2D Dynamic Scene Occlusion Culling using a Regular Grid

Harlen Costa Batagelo Wu, Shin-Ting

Technical report – December 2001

Unicamp – State University of Campinas
School of Electrical and Computer Engineering
Department of Computer Engineering and Industrial Automation

Abstract

The growth of complexity in the scene geometry of interactive applications makes the dynamic scene visibility determination a problem of increasing importance in computer graphics. In this work, we present an efficient occlusion culling technique for 2D densely occluded scenes where both the viewpoint and objects move arbitrarily. Our method operates on a regular grid that represents a discretization of the space in volumetric cells and uses the opaque regions of the scene as occluders. We introduce several optimizations for fast discretization of objects and traversal of grid's cells that result in efficient updates of the data structure for multiple dynamic objects – regardless their number of polygons – and therefore encourage the use of regular grids with occlusion culling algorithms. The method also exploits temporal coherence to reduce runtime in proportion to the number of visible dynamic objects, and realizes *occluder fusion* by aggregating adjacent opaque and occluded cells in order to maximize occlusion effectiveness. For each frame, the algorithm computes an overestimate of potentially visible objects in an approximate front-to-back order that greatly accelerates the visualization of large and complex scenes. We discuss the results of a 2D case implementation and present suggestions of an easy extension to three-dimensional scenes.

1 Introduction

The efficient visualization of complex scenarios composed by hundreds of objects and millions of polygons is one of the most challenging problems in today's computer graphics. In general, the fraction of visible geometry with respect to any viewpoint in these environments is only a small subset of the overall model. Such scenes are called *densely occluded* [3, 8], and are commonly found in complex CAD models, urban scenes and indoor architectural scenes.

The exhibition and animation of densely occluded scenes can be greatly accelerated by algorithms that avoid sending primitives through the rendering pipeline by quickly discarding trivially hidden geometry. Methods based on this *culling* stage are called *visibility culling* techniques, and have gained important research effort in the last years [6, 7, 19]. However, the efficient visibility determination in dynamic scenes has been an open area of research in computer graphics [7]. In general, visibility culling techniques involves expensive preprocessing stages in order to build data structures for efficient visibility queries in runtime. These queries are commonly accelerated by hierarchical data structures since large parts of the scene can be early classified as hidden in high levels of the hierarchy. Nevertheless, the updates of the data structure for dynamic objects that change much of these hierarchy relations may be prohibitive to be done on-the-fly.

Although considerable research effort has been devoted to the acceleration of updates in hierarchical databases (see Section 2), in this work we suggest to abandon the use of such structures in dynamic scenes. Instead, we introduce a simple and flexible regular grid combined with several optimizations of discretization of objects and traversal procedures in order to perform efficient visibility queries and real-time updates for multiple dynamic objects. The visibility algorithm proposed to work with this structure is based on previous works of *occlusion culling*, specially on the approaches by Schaufler *et al.* [22] and Sudarsky and Gotsman [26, 25, 27]. The originality of our algorithm lies mainly on the optimizations used to adapt efficiently these cited works to a regular grid.

The regular grid represents a discretization of the space in volumetric cells (voxels). Each voxel identify volumetric features of the scene such as opaque/occluded regions and spanning objects. For each frame, the cells that span the view-frustum are traversed in an approximate front-to-back order, searching for opaque cells that can be used as occluders. According to the approach introduced by Schaufler et al. [22], each occluder can be extended by aggregating opaque and occluded cells in the neighborhood of the initial opaque cell, thus maximizing occlusion effectiveness. For each extended occluder, a *shadow volume* is computed and used to determine occluded cells according to the viewpoint. These occluded cells can act as new opaque cells which further increase the size of the next occluders, thus realizing *occluder fusion*, *i.e.*, the aggregation of sets of small and disjoint occluders to build larger and more effective ones. During this traversal of view-frustum cells, only objects entirely contained in occluded cells are considered invisible. Therefore, the set of objects reported for rendering is always an overestimate of the visible objects.

The maintenance of the grid follows a principle of *lazy evaluation* of the dynamic objects, which means that a dynamic object is updated in the data structure only when strictly needed. We use a technique of *temporal bounding volumes* [25, 26, 27] that achieves an output-sensitive complexity of updates in the data structure with respect to the number of visible dynamic objects. The decrease in the number of updates, combined with the simplicity of representing dynamic objects by regular grid's cells, results in an efficient occlusion culling solution for applications involving highly interactive and complex scenes.

The rest of the paper is organized as follows. We first review previous works on visibility culling techniques, emphasizing the approaches more suitable for dynamic scenes (Section 2). Then, we give an overview of our technique and describe the proposed data structure, in Section 3. We detail the algorithm in Section 4 and discuss, in Section 5, the results of the timing tests based on a 2D case implementation. In the following, we suggest ideas for extending the algorithm, along with its optimizations, for 3D scenes. Finally, in Section 7 we briefly describe the steps which should be followed in a future work.

2 Previous Works

The word visibility comprehends a vast number of problems in computer graphics, including hidden surface removal, shadow generation, form-factor calculation for radiosity, global lighting sim-

ulation, image-based rendering, object recognition, path planning and guarding of art galleries. For a more extensive survey, we suggest Durand's thesis [10]. Recent surveys focusing visibility culling algorithms, are presented Cohen-Or *et al.* [7] and Aila and Miettinen [2]. A discussion about the use of visibility culling techniques in games is shown by Riegler [21]. For previous surveys see the book by Möller and Haines [19] and Zhang's thesis [31].

Visibility culling techniques has been fundamental for the treatment of densely occluded scenes. In these scenes, most objects can be detected as trivially occluded with a runtime proportional to the number of visible primitives, and without the computational effort of more accurate analyses. This differs mainly from the traditional methods of hidden surface removal [12], which try to identify the exact fragments of the visible primitives by a process at least linear in the input size. In general, the set of objects reported as potentially visible (called PVS – *potentially visible set*) by a visibility culling algorithm is a *conservative* set, *i.e.*, it contains all objects at least partially visible, but maybe some invisible. For a correct rendering, only this small set is filtered by a traditional hidden surface removal algorithm, usually the Z-buffer [5].

Visibility culling comprehends three strategies that can be used in tandem: *back-face culling* [12, 16, 32], *view-frustum culling* [4, 12, 23] and *occlusion culling* [9, 11, 15, 18, 22, 30]. Such strategies avoid rendering, respectively, primitives that are not facing the viewer, primitives that are outside the view-frustum, and primitives hidden by some portion of the scene. In comparison with back-face and view-frustum culling approaches, occlusion culling involves a far more challenging problem due to complex global relationships of visual events among occluders and occludees [10, 20]. However, the use of occlusion culling has been inevitable for the efficient visualization of densely occluded scenes.

There are relatively few occlusion culling algorithms specially devoted to dynamic scenes when compared to the number of available literature about occlusion culling in static 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 the static objects (*i.e.*, cannot answer whether a dynamic object occludes some part of the scene) [11, 22]. On the other hand, in dynamic scenes of arbitrary motion, any object can be a potential occluder, for instance, moving right in front of the viewpoint and blocking entirely its field of view, or simply growing in size.

