approaches proposed by Schaufler *et al.*[12] and Sudarsky and Gotsman[14]. Our algorithm inherits most benefits from these techniques, such as the ability to *fuse* occluders in object-space and the reduction in output-sensitive complexity, *i.e.*, the execution time of the visibility determination is proportional only to the number of visible objects (see Sudarsky[14] for details). However, our algorithm is completely *online*; it does not depend on intensive preprocessing stages, pre-selection of occluders or precomputation of PVSs (Potentially Visible Sets).

We hope that this work will motivate a discussion on applying regular grids in dynamic scenes. While the benefits of most hierarchical approaches do not seem to overcome the cost of handling a large number of dynamic objects, regular grids have the drawback of handling dense and sparse areas of the scene with the same subdivision, thus being unable to cull out large portions of the scene in high levels of the hierarchy. We have focused on these issues aiming

at presenting contributions for minimizing such problems and therefore encouraging the use of regular grids with occlusion culling algorithms. In particular, we exploit a natural correspondence between regular voxels and pixels of the frame buffer to: (1) develop a fast procedure of front-to-back traversal of regular voxels enclosed by the view-frustum; (2) classify occluded regions of the space by efficiently "rasterizing" *occlusion volumes* in the regular grid; (3) optimize the discretization of *temporal bounding volumes* to reduce the overhead due to handling of hidden dynamic objects[14].

The rest of the paper is organized as follows. In the next section, we discuss the occlusion culling problem and review related work on dynamic scene occlusion culling. In section 3, the basic data structures and the principal steps of our algorithm are given. Next, we detail our algorithm for 3D scenes (section 4). We discuss the implementation results in section 5 and conclude with suggestions for future work in section 6.

2. Previous Work

For a comprehensive survey about visibility we suggest Durand's thesis[4]. Visibility culling is specially covered by Cohen-Or *et al.*[3], Möller and Haines[9], and also by Aila and Miettinen[1].

A relatively few of the occlusion culling algorithms in the literature are devoted to dynamic environments. Although many visibility techniques can answer whether a dynamic object is being occluded by some portion of the scene, they consider as occluders only static objects and cannot answer whether a dynamic object occludes some part of the scene[5, 12]. Nevertheless, in dynamic scenes with objects in arbitrary motion, any object can be a potential occluder, for instance, moving right in front of the viewpoint and blocking its field of view, or simply growing in size.

While we can find efficient dynamic scene occlusion culling algorithms for indoor architectural scenes[8] and 2.5D urban scenes[15], 3D cases remain almost unexplored (a remarkable exception is the *dPVS* API[1]). Besides the technique of *temporal bounding volumes*[14], we considered methods that can be further adapted to dynamic scenes, such as the *hierarchical z-buffer*[6, 7] and the *hierarchical occlusion map*[17].

The hierarchical z-buffer (HZB) uses a pyramid of z-buffers and an octree to remove large parts of the scene with few comparisons. The levels of the pyramid are built by an iterative process that attributes the farthest z-value of 2x2 arrays of pixels of the current level to a single pixel of the subsequent level, beginning at the base of the pyramid that is a standard z-buffer. In runtime, the octree is traversed in front-to-back, top-down order, and each node is compared with the pyramid of z-values. If a node is completely occluded, then its sub-nodes and objects contained in its interior are discarded. Unfortunately, HZB needs to read back the contents of the z-buffer – an operation hardly supported by graphics hardware (a recent exception is the nVidia's *z occlusion query* feature available in GeForce3 and subsequent chipsets[11]). Recently, Greene has proposed changes of the original HZB for feasible hardware implementations[6]. In his new approach, the bandwidth traffic of z-values can be greatly reduced and in some cases it becomes more efficient than using a standard z-buffer with the visible geometry known in advance.

An alternative to HZB that does not depend on special graphics hardware is the technique of hierarchical occlusion maps (HOM), which decomposes the visibility test in a coverage and a depth test. The hierarchical occlusion map is similar to the HZB pyramid, differing in containing opacity values instead of depth values in each of the level maps. For each frame, a HOM is built for a large group of occluders selected from an occluder database. The scene geometry, previously organized as a hierarchy of bounding boxes, is tested for coverage against the pyramid. The depth test, implemented as a low-resolution z-buffer in software, is then performed only for the geometry that covers (both fully and partially) the discretized occluders in the HOM. An object is occluded if its projected bounding box covers only opaque pixels in the HOM and is behind the occluders according to the depth test.

For dynamic scenes, the hierarchical data structures used by HZB and HOM are replaced by oriented bounding boxes. In the HOM technique, the precomputation of an occluder database is circumvented. Instead, occluders are chosen in runtime according to the size and distance from the viewer. The cost to select a good set of occluders in runtime is reduced by using frame coherence. However, while these methods work in dynamic scenes more efficiently than a traditional z-buffer (see *e.g.*, ATI's *Hyper-Z technology*[10] and nVidia's *z occlusion culling*[11]), the complexity of the visibility determination is still at least linear in the number of input objects. All objects must be tested against the pyramid, even those that do not contribute any pixel to the final image.

The main problem that arises in handling dynamic scenes is the difficulty in efficiently updating the hierarchical data structures that most visibility algorithms use (usually octrees or kD-trees). In addition, if the data structure is updated for each frame and for all dynamic objects, the output-sensitivity is lost.

