Dynamic Scene Occlusion Culling using a Regular Grid

HARLEN COSTA BATAGELO WU, SHIN-TING

Unicamp – State University of Campinas, School of Electrical and Computer Engineering
{harlen,ting}@dca.fee.unicamp.br

Abstract. We present an output-sensitive occlusion culling algorithm for densely occluded dynamic scenes where both the viewpoint and objects move arbitrarily. Our method works on a regular grid that represents a volumetric discretization of the space and uses the opaque regions of the scene as *virtual occluders.* We introduce new techniques of efficient traversal of voxels, object discretization and occlusion computation that strengthen the benefits of regular grids in dynamic scenes. The method also exploits temporal coherence and realizes *occluder fusion* in object-space. For each frame, the algorithm computes a conservative set of visible objects that greatly accelerates the visualization of complex dynamic scenes. We discuss the results of a 2D and 3D case implementation.

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. In general, these scenes are *densely occluded*, which means that the fraction of visible geometry with respect to any viewpoint is only a small subset of the overall model [2]. The exhibition of densely occluded scenes can be accelerated by *occlusion culling* algorithms that quickly detect occluded geometry and avoid sending them to the rendering pipeline. However, the efficient visibility determination of arbitrary dynamic scenes is still an open area of research in computer graphics [4]. In general, occlusion culling algorithms require expensive preprocessing stages in order to build hierarchical data structures for efficient visibility queries in runtime, as large parts of the scene can be early classified as hidden in high levels of the hierarchy. Nevertheless, handling 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.* [13] and Sudarsky and Gotsman [15]. Our algorithm inherits most benefits from these algorithms, such as the ability to *fuse* occluders in object-space, the implicit use of dynamic *virtual occluders* [4] and the reduction in output-sensitive complexity (*i.e.*, its runtime is proportional to the number of visible objects only - see Sudarsky [15] for details). However, our algorithm is completely *online*; it does not depend on preprocessing, 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. We exploit the 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*; (3) optimize the discretization of *temporal bounding volumes* to reduce the overhead due to handling of hidden dynamic objects [15].

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 both 2D (section 4) and 3D scenes (section 5). We discuss the implementation results in section 6 and conclude with suggestions for future work in section 7.

2 Previous Work

For a comprehensive survey about visibility, we suggest Durand's thesis [5]. Visibility culling is specially covered by Cohen-Or *et al.* [4], Möller and Haines [10], 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 allow efficient visibility queries of dynamic objects (*i.e.*, answer whether a dynamic object is being occluded by some portion of the scene), they consider as occluders only static objects (*i.e.*, cannot answer whether a dynamic object occludes some part of the scene) [6, 13]. 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 [9] and 2.5D urban scenes [16], 3D cases remain almost unexplored (a remarkable exception is the *dPVS* API [1]). Besides the technique of *temporal bounding volumes* [15], we considered methods that can be further adapted to dynamic scenes, such as the *hierarchical z-buffer* [7, 8] 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 most graphics hardware (an exception is the nVidia's GeForce3, using *z occlusion culling* [12]). Recently, Greene has proposed changes of the original HZB for feasible hardware implementations [7]. The bandwidth traffic of z-values is greatly reduced with this new approach and, in some cases, it is 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 their 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 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, even considering that these methods work in dynamic scenes more efficiently than a traditional z-buffer (see *e.g.*, ATI's *Hyper-Z technology* [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 handling objects subjected to rigid-body transformations [14]. Sudarsky and Gotsman use temporal coherence to restrict the change of the octree to the small 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 [15].

In order to reduce the number of updates of the data structure, it is possible to associate to each dynamic object a region of space that completely encloses the object during a whole 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 suggest to calculate bounding volumes for short periods of time, called temporal bounding volumes (TBVs) [15]. 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 for any future time. This future time, or "TBV's expiration date", 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. Output-sensitivity with respect to the number of dynamic objects is achieved because the spatial database is updated only when the objects really happen to be potentially visible. The *dPVS* API [1], a commercial visibility culling library, handles dynamic objects by using TBVs. It organizes the scene geometry into an axis-aligned BSP tree that allows faster updates than octrees. The visibility culling algorithm is based on several optimizations of the HOM technique, which results in a very efficient culling solution for a broad class of complex scenes.