Based on previous works of visibility propagation through cells and portals in indoor architectural scenes [3, 28], Luebke and Georges [18] proposed an occlusion culling algorithm in which the elements that propagate visibility between rooms (*e.g.* doors and windows) can be added, moved or resized on-the-fly. However, this method is restricted to environments that can be divided in disjoint cells connected by portals of visibility propagation, as is the case of indoor architectural scenes.

Wonka and Schmalstieg [30] introduced a scheme of occlusion culling for 2.5D urban scenes without visibility precomputation or pre-selection of occluders, thus capable of handling dynamic occluders. The scene is discretized in a 2D regular grid such that occluders near the viewer can be quickly detected in runtime and updated efficiently. The shadow volumes of the selected occluders (in general, façades of buildings or walls) are rendered onto a Z-buffer with a top orthographic view of the scene in which each pixel coincides with a grid cell. In order to classify the occluded regions, the height of each cell is compared with the depth value of the corresponding Z-buffer pixel. An object is stated as hidden if the height of each cell that intersects it implies a depth value greater than the depth value indicated by the coincident Z-buffer pixel.

For the handling of dynamic three dimensional scenes, we emphasize the algorithms that work in image-space precision, such as the *hierarchical Z-Buffer* [13, 14] and the *hierarchical occlusion map* [33].

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 furthest Z-value of 2x2 arrays of pixels of the current level to one pixel of the subsequent level, beginning with the base of the pyramid that is a traditional Z-buffer. Thus, each level has half the resolution of the preceding level and the tip of the pyramid is a single pixel containing the image's furthest Z-value. In runtime, the octree is traversed in front-to-back, top-down order, and each node is compared with the pyramid of Z-values, beginning with the finest level where pixels still cover the projection of the bounding box of the tested node. If a node is completely occluded, then its sub-nodes and objects contained in its interior are removed. On the contrary, the test is recursively repeated for the sub-nodes. Objects associated with visible leaf nodes are rendered and used to update the pyramid.

HZB has the important advantage of considering every visible primitive as a potential occluder, thus achieving a complete occlusion fusion. On the other hand, the main drawback to use HZB as a more powerful alternative to traditional hardware Z-buffer is the necessity of reading back Z-buffer data. Unfortunately, most accelerators are too slow on Z-buffer queries, or simply cannot perform this operation.

An alternative approach to HZB that does not depend on special graphics hardware, is the technique of hierarchical occlusion maps (HOM). It works like the HZB, but tends to be more conservative and requires the precomputation of an occluder database. The visibility test is decomposed in an overlay test and a depth test. The hierarchical occlusion map is used in the first test. It consists in a pyramid of maps similar to a HZB pyramid that contains opacity values instead of depth values. For each frame, a HOM is built for a large group of occluders extracted from the occluder database. The scene geometry, previously organized as a bounding box hierarchy, is tested for coverage against that pyramid. The depth test is then performed only for the geometry that covers (both entirely and partially) discretized occluders in the HOM. An object is stated as 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 the HZB and HOM are substituted by oriented bounding boxes. Only the pyramid hierarchy is maintained. In the HOM technique, the precomputation of an occluder database is abandoned. Instead, occluders are chosen in runtime according to the size and distance from the viewer. The cost to select a good set of occluders on-the-fly is reduced by using frame coherence. However, even considering that these methods, when implemented in hardware, work in dynamic scenes much more efficiently than a traditional Z-buffer approach, the complexity of visibility determination is still at least linear in the number of input objects. All objects are tested against the pyramid, even those that do not contribute any pixel to the final image. This complexity can be prohibitive in densely dynamic scenes of millions of polygons.

The main problem in handling dynamic scenes is the difficulty to update efficiently 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 made about the adaptation of octrees for dynamic scenes. Based on the works of Ahuja, Nash and Weng, [1, 29], Smith *et al.* [24] present an algorithm for efficient adaptation of octrees for objects moved by rigid transformations that can be decomposed in translations and rotations of voxels. Libes [17] presents a technique to represent octrees models that expand or shrink arbitrarily. Sudarsky and Gotsman [26, 25, 27] use temporal coherence to update the octree only to the small voxel that

still encloses both the previous and current positions of the modified object, called *least common ancestor* voxel. Eventually, the least common ancestor can be the root level of the hierarchy, but for objects that move smoothly it is expected that such case does not happen.

In order to reduce the number of updates of the data structure, it is possible to designate for each dynamic object a region of space that completely encloses the object during an entire sequence of animation. These bounding volumes can be inserted in the spatial data structure such that the corresponding dynamic objects can be ignored until the visibility culling algorithm classifies the bounding volumes as potentially visible. Such regions can be determined, for instance, for sweeping volumes described by revolving doors, railroads, moving parts of fixed machines, and any other trajectory for which the motion is constrained.

The performance in these cases depends on the "tightness" of the bounding volumes. In one extreme, nothing is known about the future positions of the dynamic objects, and the bounding volumes are equal to the bounding volume of the entire scene. Since these volumes are always visible, all dynamic objects are updated for all frames, and the output-sensitivity is lost. In the other extreme case, the bounding volumes are so tight that they are coincident with the objects' bounding boxes. In fact, these objects are static; there is no space to move, and the visibility culling is as output-sensitive as a traditional algorithm for static scenes.

In scenes containing dynamic objects of arbitrary motion, it is not always possible to find bounding volumes for complete periods of the animation. In fact, these periods can be unlimited, and if they were determined, they would probably have the size of the scene.

Instead of designating a bounding volume for a full sequence of animation, Sudarsky and Gotsman [26, 25, 27] suggest to calculate bounding volumes for short periods of time, called temporal bounding volumes (TBVs). 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 guarantee to contain this object for any future time. TBVs do not need always to be spheres; they may assume other shapes provided that some dynamic characteristics of the objects are known, like changes in velocity and direction of motion. It is assumed that each dynamic object can have a TBV assigned to it from a given starting frame until a finish future frame. This future frame constitutes the "TBV's expiration date"; the time interval until this date is 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 obtained because they are considered to update the data structure only when they really happen to be potentially visible. In most frames the algorithm do not waste time updating trivially hidden dynamic objects, not even testing if such objects are hidden, since they are simply ignored during the traversal of the data structure. The *dPVS* API [2], 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 general complex scenes.