Many works have been conducted to adapt octrees for dynamic scenes. Smith *et al.* present an algorithm for maintaining the octree for objects subjected to rigid-body transformations[13]. Sudarsky and Gotsman use temporal coherence to restrict the change of the octree to the smallest voxel that encloses both the previous and current positions of the modified object (called the *least common ancestor* voxel), thus reducing the overhead due to the update of dynamic objects[14].

In order to reduce the number of updates of the data struc-

ture, it is possible to associate to each dynamic object a region of space that completely encloses the object during a sequence of animation. These bounding volumes can be inserted in the spatial database such that the corresponding dynamic object can be ignored until the visibility culling algorithm classifies those bounding volumes as potentially visible. For dynamic objects of arbitrary motion, Sudarsky and Gotsman calculate bounding volumes for short periods of time, called temporal bounding volumes (TBVs)[14]. For instance, if the maximum velocity of each dynamic object is known, then given the position of an object in a certain moment, it is possible to compute a bounding sphere that surely contains this object until a future time. This future time is called the TBV's expiration date and determines the validity period of the bounding volume. A hidden dynamic object only needs to be considered if its bounding volume becomes visible or the expiration date is reached. The output-sensitivity with respect to the number of dynamic objects is achieved because the spatial database is updated only when the objects really become potentially visible. Applications currently using TBVs to handle dynamic objects include the *dPVS* API[1], a commercial visibility culling library. The dPVS organizes the scene geometry into a kD-tree that, according to the authors, allows faster updates than octrees (the first data structure exploited by Sudarsky). The visibility culling algorithm is based on several optimizations of the HOM and other techniques, which results in an efficient culling optimization for a broad class of scenes.

3. Overview

The regular grid represents a discretization of the space where each voxel identifies local features of the scene such as opacity, occlusion and spanned objects. At each frame, all voxels that intersect the view-frustum are traversed in a front-to-back order from the viewer, searching for opaque voxels that can be used as occluders. According to the approach introduced by Schaufler *et al.*[12], each occluder can be extended by the aggregation of opaque and occluded voxels in the neighborhood of the initial opaque voxel, thus realizing *occluder fusion* – the combination of sets of small and disjoint occluders to build larger and more effective ones. Only objects fully contained in occluded voxels are considered hidden. Therefore, the set of objects sent to the rendering pipeline is always an overestimate of the visible objects.

For optimization purposes, we have organized the information of the regular grid into four matrices.

- *Occluder matrix* ($\mathcal{O}$), which classifies each voxel as *opaque* or *non-opaque*. A voxel is opaque if it is totally contained in a potentially visible object.
- *Occlusion matrix* ($\mathcal{H}$), that classifies each voxel as *occluded* or *non-occluded*. A voxel is occluded if it is fully hidden by opaque or occluded voxels with respect to the viewpoint.
- *Identifiers matrix* ($\mathcal{I}$), which associates to each voxel a list of identifiers (IDs) of potentially visible objects that span its spatial region in the scene.
- *TBVs matrix* ($\mathcal{T}$), that associates to each voxel a list of IDs of TBVs that span its spatial region in the scene.

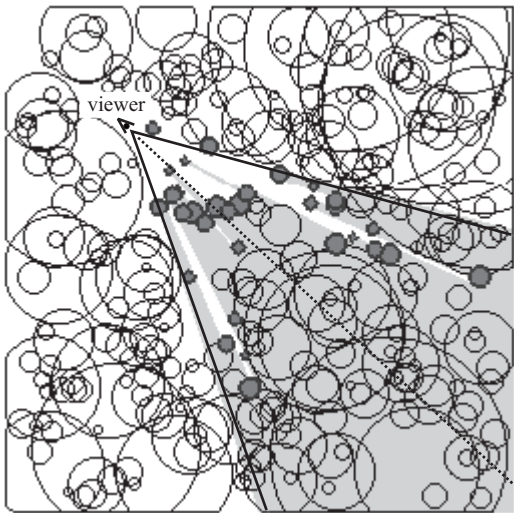

Figure 1: *Visualization of the data in a 2D regular grid.*

Without loss of generality, let us illustrate these matrices through a 2D scene with 300 dynamic circles (32 potentially visible) which were discretized into a 256x256 regular grid. Figure 1 shows an overlaid view of those four matrices where each point illustrates a voxel. For the sake of clarity, we also included the point of the viewer, the limits of the field of view (solid lines) and the view direction (dashed line). Opaque voxels are shown in dark gray and occluded voxels are drawn in light gray. Note that opaque voxels are contained in the dark gray circles, which are the potentially visible objects. The non-empty $\mathcal{I}$-voxels are also depicted as dark gray circles, since they are coincident with the opaque voxels. In addition, they include the boundary voxels of these circles (in dark). Finally, the non-empty $\mathcal{T}$-voxels are shown in black. Each TBV has the shape of an empty circle.

We assume that each object has three main attributes: an identifier, a maximum velocity and a flag that indicates whether a TBV is associated with it. When this flag is true, the object should also provide a TBV expiration date, a TBV position and a TBV diameter. IDs of TBVs may have the same value of the IDs of the objects the TBVs belong to.

The dynamic scene occlusion culling algorithm comprises the following main steps, which are executed for each frame:

- **Scene discretization:** Update the regular grid for objects reported in the PVS of the last frame and handle the hidden objects according to their TBVs.
- **View-frustum traversal:** Traverse the voxels of the view-

frustum in a front-to-back order to detect potentially visible objects as well as opaque voxels that can be used as occluders.
- **Occluder extension:** Extend each occluder found during the view-frustum traversal to the adjacent opaque and occluded voxels.
- **Occlusion computation:** Compute an occlusion volume for each extended occluder and determine the occluded voxels.