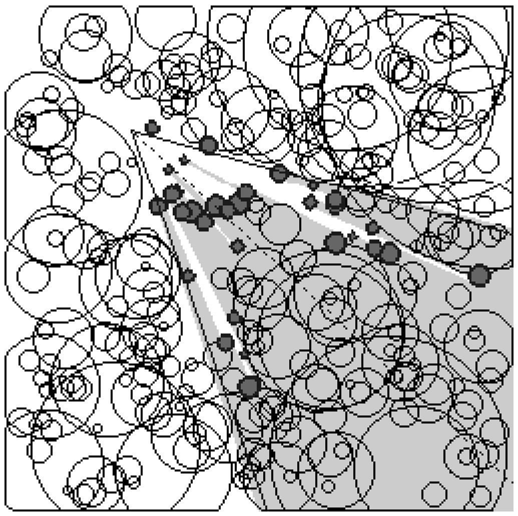


Figure 1: Visualization of the data in a 2D regular grid.

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 an approximate 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.*, 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 [13]. Only objects fully contained in occluded voxels are considered invisible. Therefore, the set of objects reported for rendering is always an overestimate of the visible objects.

For optimization purposes, we have organized the information of the regular grid into four matrices. For illustration, Figure 1 shows an overlaid view of these four matrices associated to a 256x256 regular grid that represents a 2D scene with 300 dynamic objects (32 potentially visible). Potentially visible objects are shown as filled circles. The line-of-sight and the view-frustum boundaries are shown as a dashed black line and two solid black lines, respectively. In the rest of the paper, for the sake of uniformity and without compromising the comprehension, we will call a regular grid element a voxel in both 2D and 3D scenes.

- *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. The opaque voxels are shown in dark gray in Figure 1.
- *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. In Figure 1 the occluded voxels are drawn in light gray.
- *Identifiers matrix* ($\mathcal{I}$), which associates to each voxel a list of identifiers (IDs) of objects that span its spatial region in the scene. In Figure 1 the non-empty $\mathcal{I}$-voxels are depicted as filled circles.
- *TBVs matrix* ($\mathcal{T}$), that associates to each voxel a list of IDs of TBVs that span its spatial region in the scene. Non-empty $\mathcal{T}$-voxels are in black (each TBV has the shape of an empty circle) in Figure 1.

We assume that each object has a unique ID, a maximum velocity and a flag indicating 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 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 an approximated 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 2D Case

For analyzing the proposed algorithm, we firstly tested it with dynamic 2D scenes. Instead of 3D view-frustum and occlusion volumes, 2D view-polygon and occlusion polygons were used. Since its extension to 3D requires minor modifications, its implementation is given in this section for the sake of clarity.

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 invisible objects are updated in $\mathcal{T}$. 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 is done by rasterizing the top-view orthographic projection of each object, then associating each pixel of the resulting frame buffer to a grid voxel. The only purpose of this stage is to identify the coverage of pixels by objects in a scene, so we can disable lighting, texturing and z-buffering.

The rasterization follows a sampling strategy that implies conservative results in the regular grid, *i.e.*, in which $\mathcal{O}$ underestimates the opaque voxels of the scene and $\mathcal{I}$ overestimates the voxels spanned by the objects. In particular, the outline of each object M is rasterized as a thick segments to determine the voxels that are partially covered by M. These voxels are classified as non-opaque ones and are considered to be spanned by M, thus the ID of M is added accordingly in $\mathcal{I}$. The remaining voxels are set as opaque ones in $\mathcal{O}$. The outline width required for a conservative rasterization is computed according to the work of Wonka *et al.* [16]: assuming a pixel-centered sampling strategy, each outline edge is grown and shrunk along its normal by $d/\sqrt{2}$, where d is the width of a voxel. The rasterizations of the grown and shrunk objects update $\mathcal{I}$ and $\mathcal{O}$, respectively.