3 Definitions and Overview

The scene database we use is a regular grid that represents a discretization of the scene in volumetric cells. Each cell maintains a set of values that identify volumetric characteristics of its discrete region of the scene, such as opacity, visibility and spanned objects. For optimization purposes, we have organized these values into four

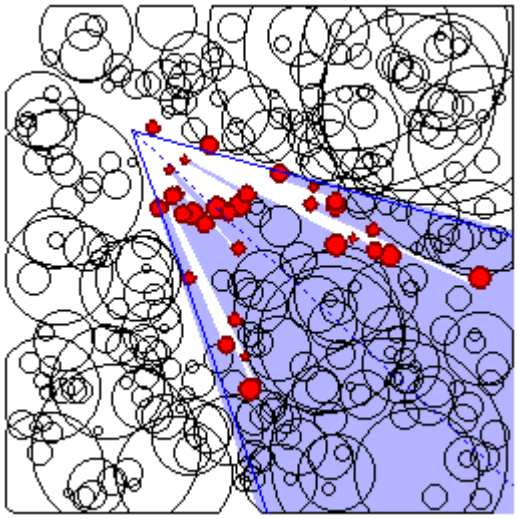

Figure 1: Compound visualization of the regular grid for a 2D scene with 300 dynamic objects (32 potentially visible) with a resolution of 256x256 cells. Potentially visible objects are shown in red; hidden cells in light blue and cells containing TBVs of hidden objects are shown in black. View-frustum boundaries are visualized as solid blue lines. The line-of-sight is shown as a dashed blue line.

matrices, visualized in a compound mode in Figure 1 and detailed in the following:

- *Occluders matrix* ($\mathcal{O}$): Classifies each cell[1] as *opaque* or *non-opaque*. A cell is opaque if it is fully inside the solid interior of a potentially visible object. Opaque cells are shown in light red in Figure 1.
- *Occlusion matrix* ($\mathcal{H}$): Classifies each cell as *occluded* or *non-occluded*. A cell is occluded if it is entirely hidden by one or more opaque cells with respect to the viewer. Figure 1 shows occluded cells in light blue.
- *Identifiers matrix* ($\mathcal{I}$): Associates for each cell a list of identifiers (IDs) of objects that span its spatial region in the scene. Figure 1 shows non-empty $\mathcal{I}$-cells in light and dark red.
- *TBVs matrix* ($\mathcal{T}$): Associate for each cell a list of IDs of TBVs that span its spatial region in the scene. IDs of TBVs may have the same value of the IDs of the objects the TBVs belong to. Figure 1 shows non-empty $\mathcal{T}$-cells in black.

We assume that each object has a unique ID, a maximum velocity and a flag indicating whether a TBV is associated with the object. When this flag is true, the object should also provide a TBV expiration date, a TBV position and a TBV diameter (for objects of arbitrary motion, TBVs are always of circular shape in 2D and spherical shape in 3D). Finally, the algorithm must keep the PVS result of the last frame so we can gather information about visibility coherence.

The dynamic scene occlusion culling algorithm is defined by the following procedures which are executed for each frame:

- **Scene discretization:** Objects reported in the PVS of the last frame are discretized in $\mathcal{O}$ and $\mathcal{I}$. The remaining objects are used to update $\mathcal{T}$ according to their TBVs.

[1] We use the word *cell* to designate both the matrix element and the area or volume of the scene represented by this element.

- **View-frustum traversal:** Cells that span the view-frustum are traversed in an approximated front-to-back order from the viewer. Objects that span each cell, as stated by $\mathcal{I}$, and at the same time declared non-occluded in $\mathcal{H}$, are added to the PVS of the current frame. Opaque cells are handled according to the procedures of *occluder extension* and *occlusion computation* as follows.

- **Occluder extension:** Each opaque non-occluded cell found during the view-frustum traversal is extended to adjacent opaque and occluded cells, as stated by $\mathcal{O}$ and $\mathcal{H}$, respectively, in order to maximize the angle subtended by the occluder and viewpoint. This set of aggregated cells defines the so-called *extended occluder*.

- **Occlusion computation:** For each extended occluder, a shadow volume is computed and discretized in $\mathcal{H}$, such that cells hidden by the occluder are classified as new occluded cells.

4 Algorithm

4.1 Scene discretization

As a first procedure for the current frame, all objects reported 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 $\mathcal{T}$. Note that in the very first frame, all objects are handled as if they were potentially visible. In addition, no object has a TBV assigned, since at this point the algorithm cannot say which objects are hidden.

In 2D, 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 cell. Since the only purpose of this rasterization stage is to identify opacity and coverage of pixels associated to cells, the lighting, texturing, and Z-buffering can be disabled.

The rasterization for scene discretization should follow a strategy that generates conservative results, *i.e.*, in which $\mathcal{O}$ underestimates the solid volumes of the scene and $\mathcal{I}$ overestimates the volumes spanned by the objects. The ideal rasterization, *i.e.*, the one that produces the less conservative discretization possible, is obtained with area anti-aliasing (see Figure 2). Area anti-aliasing defines the opacity of pixels according to the exact percentage of coverage of the geometry projection incident on these pixels. Thus, we can associate opaque cells only to pixels entirely covered by the object's projection, which are the pixels with maximum opacity values. For instance, if an object is rendered in white over a black background, only the cells associated with pure white pixels are regarded as opaque, since they are totally covered by the projection of the object's geometry. Also, all pixels with different colors than the background color (*e.g.* grey shades) correspond to cells at least partially spanned by the object, and the object's ID should be added to the list of each corresponding cell in $\mathcal{I}$.

Unfortunately, area anti-aliasing is usually too expensive in software and unavailable in most popular graphics hardware. An alternative, more efficient solution, yet potentially more conservative, consists in rasterizing the object with a thick outline that overestimates the cells at least partially covered by the geometry's projection, and associate these outline pixels to non-opaque cells spanned by the object (see Figure 3). The correspondence between pixels and cells of $\mathcal{O}$ and $\mathcal{I}$ is the same as before. The outline width needed for a conservative rasterization is computed according to the method used by Wonka *et al.* [30]. Assuming a pixel-centered sampling strategy, each outline edge is grown and shrunk along its normal by an epsilon of half the largest cell diagonal, *i.e.* , $\varepsilon = d/\sqrt{2}$,

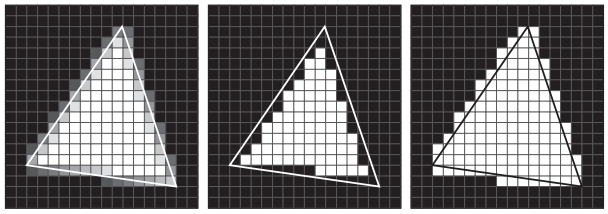

Figure 2: Ideal discretization using area anti-aliasing. Left: the rasterization (exact outline in solid lines). Center: opaque cells. Right: cells spanned by the object.

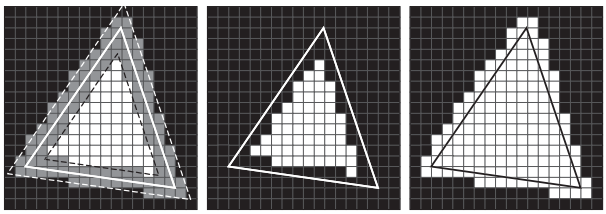

Figure 3: Conservative discretization using thick outline. Left: rasterization and sampling correction for conservative discretization (in dashed lines). Center: opaque cells. Right: cells spanned by the object.

where d is the cell width. The rasterizations of the grown and shrunk objects update, respectively, $\mathcal{I}$ and $\mathcal{O}$. This procedure can be even simplified in OpenGL by rasterizing the object's outline just as a loop of lines of width 2 or 3 pixels. The correct value for a conservative rasterization will depend on how the OpenGL implementation handles thick lines without anti-aliasing.