4. Algorithm

The main bottleneck of a visibility culling algorithm using regular grids is the high number of voxels that need to be accessed and updated each frame, since hierarchies are not available to ignore large subsets of voxels during the view-frustum traversal. Therefore, when converting the main steps of the algorithm after Schaufler's work[12] and after Sudarsky's technique[14] to the regular grid approach, we tried to accelerate the accesses to voxels using coherence between consecutive voxels.

4.1. Scene discretization

All objects classified as potentially visible by the PVS of the last frame are discretized in $\mathcal{O}$ and updated in $\mathcal{I}$. The remaining objects update $\mathcal{T}$. In the very first frame, all objects are handled as if they were potentially visible, once they do not have TBVs associated and we cannot tell which objects are hidden.

The discretization of potentially visible objects is performed in object-space by a sequence of intersection tests between voxels and simplified geometries. The simplified models are pre-computed bounding spheres or boxes that represent the original objects and are used for determining the opaque and spanned regions of the space according to an *occlusion-preserving* principle: for the classification of opaque voxels, the simplified model should be entirely inside the original geometry and for the classification of spanned voxels it should fully contains the original geometry. From our experiments, this strategy seems to be efficient. Figure 2 illustrates the use of cubes as simplified geometries.

The handling of temporal bounding volumes is based on the procedure proposed by Sudarsky and Gotsman[14]. It is only applied to hidden dynamic objects, according to the following criteria: (1) Objects not contained in the current PVS, without TBVs, were potentially visible objects in the previous frame and are becoming hidden in the current one. In this case, new TBVs are allocated to them and $\mathcal{T}$ is updated accordingly. (2) Objects not contained in the current PVS, but with TBVs, were hidden in the previous and are hidden in the current frame. In this case, if TBV validity period is expired, a new validity period is attributed.

For dynamic objects of arbitrary motion, TBVs can be tightly approximated by spheres. For discretization, the voxels that span the spherical TBV should be updated to include the TBV's identifier. This procedure involves updating a large number of voxels of $\mathcal{T}$, since the number of TBVs in a typical scene is much larger than the average number of potentially visible objects. In addition, the discretization of TBVs may produce uneven frame rates when multiple TBVs need to be updated in the same frame.

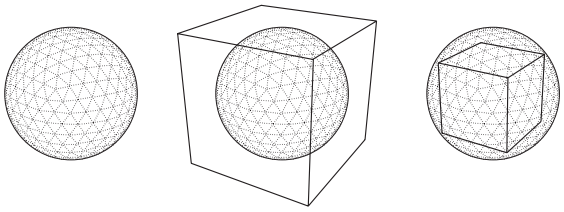

Figure 2: *Simplified models for discretization of 3D objects. Left: original geometry. Center: simplified geometry for classification of spanned voxels. Right: simplified geometry for classification of opaque voxels.*

We noted that for scenes where the viewer moves smoothly in the space, it is sufficient to update only the voxels that span the surface of the sphere, thus greatly reducing the number of $\mathcal{T}$-voxels changed. The TBVs are still correctly detected, provided the trajectory of the viewer is always 8-connected in the regular grid and the validity period of the TBVs are chosen in such a way that the radii of the correponding spheres do not enclose the viewpoint. Moreover, we can achieve further reductions of accesses of $\mathcal{T}$-voxels by representing 3D TBVs by only three 2D TBVs matrices. TBVs are discretized by orthographically projecting their geometry onto the coordinate planes XY, XZ and YZ, then using three 2D TBVs matrices ($\mathcal{T}_{xy}$, $\mathcal{T}_{xz}$ and $\mathcal{T}_{yz}$) coincident with these planes to store the IDs of the projected TBVs. During the view-frustum traversal, the voxels that span the TBVs are determined by testing whether the projections of the current (x,y,z) voxel onto the coordinate planes have the same TBV ID in $\mathcal{T}_{xy}$, $\mathcal{T}_{xz}$ and $\mathcal{T}_{yz}$. An illustration of this procedure is shown in Figure 3. Unfortunately, the TBV rebuilt from the projections of a sphere is not a (digitized) sphere, but a solid aggregate of voxels that comprehends the intersection of three digitized cylinders and includes the redundant interior voxels of the TBV. Further reductions in the number of $\mathcal{T}$-voxels can be obtained by representing TBVs as cube's faces instead of spheres. The cubical TBVs are discretized by projecting only the edges of the cube's faces onto the coordinate planes. During the view-frustum traversal, we can now determine which voxels span only the surface of the TBV by checking whether a projected (x,y,z) voxel has the same TBV ID in $\mathcal{T}_{xy}$, $\mathcal{T}_{xz}$ and $\mathcal{T}_{yz}$ and if, for a given ID, this existence test satisfies $(\mathcal{T}_{xy} \vee \mathcal{T}_{yz}) \wedge (\mathcal{T}_{xy} \vee \mathcal{T}_{xz}) \wedge (\mathcal{T}_{yz} \vee \mathcal{T}_{xz})$ for respective coordinates of (x,y), (x,z) and (y,z) in $\mathcal{T}_{xy}$, $\mathcal{T}_{xz}$ and $\mathcal{T}_{yz}$. Figure 4 illustrates this approach. This last op-