The handling of temporal bounding volumes is based on the procedure proposed by Sudarsky and Gotsman [15]. It is only applied to invisible objects, and follows the following criteria: (1) Objects not contained in the current PVS, without TBVs, were potentially visible objects in the previous frame and are becoming invisible 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 invisible in the previous and are invisible in the current frame. In this case, if TBV validity period is expired, a new validity period is attributed.

For arbitrary dynamic objects, 2D TBVs should be bounding circles. As the correspondence between pixels and regular voxels is straightforward, a 2D TBV can be discretized in $\mathcal{T}$ by rasterizing a filled circle, then associating each pixel of the raster image to a grid voxel. Thus, TBV's IDs are added to all voxels that correspond to pixels of the rasterized filled circle (actually, we perform the rasterization directly in $\mathcal{T}$). However, we suggest a more efficient way to update TBVs in $\mathcal{T}$ for scenes where the viewpoint moves smoothly in the space. Instead of discretizing 2D TBVs as filled circles, we can update only the voxels that span the boundary of the empty circle. Indeed, this technique can be easily implemented by a modification of the Bresenham's circle algorithm for 4-connected curves. The illustration in Figure 1 shows TBVs discretized with this approach. 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 corresponding bounding circles do not enclose the viewpoint. The proof of correctness is shown elsewhere [3].

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 chess metric[1]. Besides avoiding expensive square root operations demanding by the Euclidian metric, the chess 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 the Bresenham's line algorithm and, from each voxel that contains the discretized line-of-sight (called *seed-voxel*), traverse the adjacent voxels that have the same chess distance to the voxel containing the viewpoint. For instance, let (x, y) be the position of a seed-voxel (voxels in dark gray in Figure 2) given in coordinates relative to the voxel containing the viewpoint, the traversal directions are: (1) $+y$ and $-y$ if $|x| > |y|$; (2) $+x$ and $-x$ if $|y| > |x|$; (3) $-x$ and $-y$ if $(|x| = |y|) \wedge (x > 0) \wedge (y > 0)$; (4) $+x$ and $+y$ if $(|x| = |y|) \wedge (x < 0) \wedge (y < 0)$; (5) $+x$ and $-y$ if $(|x| = |y|) \wedge (x < 0) \wedge (y > 0)$; (6) $-x$ and $+y$ if $(|x| = |y|) \wedge (x > 0) \wedge (y < 0)$. The traversal direction (arrows in Figure 2) 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.

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 comprehends 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

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

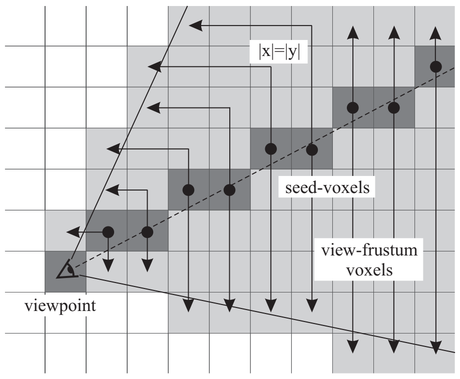


Figure 2: View-frustum traversal.

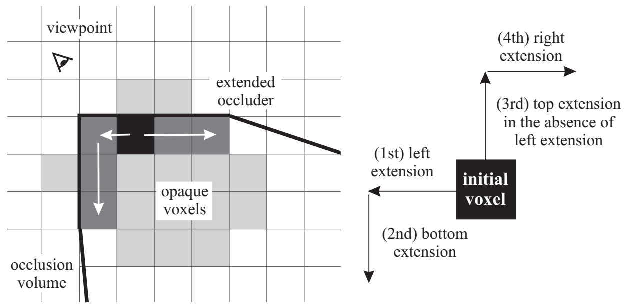
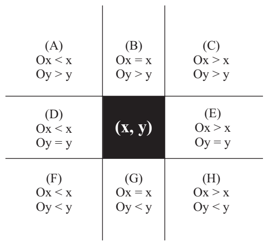


Figure 3: Occluder extension.