The handling of temporal bounding volumes is based on the procedure described by their authors, Sudarsky and Gotsman [27]. This stage handles only invisible objects, according to the following criteria: (1) Objects without TBVs and not contained in the current PVS were potentially visible objects in the preceding frame that are now invisible. In this case, new TBVs are designated to them and $\mathcal{T}$ is updated accordingly. (2) Objects with TBVs and not contained in the current PVS were invisible objects in the last frame that continue to be invisible in the current frame; nothing is done unless the TBV expiration date is reached. When this occurs, the TBV is removed from $\mathcal{T}$ and reinserted by a TBV with a new validity period.

The handling of objects that had their TBVs revealed is performed during the traversal of view-frustum cells, described in the next subsection.

TBVs validity periods are chosen according to the *adaptative* strategy proposed by Sudarsky *et al.* If an object is hidden but its TBV has been expired, we conclude that the TBV validity period was too short, since the object could stay more time without being updated. Thus, a longer expiration date is chosen for the next TBV. On the other hand, if the TBV has been revealed before the expiration date, then the TBV was too big and loose, and a smaller validity period is assigned to the new one. Therefore, fast moving or usually visible objects are expected to have TBVs of short validity periods over the time. As well, objects with a less dynamic behavior and invisible most of the time, tend to have TBVs of longer validity periods and are updated less frequently. On the other hand, we have noted that the efficiency of updates in $\mathcal{T}$ decreases approximately in the reason that the TBVs grow in size, because the number of cells spanned by the TBV is greater and an increasingly number of cells must be accessed at the time the TBV is inserted. If the TBVs are big enough such that the cost to update $\mathcal{T}$ is greater than the cost to update the original objects in $\mathcal{I}$, the frames where the

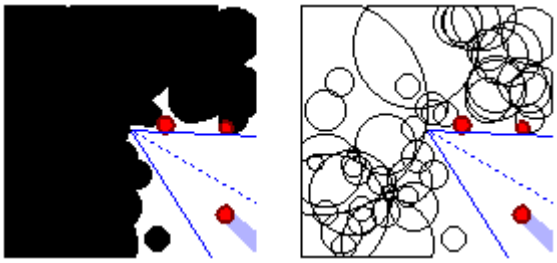

Figure 4: Optimized discretization of TBVs. Left: traditional discretization (overlapping TBVs are not distinguishable). Right: optimized discretization. The number of non-empty cells of $\mathcal{T}$ is considerably smaller.

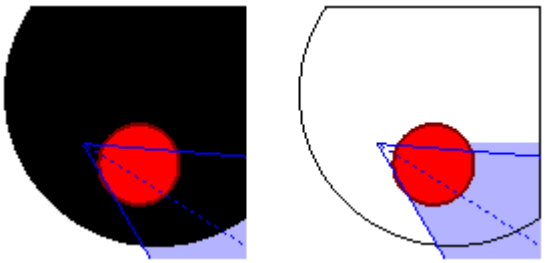

Figure 5: Left: with the traditional discretization the TBV is correctly revealed during the traversal of view-frustum cells. Right: TBV erroneously reported as invisible because the discretization of the circumference arc contained in the view-frustum is totally hidden by the red object.

TBVs are updated may lose performance and result in uneven animation sequences. In order to avoid this behavior, we have limited the maximum diameter of TBVs to a (empirically chosen) threshold proportional to the number of cells spanned by the object and its maximum velocity. Thus, objects of fast motion and many spanning cells tolerate the update of $\mathcal{T}$ for bigger TBVs, while objects of slow motion and few spanning cells limit the size of their TBVs by smaller diameters. This scheme has been worked as a satisfactory solution to the trade-off between the number and time of updates of both objects and TBVs.

We suggest a more efficient way to update TBVs in $\mathcal{T}$ for scenes where the viewer moves smoothly through the space. In 2D, instead of discretizing a TBV as a filled circle, we can discretize only the empty circle that overestimates the TBV, thus greatly reducing the number of accesses of cells of the data structure (see Figure 4). This technique can be implemented efficiently with the well-known Bresenham's circle rasterization algorithm [12]. We next show that the algorithm using this new optimized discretization is correct, *i.e.*, all TBVs are correctly detected regardless the reduction in the number of discretized cells. First, we consider that the viewer always describes a smooth 8-connected path along the grid's cells. If the viewer describes a 4-connected path, he cannot enter into a TBV without detecting it, since the Bresenham's rasterization produces an 8-connected circle. However, if the viewer enters the circle by a not 4-connected path, the TBV can be erroneously ignored. To solve this problem we modify the Bresenham's algorithm to rasterize a 4-connected circle.[2] Therefore, for a viewpoint outside the TBVs' circles, all TBVs are correctly detected. For a viewpoint inside a TBV circle, if the discretization of the circumference arc contained in the view-frustum is fully hidden by an occluder, *i.e.*, each TBV's cell coincide with occluded cells, the TBV is wrongly ignored (see Figure 5). A simple solution for these cases is to ensure that the validity period of each TBV is chosen such that the implied radius of the bounding circle is not greater than the distance of the cell containing the center of the circle to the cell containing the viewpoint. Thus, TBVs never enclose the viewer, and since we already showed that the viewer could not enter a TBV without being detected, the correctness of the algorithm holds.

[2] In practice, these non-detections of TBVs rarely produce non-conservative results. Although trivial, the suggested solution was not implemented here in order to maintain the number of TBV cells at a minimum.

4.2 View-Frustum Traversal

The visibility determination is actually done in the view-frustum traversal. It comprehends the traversal of grid's cells that span the view-frustum in order to identify occluders and potentially visible objects. Both are found as non-occluded cells. The former corresponds to opaque cells, and the latter to cells containing non-empty ID lists of $\mathcal{I}$ or $\mathcal{T}$.

The traversal of view-frustum cells is performed in a front-to-back order from the viewer. Therefore, the number of occluders found is equal to the number of potentially visible occluders, and the algorithm does not waste time handling hidden occluders. In addition, the PVS can be determined incrementally in only one traversal.

In order to compute efficiently the distance from the viewer to the cells, and hence perform a front-to-back traversal, we have used the chess metric.[3] While avoiding expensive square root operations of the Euclidian metric, the chess metric induces a fast traversal in axis-aligned directions only. Since the line-of-sight is always within the view-frustum, it is possible to discretize it incrementally from the viewer using the Bresenham's line algorithm [12] and, from each cell that contains the discretized line-of-sight (called *seed-cell*), traverse adjacent cells that have the same chess distance from the viewer. Starting from any seed-cell, the traversal in the 2D case is always performed in two axis-aligned directions easily determined by the signal of the coordinates of the seed-cell relative to the viewer. For instance, let (x, y) be the position of a seed-cell given in coordinates relative to the cell containing the viewer, the traversal directions can be: (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 $(|x| = |y|) \wedge (x > 0) \wedge (y < 0)$. Moreover, these directions are changed only when a cell with equal absolute value coordinates is reached. If this happens, the direction proceeds in a perpendicular direction to the original one, and the traversal ends when a cell completely outside the view-frustum is reached. Figure 6 shows this traversal procedure from the seed-cells.