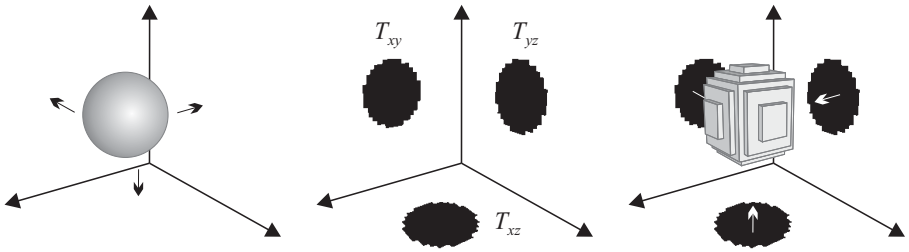

Figure 3: *Optimized discretization of TBVs. Left: original TBV. Center: projection of the TBV onto three 2D TBVs matrices. Right: TBV rebuilt from the projections.*

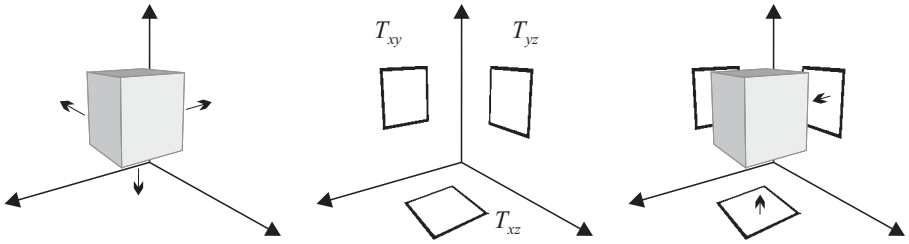

Figure 4: *Optimized discretization of cubical TBVs. Left: original TBV. Center: projection of the edges of the cube's faces onto three 2D TBVs matrices. Right: surface of the TBV rebuilt from the projections.*

timization is remarkable, as we decrease the number of updates in voxels of $\mathcal{T}$ from $O(n^3)$ to only $O(n)$ for cubical TBVs.

4.2. View-Frustum Traversal

The visibility determination is actually done in the view-frustum traversal. It comprehends the traversal of voxels that span the view-frustum in order to identify occluders and potentially visible objects. Both are found as non-occluded voxels of $\mathcal{H}$: the former as opaque voxels of $\mathcal{O}$; the latter as voxels containing non-empty ID lists of $\mathcal{I}$ or $\mathcal{T}$.

The traversal of view-frustum voxels is performed in a front-to-back order from the viewer, so the algorithm does not waste time on handling hidden occluders and can determine the PVS incrementally in a single traversal.

In order to efficiently compute the distance from the viewpoint to each voxel and hence perform a front-to-back traversal, we propose to use the chessboard metric[†]. Besides avoiding expensive square root operations demanding by the Euclidian metric, the chessboard metric induces a fast traversal of regular voxels in axis-aligned directions. Since the line-of-sight is always inside the view-frustum, it is possible to discretize it incrementally from the viewpoint using a 3D line-drawing algorithm and, from each voxel that contains the discretized line-of-sight (called *seed-voxel*), traverse the plane of voxels that have the same chessboard distance to the voxel containing the viewpoint. For instance, let (x,y,z) be the position of a seed-voxel given in coordinates relative to the voxel containing the viewpoint, the initial planes of traversal are: (1) *zy* if $(|x| > |y|) \wedge (|x| > |z|)$; (2) *zx* if $(|y| > |x|) \wedge (|y| > |z|)$; (3) *xy* if $(|z| > |x|) \wedge (|z| > |y|)$; (4) *zy* and *xy* if $|x| = |z|$; (5) *zx* and *yx* if $|y| = |z|$; (6) *yz* and *xz* if $|y| = |x|$. Figure 5 illustrates a chessboard metric traversal on a xy-plane. Observe that the traversal direction (arrows in Figure 5) only changes when a voxel with $|x| = |y|$ is reached and proceeds in a direction perpendicular to the previous one, until a voxel completely outside the view-frustum or a seed-voxel (in the case of a 360° FOV) is reached. It is easy to infer that the directions of traversal are changed only when $|x| = |y|$, $|x| = |z|$ or $|y| = |z|$.

During the traversal, if a non-occluded voxel is reached, all objects contained in its ID list of $\mathcal{I}$ are added to the PVS of the current frame. Opaque non-occluded voxels are considered as occluders, therefore should be used to determine which voxels are being hidden with respect to the viewpoint (this step consists of the processes of occluder extension and occlusion computation, detailed in the next sections). Non-occluded voxels that contain TBVs indicate that these TBVs were revealed and that the corresponding objects may be visible. Therefore, these TBVs are removed from $\mathcal{T}$ and dissociated from the respective objects. Moreover, such objects are immediately discretized in $\mathcal{I}$ and $\mathcal{O}$, so the algo-

[†] In the chessboard metric, the distance between two points (x_1,y_1,z_1) and (x_2,y_2,z_2) is given by $max(|x_1-x_2|,|y_1-y_2|,|z_1-z_2|)$.