Extension directions:

(A): left, bottom, top, right
(B): left, right
(C): top, left, right, bottom
(D): top, bottom
(E): top, bottom
(F): bottom, right, left, top
(G): left, right
(H): right, top, bottom, left

Figure 4: Directions of extension.

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

The computation of the PVS is done when 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 traversal in the two 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 [13]. 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*, or *virtual occluder*, since it acts as an object-space occluder that actually is not part of the scene model.

Occluders are extended by searching for opaque or occluded voxels in axis-aligned directions, according to the method used by Schaufler *et al.* where the aggregation of voxels subtends a convex L-shaped occluder. The heuristic, however, is much simpler for regular voxels. Figure 3 shows an example of occluder extension using the following steps: (1st) left extension from the initial voxel (shown as a black voxel); (2nd) bottom extension from the left-most voxel determined by the first step; (3rd) top extension from the initial voxel; (4th) right extension from the top-most voxel determined by the third step. For the cases that the occluder is convex, the third step is executed only if the first step did not extend any voxel. In the general case, the directions of each step will depend on the relative position of the voxel (O_x, O_y) containing the viewpoint to the initial voxel (x, y) of the occluder extension, as shown in Figure 4.

4.4 Occlusion Computation

With extended occluders computed, an occlusion volume can be obtained to determine the visibility of the voxels with respect to the viewer. In 2D, the occlusion volume is a semi-infinite convex polygon whose semi-infinite edges are collinear to the supporting lines of the viewpoint and the occluder, and the finite edges are the edges of the occluder that are visible to the viewpoint. Therefore, we can compute the occluded voxels by rasterizing the clipped occlusion volume (in fact, an *occlusion polygon*) in the data structure. The voxels that are rasterized are considered as *occluded*. However, for a conservative result we should guarantee that only voxels totally inside the occlusion polygon are classified as occluded.

We can underestimate the number of occluded voxels with the strategy of polygon shrinking proposed by Wonka *et al.* [16] and then use a standard polygon rasterization algorithm. However, we have adapted this idea to an even simpler algorithm that makes no use of normal vectors, but possibly generates more conservative results. We consider that the center of each voxel is given in integer coordinates, and use them as vertices of the occlusion polygon. The slope of the semi-infinite edges is also computed considering that the viewpoint is centered in its voxel. Finally, we discretize the occlusion polygon into the regular grid with an ordinary convex polygon rasterization algorithm, where the sampling is taken in integer coordinates at the center of the voxels. The effect we achieve is the same of shrinking the edges of a *from-region* occlusion polygon (*i.e.,* an occlusion polygon valid for all possible point-of-views inside the voxel containing the viewer) by $(|n_x|+|n_y|)/\|\vec{n}\| \cdot d/2$, where d is the width of a voxel and $\vec{n} = (n_x, n_y)$ is the normal vector of the edge that should be corrected [16]. This term describes the greatest normal distance of a line from the center of a voxel to its boundary, thus implying a conservative rasterization.

4.5 Adaptation for Static Scenes

Although static objects could be merely considered as dynamic objects of null motion, it is possible to handle these objects more efficiently taking into account that invisible 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 $\mathcal{I}$ as 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 reinitialized at the end of the view-frustum traversal). Finally, we do not allocate TBVs to static objects found in $\mathcal{I}$.

5 3D Case

Unlike most visibility algorithms, the 2D regular grid occlusion culling algorithm can be easily adapted to the 3D case, as the visual events of occluders remain planar for axis-aligned voxels [5]. Therefore, most concepts used so far for the 2D case are the same for 3D scenes.

5.1 Scene Discretization

Although we could extend the strategy of 2D scene discretization to the 3D case (*e.g.,* by slicing the object in 2D planes and using the 2D algorithm for each cross section), we use pre-computed simplified models (bounding spheres and boxes) 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 more efficient.