During the traversal, if a non-occluded cell is reached, all objects contained in its ID list of $\mathcal{I}$ are added to the PVS of the current frame. Opaque non-occluded cells are considered as occluders. Such occluders should determine the hidden cells with respect to the viewer (this step comprehends the processes of occluder extension and occlusion computation, detailed in the next sections). Non-occluded cells that contains TBVs according to $\mathcal{T}$, indicate that the objects which own these TBVs may be visible. Therefore, their TBVs are removed of $\mathcal{T}$ and dissociated of the assigned objects. In addition, these objects are immediately discretized in $\mathcal{I}$ and $\mathcal{O}$, so the algorithm can further determine, during the rest of the traversal, whether these objects are in fact potentially visible.

When the traversal finishes, the PVS was determined completely. For the start of the next frame, each cell of $\mathcal{O}$ and $\mathcal{H}$ is classified as non-opaque and non-occluded, respectively.

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

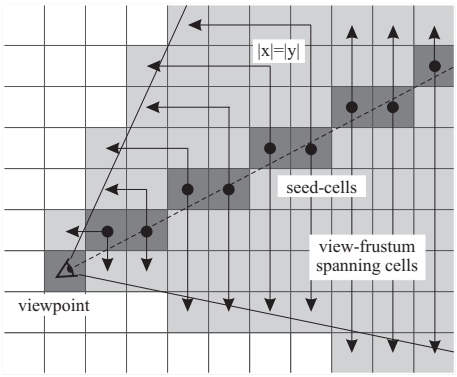

Figure 6: View-frustum traversal. Seed-cells are shown in dark gray. The arrows indicate the traversal directions from each seed-cell according to the chess metric. Note that the traversal directions changes only when $|x| = |y|$ for a cell with coordinates (x, y) relative to the cell containing the viewer.

The view-frustum traversal can be finished early if, during the traversal in the two directions determined by a seed-cell, only occluded cells had been detected. This event means that the remaining cells of the scene are hidden and the PVS surely will not be changed at least until the next frame.

4.3 Occluder Extension

The process of occluder extension described here is an adaptation of the blocker extension technique used by Schaufler *et al.* [22] originally for octrees. The idea consists in aggregating opaque and occluded cells to the initial opaque cell in order to maximize the angle between the viewer and occluder, thus increasing occlusion effectiveness.

Each opaque cell found during the view-frustum traversal can be extended by aggregating adjacent opaque and occluded cells to the initial opaque cell. Occluded cells can be handled as opaque cells according to the argument that the viewer is incapable of distinguishing whether a hidden cell is opaque or not. However, when considering occluded cells as opaque cells it is possible to extend occluders into hidden space and increase their occlusion size, thus realizing the *occluder fusion*. Starting from this set of aggregated cells, called *extended occluder*, a shadow volume is computed in order to determine the occluded cells with respect to the viewer, which are the cells fully inside the shadow volume. In 2D, the shadow volume is a semi-infinite convex polygon whose semi-infinite edges are collinear to the support lines of the viewer and the occluder, and the finite edges are the edges of the occluder that are visible to the viewer. The occlusion caused by only one shadow volume from a extended occluder is, in general, more effective than the union of the occlusion caused by the shadow volumes individually built for each cell that form this same extended occluder. This happens because there are regions in the scene that are not being entirely hidden by only one cell, but by the combination of occlusion caused by two or more cells, as shown in Figure 7.

Extended occluders should have a shape from which the resulting shadow volume can classify interior cells as fast as possible. We decided to use convex shadow volumes only, and so the edges of the extended occluder which are visible to the viewer must be convex too. Therefore, occluders are extended by searching for opaque cells in axis-aligned directions only. This heuristic is analogous to the method used by Schaufler *et al.* [22] where the aggregation of cells subtends a L-shaped occluder. Figure 8 shows an example of

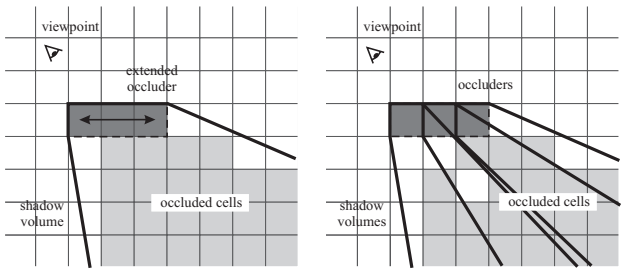

Figure 7: Occlusion effectiveness. Left: with occluder extension. Right: without occluder extension. The additional non-occluded cells are being blocked by the aggregation of the first two left occluder cells.

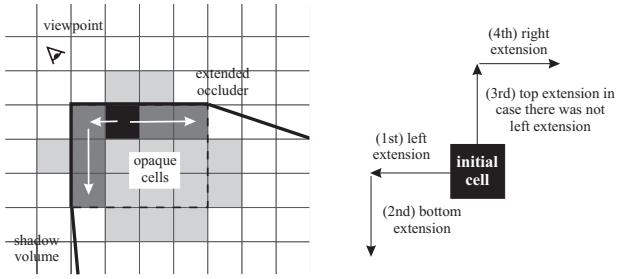

Figure 8: Left: occluder extended by the fusion of opaque cells in axis-aligned directions. The initial extension cell is shown in black. Right: extension heuristic used.

occluder extension composed of the following steps: (1st) left extension from the initial cell; (2nd) bottom extension from the left-most cell determined by the first step; (3rd) top extension from the initial cell; (4th) right extension from the topmost cell determined by the third step. Since our occluder should be convex, the third step is only executed if the first step did not extend any cell. In the more general case, the directions of each step will depend on the relative position of the cell containing the viewer to the initial cell of the occluder extension, as shown in Figure 9.

The occlusion computation performed just after the occluder extension (see next Section) is an expensive process since it involves updating a large number of cells in $\mathcal{H}$. Thus, it is desirable to accomplish this step for few occluders, possibly with big size and near the viewer, which encloses most of the scene geometry. We have used a scheme to determine good occluders based on the following formula of solid angle estimation proposed by Coorg and Teller [9]: $-a(\vec{n} \cdot \vec{v})/\|\vec{d}\|^2$, where a is the occluder area (squared number of occluder cells), $\vec{n}$ the occluder normal, $\vec{v}$ the direction of view and $\vec{d}$ the vector from the viewer to the center of the occluder. Since the occluder is always facing the viewer, we take $(\vec{n} \cdot \vec{v}) = 1$. Only extended occluders that exceed a minimum value of solid angle (we have empirically tuned this value) are considered for occlusion computation.

4.4 Occlusion Computation

Occlusion computation is the process of determining which cells are being hidden by a given occluder with respect to the viewpoint.