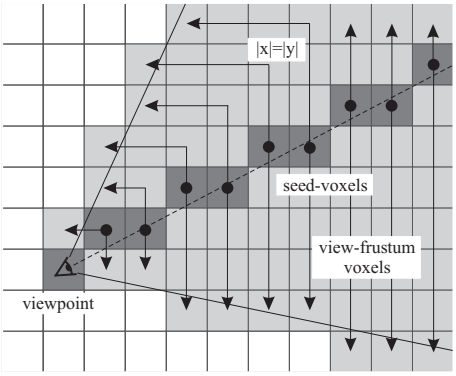

Figure 5: *View-frustum traversal in a xy-plane.*

rithm can further determine whether these objects are really potentially visible.

The computation of the PVS concludes as the traversal finishes. Before starting the next frame, each voxel of $\mathcal{O}$ and $\mathcal{H}$ is classified as non-opaque and non-occluded, respectively.

The view-frustum traversal can be terminated earlier if, during the plane traversal in the directions determined by a seed-voxel, only occluded voxels are detected. This means that the remaining voxels of the scene are hidden and the PVS will surely not be changed at least until the next frame.

4.3. Occluder Extension

The occluder extension is an adaptation of the blocker extension technique originally suggested by Schaufler *et al.* for octrees[12]. This process tries to aggregate maximally the adjacent opaque or occluded voxels of an initial opaque voxel in order to increase occlusion effectiveness. This set of aggregated voxels is called an *extended occluder*.

In Schaufler's technique, occluders are extended by searching for opaque or occluded voxels in axis-aligned directions, where the aggregation of voxels subtends a convex L-shaped occluder. Instead of using this heuristic which is restricted to convex occluders, we propose a novel strategy that explores the fact that we are using regular voxels and a fast occlusion computation without convexity constraints. The search for adjacent opaque or occluded voxels is restricted to the set of voxels that have the same relative chessboard distance of the current seed-voxel to the voxel containing the viewpoint. Due to the chessboard metric, this set of voxels always lies on axis-aligned planes (a maximum of six planes, which are the sides of a cube formed by these voxels). Recalling the natural correspondence between regular voxels and pixels, we consider each of these planes as bitmap images where opaque and occluded voxels are the opaque pixels (Figure 6). By vectorizing each bitmap, we build "cap" polygons of the occlusion volumes, which are polygons (possibly concave and with holes) that represent the top of the occlusion volumes as seen by the viewer. For a conservative result, the cap polygons should underestimate the size of the occlusion volumes in order to guarantee that only voxels totally inside the occlusion volume are classified as occluded. Our approach for a conservative rasterization simply considers that the center of each voxel is given in integer coordinates and uses them as vertices of the cap polygon. The effect we achieve is the same of building a cap polygon that generates oclusion volumes which are always inside a *from-region* occlusion volume (*i.e.*, an occlusion volume valid for all possible point-of-views inside the voxel containing the viewer), thus ensuring a conservative result.

4.4. Occlusion Computation

With cap polygons computed, we should generate the occlusion volumes to query the occlusion state of voxels with respect to the viewer. The occlusion volumes are generated by simply extending each vertex of the cap polygons along the focal axis determined by the considered vertex and the viewpoint. The occlusion volumes thus generated are semi-infinite polyhedra whose semi-infinite sides are collinear to the supporting planes of the viewpoint and the edges of the cap polygons, and the finite sides are the cap polygons. As the correspondence between pixels and regular voxels is straightforward (both pixels and voxels are regularly and uniformly distributed in a grid), we can compute the occluded voxels by rasterizing the clipped occlusion volumes in the data structure and setting the rasterized voxels to *occluded*. To do so, the occlusion volumes are sliced in regular cross sections, which are simple polygons, and each polygon is rasterized in $\mathcal{H}$ (Figure 7). By using this procedure of occlusion computation, we free the algorithm from explicit intersection tests between voxels and occlusion volumes. In addition, a large number of voxels can be quickly classified as hidden by taking advantage of span and scan-line coherence of the rasterization.

4.5. Adaptation for Static Scenes

Although static objects could merely be considered as dynamic objects of null motion, it is possible to handle these objects more efficiently taking into account that hidden static objects do not need TBVs at all, and static objects only need to be discretized once in the structure.

We have introduced a new occluder matrix, the *static* occluder matrix $\mathcal{O}_s$ that contains opaque voxels of static objects only. Each static object is discretized in $\mathcal{O}_s$ and $\mathcal{I}$ in a preprocessing stage. In runtime, the contents of $\mathcal{O}_s$ are transferred to $\mathcal{O}$ before starting the visibility determination for the current frame (this is required since $\mathcal{O}$ is re-initialized at the end of the view-frustum traversal). Finally, we do not allocate TBVs to static objects found in $\mathcal{I}$.

