EUROGRAPHICS '2000 / M. Gross and F.R.A. Hopgood
(Guest Editors)

Volume 19, (*2000*), *Number 3*

Directional Discretized Occluders for Accelerated Occlusion Culling

Fausto Bernardini[†] Jihad El-Sana[‡] James T. Klosowski[§]

IBM T. J. Watson Research Center

Abstract

We present a technique for accelerating the rendering of high depth-complexity scenes. In a preprocessing stage, we approximate the input model with a hierarchical data structure and compute simple view-dependent polygonal occluders to replace the complex input geometry in subsequent visibility queries. When the user is inspecting and visualizing the input model, the computed occluders are used to avoid rendering geometry which cannot be seen. Our method has several advantages which allow it to perform conservative visibility queries efficiently and it does not require any special graphics hardware. The preprocessing step of our approach can also be used within the framework of other visibility culling methods which need to pre-select or pre-render occluders.
In this paper, we describe our technique and its implementation in detail, and provide experimental evidence of its performance. In addition, we briefly discuss possible extensions of our algorithm.

1. Introduction

Interactive visualization of large geometric data sets with high depth complexity has long been a major problem within computer graphics. Visualizing such data sets, for example computer-aided design (CAD) models, places an enormous burden upon today's graphics hardware, even for high-end systems, due to the large number of polygons in each model. Although graphics hardware has improved greatly, advances have not been able to keep up with the rapid growth of model complexity. In fact, it is no longer unusual to have data sets whose sizes are on the order of tens or hundreds of millions of polygons.

Consequently, many methods have been developed to reduce the number of polygons that need to be sent down the graphics pipeline for rendering. Very often simplification or levels-of-detail will be used to provide a tradeoff between the accuracy of the rendered image and the level of interactivity provided to the user. There have also been many conservative approaches taken, which guarantee that the rendered image is identical to that which would result if all of the geometry were rendered. These techniques attempt to reduce, or *cull*, the number of polygons that need to be rendered by not drawing polygons that cannot be seen from the current viewpoint.

One of the most common examples is *view frustum culling*, which does not render any polygons which lie completely outside of the viewer's field of view. Rather than performing this test individually on a per-polygon basis, object-space hierarchies are often constructed upon the input models so that large pieces of data can be rejected with minimal overhead. Similarly, if solid objects are being used, those polygons that are facing away from the viewpoint, *i.e.*, the *back-facing* polygons, can also be safely ignored. Even greater savings in the number of polygons to be processed by the rendering engine can be achieved by culling away those portions of the geometry which are hidden from the viewer by other geometry. This type of culling is the subject of this paper and is referred to as *occlusion culling*.

Figure 1 is a simple two-dimensional illustration of the problem of occlusion culling in computer graphics. The

[†] Visual Technologies, IBM T. J. Watson Research Center, PO Box 704, Yorktown Heights, NY 10598; fausto@watson.ibm.com.

[‡] Department of Mathematics and Computer Science, Ben-Gurion University; el-sana@cs.bgu.ac.it. This research was conducted during a summer internship at the IBM T. J. Watson Research Center.

[§] Visual Technologies, IBM T. J. Watson Research Center, PO Box 704, Yorktown Heights, NY 10598; jklosow@us.ibm.com.

viewpoint with respect to the scene is represented by the camera icon. The view frustum (dotted lines) represents the field of view of the viewer, so that only objects within this region need to be considered for rendering. In this particular case, objects *A* and *B* together act as occluders of object *C*. It is important to note that by themselves, neither object *A* nor *B* occludes object *C*; only by considering both objects together can object *C* be found to be occluded. By avoiding rendering occluded objects, a graphics engine can potentially render more quickly, and as a result, larger scenes can be processed at interactive rates.

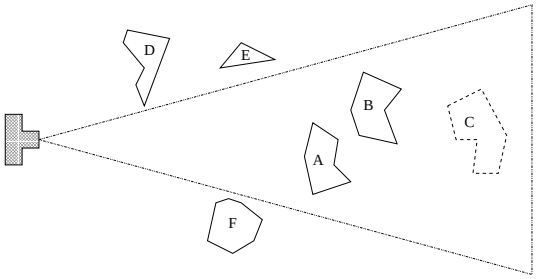

Figure 1: *The occlusion culling problem. Given a viewpoint (camera), a view frustum, and a set of objects, the goal is to determine which objects are occluded by other objects. In this case, object C is hidden from the viewer by objects A and B and therefore does not need to be rendered.*

Main Contribution

We present a technique for accelerating the rendering of large, high depth complexity scenes. Using a front-to-back order, our goal is to avoid drawing objects which are completely hidden by what has already been rendered. Unfortunately, using the input geometry for this task is extremely time consuming due to the complexity of visibility tests for a large collection of polygons. Our approach is to find very simple polygonal objects which can take the place of the complex collection of geometry and can be used to answer these questions very simply.