Subsequent to each occluder extension, we perform the occlusion computation by building a shadow volume of the occluder and testing the grid's cells against it. In the 2D case, the shadow volume is a polygon; more precisely, a convex polygon due to our occluder

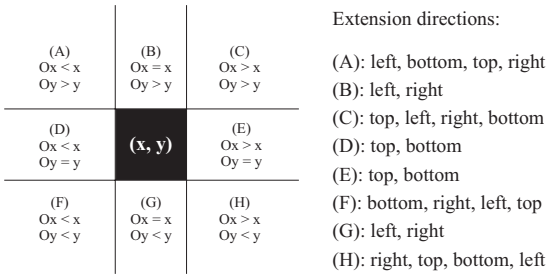

Figure 9: Partition of extension directions according to the position of the initial cell (x, y) relative to the cell (O_x, O_y) containing the viewer. The directions shown in Figure 8 correspond to those of partition (A).

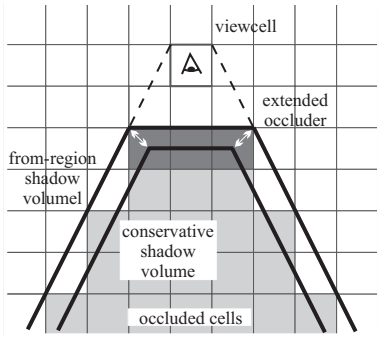

Figure 10: Efficient determination of a conservative shadow polygon. The cells classified as occluded are entirely contained in the from-region shadow polygon (sampling is taken at the center of the cell).

extension heuristic. Hence, we can discretize shadow polygons in the data structure by just adapting a convex polygon rasterization algorithm to change to *occluded* the cells coincident with the rasterized polygon. However, in order to maintain conservativeness, we must ensure that only fully hidden cells are classified as occluded.

The exact determination of the cells inside a shadow volume can be an expensive task even if spatial coherence is used to test few cells per sweep-line of rasterization. A more efficient approach, but that underestimates the number of occluded cells, consists in using the strategy of polygon shrinking along its normals as proposed by Wonka *et al.* [30]. We have adapted this idea to an even more simple algorithm that abstracts the use of normal vectors but generates more conservative results. We consider that the center of each cell is given in integer coordinates, and use them as vertices of the shadow edges coincident with the occluder edges. The slope of the semi-infinite edges is then calculated considering that the viewer is centered in its cell. Finally, we discretize the shadow polygon with an ordinary convex polygon rasterization algorithm, doing the sampling at the center of the cells. The effect we achieve is the same of shrinking the edges of a *from-region* shadow polygon (*i.e.*, a shadow polygon valid for each possible point-of-view inside the cell containing the viewer) by $(|n_x| + |n_y|)/\|\vec{n}\| \cdot d/2$, where d is the width of a cell and $\vec{n}$ is the normal vector of the edge taken. This term describe the greatest normal distance of a line from the center of a cell to the cell's boundary. A direct consequence of this fact is that the shadow polygon becomes valid for any viewpoint contained in the cell with the viewer – the *viewcell* – as illustrated in Figure 10.

4.5 Adaptation for Static Scenes

So far, we have discussed handling scenes where all objects, without exception, move arbitrarily. However, the applications we often found in practice handle most objects as static entities. For instance, in a urban environment, only cars are dynamic objects; buildings, walls and all remaining objects are static.

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 the following observations: (1) invisible static objects do not need TBVs at all; the TBVs are their geometries themselves. (2) visible static objects do not need to be discretized in the data structure for each frame, since we already know they are always at the same place. To handle these objects properly, we have introduced a new occluders matrix, the *static* occluders matrix $\mathcal{O}_s$ that contains opaque cells 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 because $\mathcal{O}$ is cleared at the end of the view-frustum traversal). Finally, we do not designate TBVs to static objects found in $\mathcal{I}$.

5 Implementation and Results

The 2D version of the algorithm has been implemented in OpenGL and tested on a PC with a 966Mhz Pentium III processor and graphics accelerator using a GeForce2 GTS chipset. We have used filled circles to represent the dynamic objects. Such objects were rendered as cylinders, that is how they would be seen by a *flatland* spectator (the height of the cylinders is immaterial), each cylinder composed by 1,040 triangles. The size, speed and position of the objects were determined at random. More specifically, the diameter of the circles were chosen within the range of 6 to 12 cells for a data structure of resolution 256x256, and the speed was chosen between 0.01 to 1.96 cells per frame.

On the tested scenes, the viewpoint was always located in the "corner" of the grid, looking along the greatest diagonal of the grid. For this configuration, the view-frustum traversal comprises a large number of spanning cells and the shadow polygons must set a great number of cells of $\mathcal{H}$. As a result, the visibility determination is somewhat slower than it would be for general viewpoints, and acts as a unfavorable case of the algorithm in practice.

On the first measurement we tested the behavior of the algorithm during the addition of dynamic and static objects for a grid resolution of 256x256 cells, as shown in Figure 11. The result is compared with traditional hardware Z-buffering (ZB) and three variations of input data for our regular grid visibility culling approach (RGVC), given as follows: (1) dynamic objects only; (2) mixed approach with a fixed number of 500 dynamic objects and an increasing number of static objects; (3) static objects only.

The results in Figure 11 show that the Z-buffering technique has a linear behavior, as we already expected. In contrast, the RGVC technique reveals a nearly constant increasing of processing time, proportional to the number of visible objects even handling dynamic objects only. The output-sensitivity is due to the use of the temporal bounding volumes. The RGVC for static scenes is faster because it does not need to handle cases of TBV expiration and discretization of objects for each frame. Curiously, the algorithm is slow for few objects, let they be dynamic or static. For instance, for 100 objects (104,000 triangles), the Z-buffering technique alone is still more efficient. This occurs because the scene is not densely occluded, and the algorithm needs to traverse more view-frustum cells, build more shadow polygons (see Figure 12) and take more potentially visible objects into account (see Figure 13). We note that this behavior is expected for an algorithm with runtime pro-

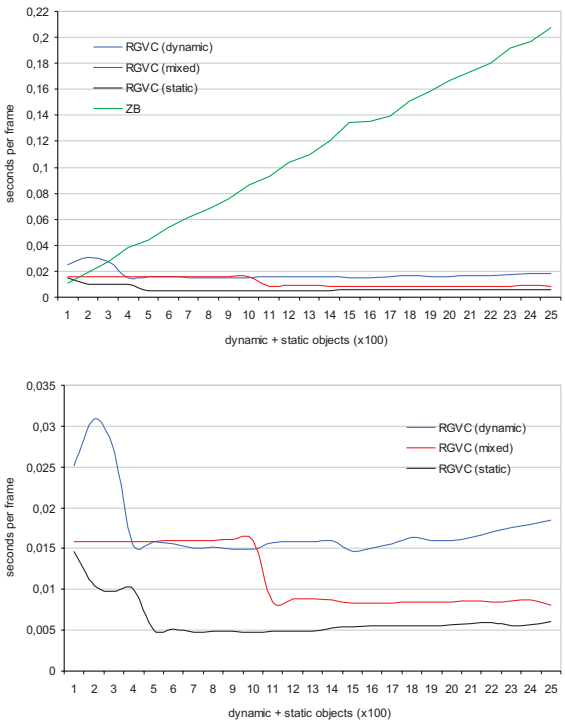

Figure 11: Time measurement for scenes with different number of dynamic and static objects. Above: comparison between three variations of regular grid visibility culling (RGVC) and Z-buffering (ZB). Below: the same graphic, but without comparison with ZB.