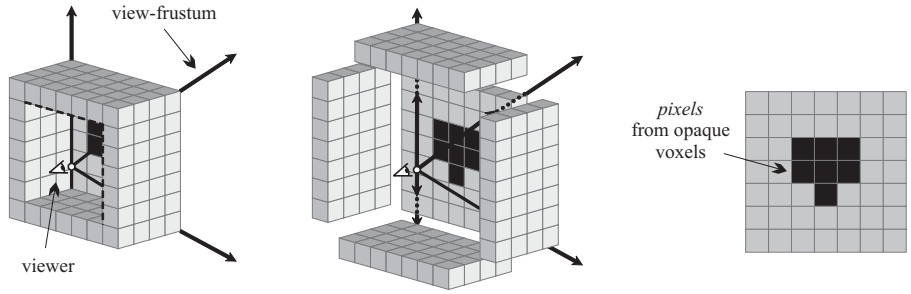

Figure 6: *Left: subset of voxels with the same chessboard distance to the viewer. Center: splitting the set of voxels in planes of voxels. Right: a bitmap from a plane of voxels.*

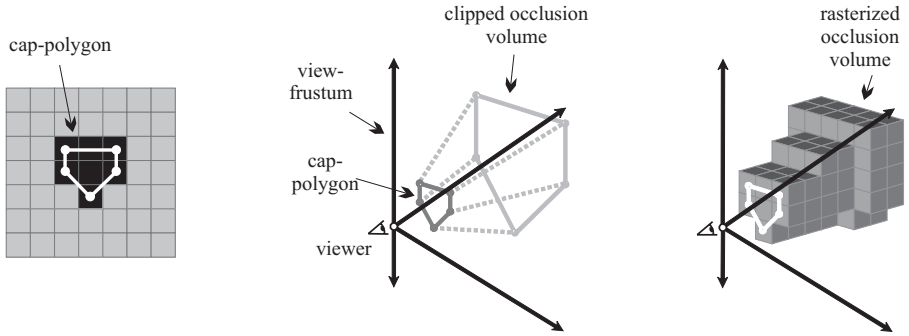

Figure 7: *Left: cap-polygon. Center: occlusion volume from the cap-polygon. Right: rasterized occlusion volume.*

5. Implementation and Results

The algorithm and visualizer has been implemented in C++ using OpenGL and tested on a PC with a 1.5 GHz Athlon XP processor and graphics accelerator using a GeForce2 GTS chipset. Snapshots are shown in Figure 8. The 3D scene used in the tests was a clustering of spheres of varying size and arbitrary motion. A different number of objects was used, up to 5,000 spheres that totalize 6 million of polygons.

Timing tests were only compared with hardware z-buffering since other implementations of dynamic scene occlusion culling techniques were not freely available or were not available as open source (*e.g*, the dPVS API[1]).

The first test (Figure 9) measured the rendering time of our regular grid visibility culling algorithm (RGVC) against standard hardware z-buffer (ZB) for an increasing number of hidden dynamic objects. The results show that the performance of our algorithm depends mostly on the number of potentially visible objects, while z-buffering has an unsurprising linear behavior. The overhead of handling TBVs is also negligible for most applications; frame rates of 100 FPS were obtained for more than 10,000 moving objects. The increase in the grid resolution from 50x30x50 to 100x60x100 has added less than a millisecond to this time. These results reflect the optimizations we used to discretize TBVs in the grid.

The second test (Figure 10) measured the rendering time (in frames per second) of a walkthrough in a scene with 200 dynamic objects and 4,800 static objects. The performance was again compared with standard z-buffering (ZB). Note that the rendering time of z-buffering was constant for all frames, while the performance of the regular grid approach was output-sensitive, producing uneven frame rates. We achieved speed-ups of up to ten times the performance of hardware z-buffering for the low resolution grid (50x30x50).

We also recorded the number of potentially visible objects and exact visible objects during the walkthrough in the scene used for the second test (Figure 11). Our algorithm (RGVC) and view-frustum culling only (VFC) were applied in order to verify the tightness of approximation of the conservative set to the exact visible set. We observed that there is a strong overestimation of potentially visible objects when using the low-resolution (50x30x50) grid. While increasing the resolution yields tighter results, the efficiency decreases due to the high number of voxels to be visited, as shown in Figure 10.

According to results presented so far, there is an evident trade-off between grid resolution and performance, but also between grid resolution and tightness of the PVS. In order to detect the resolution that yields the best performance results, we measured the rendering time as a function of grid

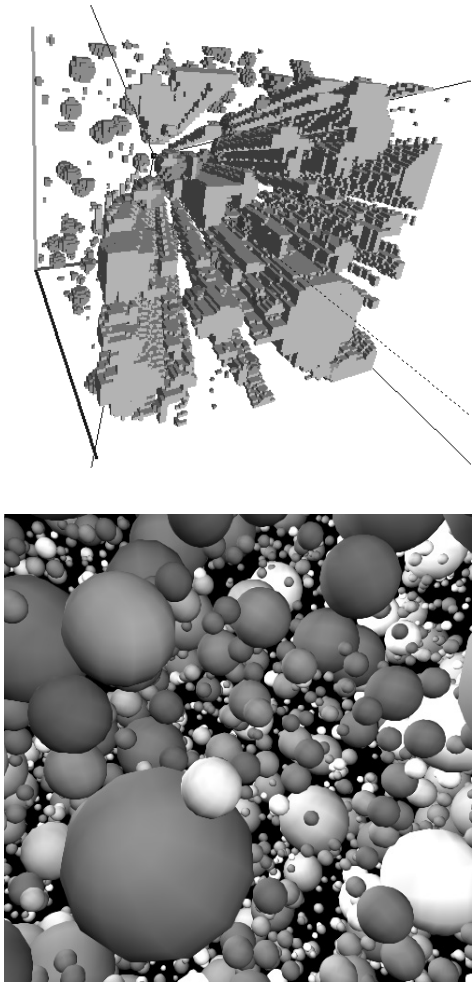

Figure 8: *Top: visualization of a 3D regular grid with 7,000 objects (resolution of* 100^3 *voxels) showing hidden and opaque voxels. Bottom: scene snapshot as seen from the viewpoint.*