Our *Directional Discretized Occluders* (DDOs) approach involves a preprocessing phase. During this phase, we build an object-space hierarchical data structure (an octree in our current implementation). The actual geometry is stored with the leaf nodes of the octree, and nodes are recursively subdivided so that a bounded number of primitives per leaf node is obtained. During preprocessing, each face of the octree is regarded as a potential occluder. The two halfspaces on the two sides of the face are partitioned into regions. For each region, we compute and store a flag that records whether that octree face is a valid occluder for a viewpoint contained in that region. Each octree face is now a view-dependent polygonal occluder that can be used in place of the complex geometry in subsequent visibility queries.

When the user is inspecting and visualizing the input model, we utilize the occluders found in preprocessing to avoid rendering geometry which cannot be seen from the current viewpoint. Our rendering algorithm visits the octree data structure in a top-down, front-to-back order. Valid occluders found during the traversal are projected and added to a 2-dimensional data structure. Each node is first tested against the current collection of occluders: if the node is not visible, traversal of its subtree stops. Otherwise, recursion continues and if a visible leaf node is reached, its geometry is rendered.

Advantages

Our method has several advantages. First, our occluders replace arbitrarily complex models with very simple polygons, thereby allowing subsequent conservative visibility queries to be computed quickly. Our method is especially effective when the scene is comprised of many small polygons, for example when approximating curved surfaces such as the body of an automobile. Previous methods would fail to find a small collection of polygons that occlude a significant portion of geometry, or would have to take into account a large number of polygons, thereby slowing down the culling process and reducing the overall frame rate. Our method will replace the finely tessellated surface with a simpler collection of polygons during preprocessing, allowing the run-time culling algorithm to perform at a higher frame rate.

The fact that all the occluders computed in preprocessing are axis-aligned squares can be exploited to design efficient data-structures for visibility queries. Also notice that storage overhead is minimal: just six 32-bit bitmasks per octree node. The DDO approach benefits from *occluder fusion*, by determining that objects which are only occluded by a combination of input models, as opposed to any one particular model, are not visible from the current viewpoint and should not be rendered. The DDO approach also does not require any special or advanced graphics hardware.

The DDO approach could be used within the framework of other visibility culling methods. Culling methods which need to pre-select large occluders, *e.g.* Coorg and Teller [3], or which pre-render occluders to compute occlusion maps, *e.g.* Zhang, *et al.* [13], can benefit from the DDO preprocessing step, which replaces complex geometry with simple polygonal occluders.

Organization

The remainder of our paper is organized as follows. In Section 2, we survey the related work on visibility culling, and in particular occlusion culling. We introduce our Directional Discretized Occluders approach in Section 3 and our run-time culling algorithm in Section 4. We analyze the performance of the DDO approach in Section 5. Finally, we men-

tion future work areas and present our conclusions in Section 6.

2. Related Work

Due to the importance of the problem of rendering large 3D models at interactive rates, there is a great deal of literature on the subject. The wide range of related visibility culling research typically falls within three categories: object-space, image-space, and domain-specific.

An object-space occlusion culling approach proposed by Coorg and Teller [4] tries to precompute a superset of the visible polygons when the viewpoint is within a particular region of space. As the viewpoint moves, this method keeps track of *visual events* which cause changes in the polygons that can been seen from the current viewpoint. This method considers only individual polygons (or objects) as occluders (as opposed to combining polygons and objects to form larger occluders) and it relies on there being large occluders in the model. Another example of an object-space method by the same authors can be found in [3].

The Hierarchical Z-buffer (HZB) of Greene, *et al.* [7] extends the traditional Z-buffer by maintaining an *image Z pyramid* to quickly reject hidden geometry. A second data structure, an octree, is built upon the geometry in object-space. At run-time, the octree is traversed and each node of the tree is queried against the current HZB. If the current node is determined to be behind the previously seen geometry, then this node can be skipped, together with all of its children and the geometry associated with these nodes. Otherwise, the geometry is rendered, the new z values are propagated through the HZB, and the node's children are visited. Temporal coherence is exploited by prerendering previously visible nodes. The main weakness of the HZB approach is that the queries against the HZB, as well as the updates, assume special graphics hardware to achieve their interactive rates. These ideas were extended in [6].

Bartz *et al.* [1] introduce another technique based on an object-space hierarchy (a *sloppy n*-ary space partitioning tree in this case) and an image-based occlusion test. Their method uses OpenGL to scan-convert bounding volumes into a *virtual occlusion buffer.* This buffer is then sampled to detect changes triggered by nodes containing potentially visible geometry. The sampling density can be adjusted to provide adaptive, non-conservative culling.