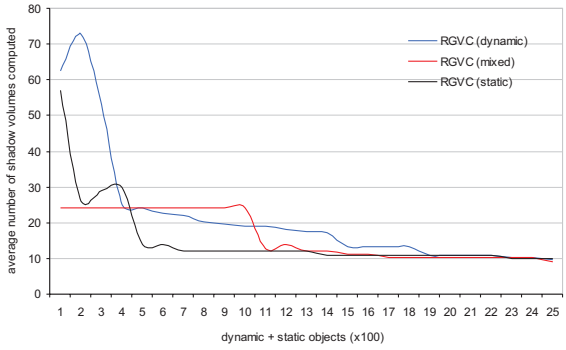

Figure 12: Average number of shadow polygons.

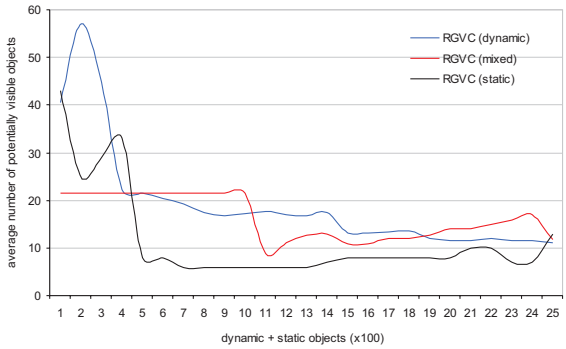

Figure 13: Average number of potentially visible objects.

portional to the number of visible objects. On the other hand, the algorithm may be eventually better than Z-buffering if the cost to render these few objects is higher than the runtime overhead of the occlusion culling algorithm.

We have also examined the behavior of the algorithm using data structures of different resolutions. In practice, the choice of the grid resolution depends on the volumetric nature of the scene. In general, the bigger the solid volumes of the objects in the scene, the smaller the resolution needed to obtain satisfactory results. This choice also respects a trade-off between accuracy and efficiency. The increase of the resolution implies accurate results, but augments the memory usage and the algorithm's time consumption as well. Figure 14 shows the timing results of executing the algorithm for a scene of 500 dynamic objects and 1,000 static objects for an increasing number of grid resolutions. The best performance is obtained with a resolution of 256^2 cells. For resolutions below 200^2 cells the algorithm tends to decrease efficiency. This occurs because the objects start to become proportionally too small to contain several cells in their interiors, *i.e.*, few cells are being classified as opaque and most objects are considered as potentially visible. For instance, in the extreme case of using a resolution of 1^2 cell, all scene objects would be reported as potentially visible. In order to consolidate this behavior, we also tested the "tightness" of aproximation to the visible set of the conservative solution reported by the algorithm in the handling of a same scene for different resolutions. As shown in Figure 15, the PVS reaches a nearly constant approximation average of 99% of the exact visible set[4] for resolutions above 200^2 cells. For greater resolutions, the gain of accuracy is negligible. This result also shows that the algorithm is effective despite the optimizations that underestimate the number of opaque and occluded cells.

[4]An exact visible set is one that contains all objects at least partially visible to the viewer, and only these objects. The worst solution of a conservative algorithm (0% of the exact visible set) is one that reports each object as potentially visible, although none is really visible.

6 Extension for 3D Case

For the handling of a 3D scene, the data structures $\mathcal{O}$, $\mathcal{H}$, $\mathcal{I}$, $\mathcal{T}$ and $\mathcal{O}_s$ are three-dimensional matrices, and the visibility determination can be performed according to the following changes of the 2D algorithm:

- **Scene discretization:** The discretization of objects in $\mathcal{O}$ and $\mathcal{I}$ can done by rasterizing the orthographic projection of the objects on the three coordinate planes, XY, XZ and YZ, such that a given (x, y, z) cell of $\mathcal{O}$ is considered opaque iff the corresponding pixels (x, y) in XY, (x, z) in XZ and (y, z) in YZ are mutually identified as opaque. Unfortunately, this discretization scheme works for convex objects only. Concave objects must be decomposed in convex components to be discretized separately. For a conservative rasterization, the strategy of using thick outlines or edges shrinking can be employed in much the same way as in the 2D case.

  For the discretization of TBVs, considering that in 2D we have used empty circles instead filled circles, in 3D we can use spheres instead of balls to reduce the number of changed cells in $\mathcal{T}$. The discretization of a sphere can be done with the Bresenham's algorithm for the rasterization of circles as cross-sections of the sphere. For instance, in order to rasterize a sphere with radius of r cells, we discretize $2r$ circles of

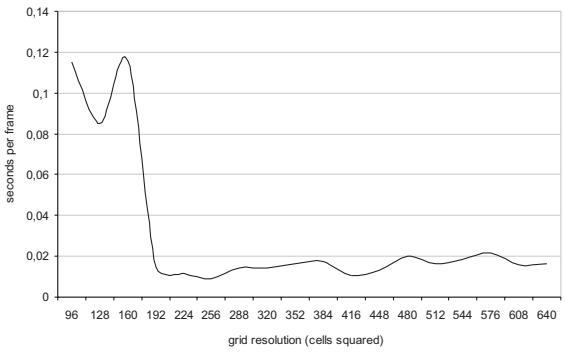

Figure 14: Average performance of a scene for different grid resolutions.

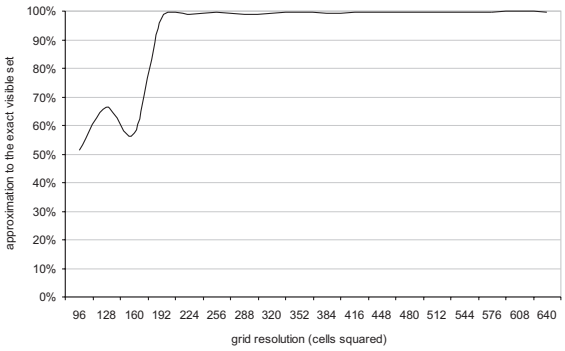

Figure 15: Tightness of approximation of the PVS to the exact result, for different grid resolutions.

diameter $d = \sqrt{-x^2 + r^2}$, where x is the cross-section of the sphere and $x \in [-r, r]$.

- **View-frustum traversal:** The view-frustum traversal is performed by following the light-of-sight discretized in seed-cells by a Bresenham's algorithm for 3D lines. Similar to the 2D case, the traversal is conducted in layers of cells that have the same chess distance to the viewpoint as the seed-cell. This includes more traversal directions than the two directions used for the 2D case.