5.2 View-Frustum Traversal

The view-frustum traversal starts at the seed-voxels of the light-of-sight discretized by a 3D Bresenham's algorithm. Similar to the 2D case, the traversal is performed in layers of voxels that have the same chess distance of the seed-voxel to the voxel containing the viewpoint. This includes a plane-sweep traversal instead of the two directions used for the 2D case. The catalogue of directions may be easily derived from the 2D case.

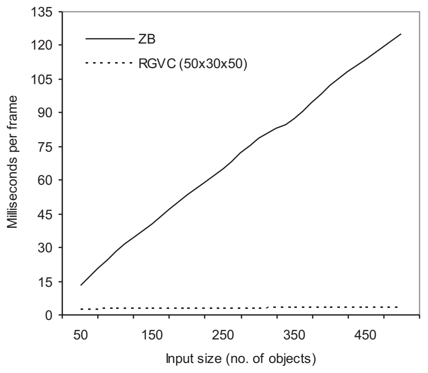
While we can discretize 2D TBVs as empty circles in order to reduce the number of $\mathcal{T}$-voxels changed, we can discretize 3D TBVs as spheres instead of balls. However, we can achieve further reductions of accesses of $\mathcal{T}$ by representing 3D TBVs as cube's faces. In addition, the decomposition of an sphere in circles is not so trivial as the decomposition of a cube in squares. The cubical TBVs are discretized by orthographically projecting the edges of the cube's faces onto the coordinate planes XY, XZ and YZ, then using three two-dimensional $\mathcal{T}$ matrices ($\mathcal{T}_{xy}$, $\mathcal{T}_{xz}$ and $\mathcal{T}_{yz}$) to store the projected TBVs of each plane. During the view-frustum traversal, TBVs are found by testing whether the current (x, y, z) voxel has the same TBV ID in $\mathcal{T}_{xy}$, $\mathcal{T}_{xz}$ and $\mathcal{T}_{yz}$. For a given ID, this existence test should 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 $\mathcal{T}_{yz}$.

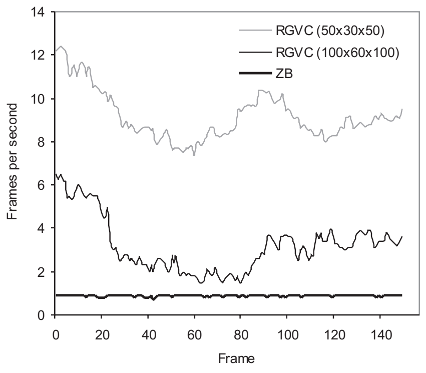
5.3 Occluder Extension and Occlusion Computation

Instead of using the 3D case heuristic proposed by Schaufler *et al.* [13], we propose a new 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 chess distance of the current seed-voxel to the voxel containing the viewpoint. Due to the chess 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 can consider each of these planes as bitmap images where opaque and occluded voxels are the opaque pixels. By vectorizing each bitmap, we build a "cap" polygon of the occlusion volume, which is a simple polygon (possibly with holes). This polygon is further extruded with respect to the viewpoint and each extruded cross section is rasterized in the data structure. Conservative results are ensured by using the sampling correction proposed in section 4.4.

6 Implementation and Results

The algorithm and visualizer for both 2D and 3D cases (Figure 6) 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. The





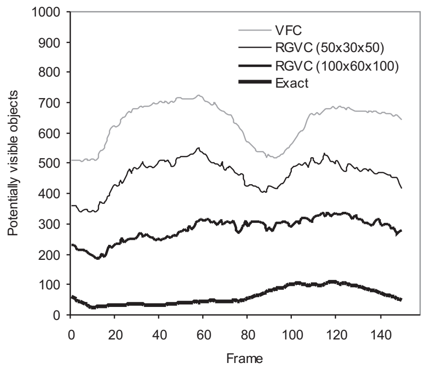


Figure 5: Test results.