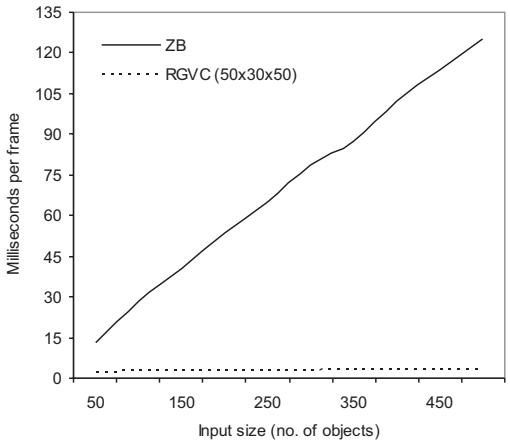

Figure 9: *Rendering time for an increasing number of hidden dynamic objects.*

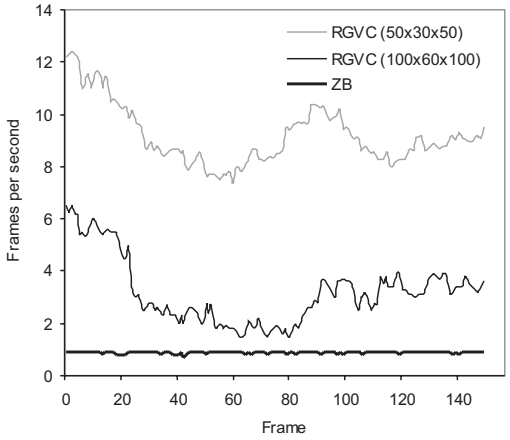

Figure 10: *Rendering time for a walkthrough.*

resolution for a fixed viewpoint in the scene of the walkthrough. The results are shown in Figure 12. The best compromisse was a resolution of approximately 70x70x70 voxels. Finally, the test shown in Figure 13 measured the tightness of approximation of the conservative set to the exact set as a function of grid resolution for the same fixed viewpoint used in the previous test. Unfortunately, the strong overestimation of potentially visible objects remains for the optimal grid resolution, as the size of the PVS is almost four times the ideal result (Aila and Miettinen[1] argued that satisfactory conservative sets should have a ratio of less than two times the exact set). We believe that the overestimation was due to our choice of using occluder fusion only in object-space, which is a strategy too dependent on grid resolution. For instance, we observed that the algorithm produces PVSs with several potentially visible objects even if only one object is enclosing the field of view. However, this is not a limitation of the regular grid and better results could be obtained by using image-space occluder fusion.

6. Conclusion and Future Work

We have presented an occlusion culling algorithm for densely occluded dynamic scenes based on a regular grid that uses opaque regions of the scene as occluders. Besides the efficiency of representing potentially visible dynamic objects and temporal bounding volumes, the benefits of using regular grids are strengthened by novel methods of view-frustum traversal and occlusion computation based on raster mathematics. In addition, the algorithm is output-sensitive:

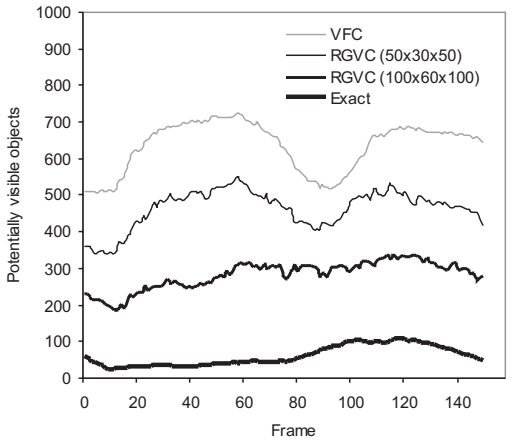

Figure 11: *Tightness of the conservative set for a walk-through.*

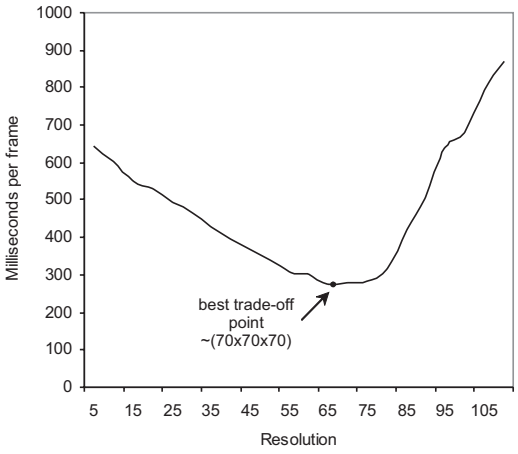

Figure 12: *Rendering time as a function of grid resolution.*

its runtime is proportional to the number of visible objects – both dynamic and static – and does not depend on the number of polygons that compose these models. Hence, it can be used in scenes of finely tessellated geometry and even in non-polygonal scenes.

An implementation of the algorithm is presented. According to the timing tests, the overhead due to the handling of hidden dynamic objects is very low for most scenes. For large and complex scenes, we achieve speed-ups of up to one order of magnitude compared with standard z-buffering (though this value greatly depends on the number of potentially visible objects). However, the results are still too conservative, and the limitation of the algorithm to closed objects (polyhedra) is not desirable. We believe that these drawbacks are a result of our choice of using occluder fusion in object-space only. Thus, they do not seem to represent limitations of the regular grid. In fact, other authors[1, 16] had already supposed that precise results would be obtained only when using occluders generated from the aggregation of occlusion in the frame buffer.