- **Occluder extension:** When an opaque cell is reached, the occluder extension is used exactly as in the 2D case, with the only difference of using more extension directions. A simple heuristic is proposed by Schaufler *et al.* [22].

- **Occlusion computation:** Since the shadow volumes created are always convex, we can use an occlusion computation scheme similar to that used to discretize objects in the 3D grid. The shadow volume is orthographically projected onto the three coordinate planes such that the cells of $\mathcal{H}$ are classified as occluded only if their projections in each of the planes coincide with pixels of the shadow volume.

Since these extensions were not implemented yet, we need to conduct further research on this topic in order to verify the effectiveness of the proposed suggestions.

7 Conclusion and Future Work

We have introduced an occlusion culling algorithm for densely occluded dynamic scenes based on a regular volumetric grid that represents a discretization of the space and uses opaque regions of the scene as occluders. The benefit of this regular database includes efficient updates of multiple dynamic objects, but also visibility queries at interactive rates due to the optimizations in the stages of discretization of objects and traversal of view-frustum cells.

The proposed algorithm is developed only for scenes that contains closed objects (polyhedra) and it is not appropriate for "polygonal soup" scenes such as forests and particles' compositions. 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 them. Hence, it can be applied in scenes of finely-tessellated geometry and even in non-polygonal scenes.

We have made an implementation of the algorithm for the 2D case and tested it with a mid-range equipment. The timing tests have shown real-time frame rates for most scenes tested (average of 100 fps). For future work, we plan to extend the algorithm to 3D scenes.

References

[1] Narendra Ahuja and Charles Nash. Octree representations of moving objects. *Computer Vision, Graphics, and Image Processing*, 26:207–216, May 1984.

[2] Timo Aila and Ville Miettinen. *dPVS Reference Manual Version 2.10*. Hybrid Holding, Ltd., Helsinki, Finland, October 2001.

[3] John M. Airey, John H. Rohlf, and Jr. Frederick P. Brooks. Towards image realism with interactive update rates in complex virtual building environments. In Rich Riesenfeld and Carlo Sèquin, editors, *Computer Graphics (1990 Symposium on Interactive 3D Graphics)*, volume 24, pages 41–50, March 1990.

[4] Ulf Assarsson and Tomas Möller. Optimized view frustum culling algorithms for bounding boxes. *Journal of Graphics Tools*, 5(1), 2000.

[5] Edwin E. Catmull. Computer display of curved surfaces. In *Proceedings of the IEEE Conference on Computer Graphics, Pattern Recognition, and Data Structure*, pages 11–17, May 1975.

[6] Daniel Cohen-Or, Yiorgos Chrysanthou, Cláudio T. Silva, and George Drettakis. Visibility, problems, techniques and applications. *SIGGRAPH 2000 Course Notes*, July 2000.

[7] Daniel Cohen-Or, Yiorgos Chrysanthou, Cláudio T. Silva, and Frédo Durand. A survey of visibility for walkthrough applications. *SIGGRAPH 2001 Course Notes*, August 2001.

[8] Daniel Cohen-Or, Gadi Fibich, Dan Halperin, and Eyal Zadicario. Conservative visibility and strong occlusion for viewspace partitioning of densely occluded scenes. *Computer Graphics Forum*, 17(3):243–254, 1998.

[9] Satyan Coorg and Seth Teller. Real-time occlusion culling for models with large occluders. In *ACM Symposium on Interactive 3D Graphics*, pages 83–90, April 1997.

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

[11] Frédo Durand, George Drettakis, Joelle Thollot, and Claude Puech. Conservative visibility preprocessing using extended projections. In *Proceedings of SIGGRAPH 2000*, pages 239–248, July 2000.

[12] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley Publishing Co., Reading, MA, 2^{nd} edition, 1990.

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

[14] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical z-buffer visibility, July 1993.

[15] T. Hudson, D. Manocha, J. Cohen, M. Lin, K. Hoff, and H. Zhang. Accelerated occlusion culling using shadow frusta. In *Proceedings of the 13th Annual ACM Symposium on Computational Geometry*, pages 1–10, 1997.

[16] Subodh Kumar, Dinesh Manocha, Bill Garrett, and Ming Lin. Hierarchical back-face computation. In *7th Eurographics Workshop on Rendering*, pages 235–244, 1996.

[17] Don Libes. Modeling dynamic surfaces with octrees. *Computers & Graphics*, 15(3), 1991. Pergamon Press, New York, NY.

[18] David Luebke and Chris 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.

[19] Tomas Möller and Eric Haines. *Real-Time Rendering*. A.K. Peters Ltd., 1999. (Chapter 7).

[20] Harry Plantinga and Charles R. Dyer. Visibility, occlusion and the aspect graph. *International Journal of Computer Vision*, 5(2):137–160, 1990.

[21] Harald Riegler. Point-visibility in computer games. *Seminar "Computer Graphics" Computer Game Technology*, June 2001.

[22] Gernot Schaufler, Julie Dorsey, Xavier Decoret, and François Sillion. Conservative volumetric visibility with occluder fusion. In *Proceedings of SIGGRAPH 2000*, pages 229–238, 2000.

[23] Mel Slater and Yiorgos Chrysanthou. View volume culling using a probabilistic cashing scheme. In S. Wilbur and M. Bergamasco, editors, *Proceedings of Framework for Immersive Virtual Environments FIVE*, December 1996.

[24] Andrew Smith, Yoshifumi Kitamura, and Fumio Kishino. Efficient algorithms for octree motion. In *IAPR Workshop on Machine Vision Applications*, pages 172–177, 1994.

[25] Oded Sudarsky. *Dynamic Scene Occlusion Culling*. PhD thesis, Technion–Israel Institute of Technology, January 1998.

[26] Oded Sudarsky and Craig Gostsman. Output-sensitive visibility algorithms for dynamic scenes with applications to virtual reality. In *Proceedings of Eurographics '96*, volume 15, Poitiers, France, August 1996.

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

[28] Seth J. Teller and Carlo H. Séquin. Visibility preprocessing for interactive walkthroughs. In *Proceedings of SIGGRAPH '91*, volume 25, pages 61–69, July 1991.

[29] Juyang Weng and Narendra Ahuja. Octrees of objects in arbitrary motion: representation and efficiency. *Computer Vision, Graphics, and Image Processing*, 39(2):167–185, 1987.

[30] Peter Wonka and Dieter Schmalstieg. Occluder shadows for fast walkthroughs of urban environments. In *Proceedings of Eurographics '99*, August 1999.

[31] Hansong Zhang. *Effective Occlusion Culling for the Interactive Display of Arbitrary Models*. PhD thesis, Department of Computer Science, UNC Chapel Hill, 1998.

[32] Hansong Zhang and Kenneth E. Hoff III. Fast backface culling using normal masks. In *1997 Symposium on Interactive 3D Graphics*, pages 103–106, April 1997.

[33] Hansong Zhang, Dinesh Manocha, Thomas Hudson, and Kenneth E. Hoff III. Visibility culling using hierarchical occlusion maps. In *Proceedings of SIGGRAPH '97*, volume 31, pages 77–88, August 1997.