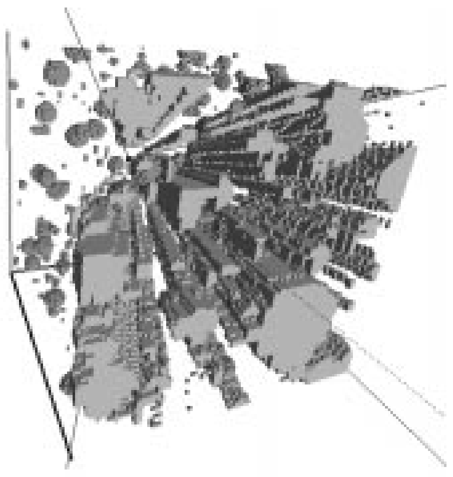
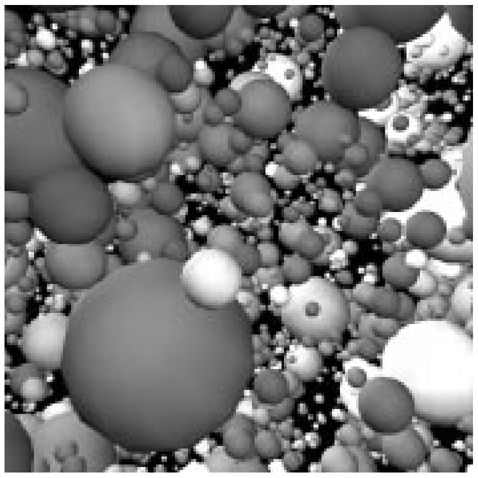



Figure 6: Top: visualization of a 3D regular grid with 7,000 objects (resolution of 100^3 voxels) showing hidden and opaque voxels. Bottom: scene snapshot as seen from the viewpoint.

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. Two grid resolutions were used: 50x30x50 and 100x60x100 voxels.

The first test (see Figure 5, left) 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 an average of only 0.20ms to this time.

The second test (Figure 5, center) measured the rendering time 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 is constant for all frames, while the performance of the regular grid approach is output-sensitive.

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 5, right), using our algorithm (RGVC) and view-frustum culling only (VFC) 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 due to our choice for only using a low resolution grid and occlusion fusion in object-space. While increasing the resolution yields tighter results, the efficiency decreases due to the high number of voxels to be visited, as shown in Figure 5 (center).

7 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 *virtual occluders*. Besides the efficiency of representing 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 principles. In addition, the algorithm fuses occluders in object-space in order to increase occlusion size and is output-sensitive: 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.

We have made an implementation of the algorithm for both 2D and 3D cases. 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. 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. It is worth noting that our algorithm satisfies all requirements needed to be adapted to image-space methods such as HZB and HOM [1]. Finally, we intend to compare the maintenance performance of TBVs in the regular grid against hierarchical approaches such as octrees and BSP trees.

References

[1] T. Aila and V. Miettinen. *dPVS Reference Manual Version 2.10*. Hybrid Holding, Ltd., Helsinki, Finland, October 2001.

[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.

[3] H. C. Batagelo and S. T. Wu. 2d dynamic scene occlusion culling using regular grids. Technical report, State University of Campinas, Campinas, Brazil, December 2001. ftp://ftp.dca.fee.unicamp.br/proj/prosim/publications/reports/batagelo-wu-2001-2dvis.pdf.

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

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

[6] 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.

[7] N. Greene. Occlusion culling with optimized hierarchical z-buffering. *SIGGRAPH 2001 Course Notes*, August 2001.

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

[9] 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.

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

[11] S. Morein. Ati radeon hyper-z technology. In *Hot3D Proceedings - Graphics Hardware Workshop*, 2000.

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

[13] G. Schaufler, J. Dorsey, X. Decoret, and F. Sillion. Conservative volumetric visibility with occluder fusion. In *Proceedings of SIGGRAPH 2000*, pages 229–238, 2000.

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

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

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

[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.