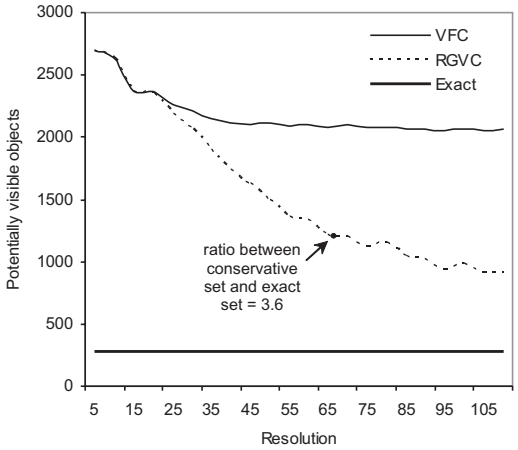

Figure 13: *Tightness of the conservative set as a function of grid resolution.*

For future work, we suggest to use both object-space and image-space occluder fusion in order to produce tight conservative results and handle arbitrary scenes. These improvements can be done by performing occlusion queries according to the coverage of voxels by the frame buffer and also by using these tests to discretize potentially visible objects in $\mathcal{O}$ instead of using the approach of intersection tests between voxels and simplified geometries. It is worth noting that our algorithm satisfies all requirements to be adapted to image-space methods such as HZB and HOM[1].

Another interesting topic for future work includes studying hierarquies of regular grids that can be used to trivially discard hidden voxels without the requirement of maintaining information in voxels different than those contained in the lowest level of the hierarchy. This approach seems to be feasible and could be used to cull-out large portions of the scene with results similar to those obtained with standard hierarchical spatial databases (octrees and kD-trees). Finally, we intend to compare the maintenance performance of TBVs in the regular grid against hierarchical approaches such as octrees and BSP trees.

Acknowledgements

This work was partially supported by Council for Improvement of Higher Education Personnel (CAPES), Brazil.

References

1. T. Aila and V. Miettinen. *dPVS Reference Manual Version 2.10*. Hybrid Holding, Ltd., Helsinki, Finland, October 2001. 2, 3, 7, 8, 9

2. J. M. Airey, J. H. Rohlf, and Jr. F. P. Brooks. Towards image realism with interactive update rates in complex virtual building environments. In R. Riesenfeld and C. Sèquin, editors, *Computer Graphics (1990 Symposium on Interactive 3D Graphics)*, volume 24, pages 41–50, March 1990. 1

3. D. Cohen-Or, Y. Chrysanthou, C. T. Silva, and F. Durand. A survey of visibility for walkthrough applications. *SIGGRAPH 2001 Course Notes*, August 2001. 1, 2

4. F. Durand. *3D Visibility: Analytical Study and Applications*. PhD thesis, Universite Joseph Fourier, Grenoble, France, July 1999. 2

5. F. Durand, G. Drettakis, J. Thollot, and C. Puech. Conservative visibility preprocessing using extended projections. In *Proceedings of SIGGRAPH 2000*, pages 239–248, July 2000. 2

6. N. Greene. Occlusion culling with optimized hierarchical z-buffering. *SIGGRAPH 2001 Course Notes*, August 2001. 2

7. N. Greene, M. Kass, and G. Miller. Hierarchical z-buffer visibility. In *Proceedings of SIGGRAPH '93*, pages 231–238, July 1993. 2

8. D. Luebke and C. Georges. Portals and mirrors: Simple, fast evaluation of potentially visible sets. In Pat Hanrahan and Jim Winget, editors, *1995 Symposyum on Interactive 3D Graphics*, pages 105–106, April 1995. ISBN 0-89791-736-7. 2

9. T. Möller and E. Haines. *Real-Time Rendering*. A.K. Peters Ltd., 2nd edition, 2002. 2

10. S. Morein. Ati radeon hyper-z technology. In *Hot3D Proceedings - Graphics Hardware Workshop*, 2000. 2

11. T. Pabst. High-tech and vertex juggling - nvidia's new geforce 3 gpu. *Toms Hardware Guide*, February 2001. 2

12. G. Schaufler, J. Dorsey, X. Decoret, and F. Sillion. Conservative volumetric visibility with occluder fusion. In *Proceedings of SIGGRAPH 2000*, pages 229–238, 2000. 1, 2, 3, 4, 6

13. A. Smith, Y. Kitamura, and F. Kishino. Efficient algorithms for octree motion. In *IAPR Workshop on Machine Vision Applications*, pages 172–177, 1994. 2

14. O. Sudarsky and C. Gotsman. Dynamic scene occlusion culling. *IEEE transactions on visualization & computer graphics*, 5(1):217–223, 1999. 1, 2, 3, 4

15. P. Wonka and D. Schmalstieg. Occluder shadows for fast walkthroughs of urban environments. In *Proceedings of Eurographics '99*, August 1999. 2

16. H. Zhang. *Effective Occlusion Culling for the Interactive Display of Arbitrary Models*. PhD thesis, Department of Computer Science, UNC Chapel Hill, 1998. 9

17. H. Zhang, D. Manocha, T. Hudson, and K. E. Hoff III. Visibility culling using hierarchical occlusion maps. In *Proceedings of SIGGRAPH '97*, volume 31, pages 77–88, August 1997. 2