Hudson *et al.* [8] use shadow frusta to cull occluded geometry. A hierarchy of bounding volumes is computed for the input geometry. Another volume hierarchy stores a set of precomputed potentially good occluders per cell. During visualization, the algorithm dynamically updates the set of effective occluders (an occluder is a convex polygon or a collection of convex polygons) for the current viewpoint. These occluders define shadow frusta. Geometry contained in bounding volumes entirely contained in a frustum can be culled.

Cohen-Or *et al.* [2] propose a method which partitions the viewspace into cells, and precomputes a superset of the objects visible from a viewpoint in the cell. The objects not included in the list are called *strongly occluded*, *i.e.*, objects for which there exist one occluding convex polygon. They show that for scenes such as city models the number of strongly occluded objects from any given viewpoint is not significantly smaller than the number of occluded objects.

A similar method using an octree rather than a uniform cell partitioning for the viewspace is described in [11].

The Hierarchical Occlusion Maps (HOM) [13] approach is similar to the HZB, and our DDO method, in that it uses hierarchies within both object and image-space. Initially, the image-space HOMs are built by rendering geometry with a high potential for occlusion, *i.e.*, geometry that is very close to the current viewpoint. To determine whether the remaining geometry is visible, the object-space hierarchy is traversed and each node is checked against the HOM, using, if necessary, a conservative depth test to determine occlusion. This technique supports occluder fusion and culling of object that contribute only few pixels to the scene. Drawbacks to this approach are that it must select occluders for each viewpoint to initially render into the HOM, and special graphics hardware which supports bilinear interpolation of texture maps is required to accelerate the HOM construction.

One example of domain-specific occlusion culling technique is targeted to walkthroughs of architectural models (Teller and Sequin [12]). Such models can naturally be broken down into *cells* and *portals*. Cells have boundaries which coincide with opaque surfaces, while portals represent non-opaque surfaces between the cells. An adjacency graph, which connects the cells via portals, is constructed and then utilized to determine the cell-to-cell visibility and the potentially visible set (PVS) of polygons for each cell. The memory consumption of this approach in the worst case can be proportional to the number of cells squared.

Another class of algorithms has the main goal of achieving a constant frame rate, by allowing a fixed number of polygons to be rendered at each frame (*polygon budget*). These algorithms tend to cull more aggressively, at the expense of image accuracy, to achieve their frame rate goal. An example of such methods is [9].

3. Directional Discretized Occluders

As we have discussed earlier, directly using the input geometry for occlusion culling is extremely time consuming due to the complexity of visibility tests for a large collection of polygons. Our solution to this problem is to find very simple polygonal objects which can take the place of the complex

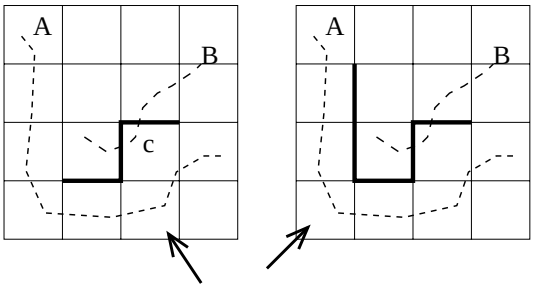

Figure 2: *Illustration of the DDO approach. The viewpoint and view direction are specified using arrows in each figure. The input geometry, A and B, is drawn as dashed lines and can be replaced by simple polygonal occluders, the grid edges, for subsequent visibility queries. The valid occluders for this example are shown as thick solid lines.*

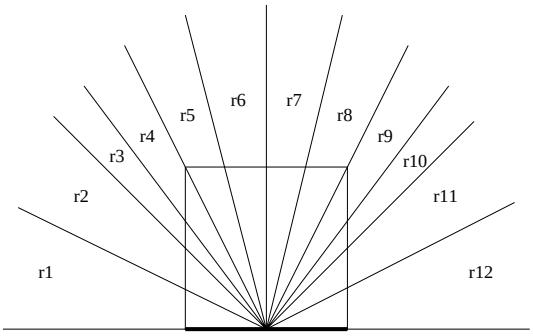

Figure 3: *Illustration of the view direction discretization. For an octree face, shown as a thick solid line, of an octree node, we partition the set of feasible viewing directions into a discrete number of viewspace regions. In this example, twelve viewspace regions are considered, labelled $r1$ to $r12$.*

geometry and can be used to answer these visibility questions very efficiently.

We have chosen to use *Directional Discretized Occluders* (DDOs) as our polygonal objects. DDOs get their name from the fact that the occluders are view dependent. For each polygonal object that we consider as a potential occluder, we have discretized the set of all possible view directions from which this object can be seen. For each of these discrete sets of directions, we then test whether the object is a *valid* occluder, meaning that all of the geometry that is hidden by the occluder, from the current viewpoint, is also hidden by the actual input geometry. Thus, an object can be a valid occluder from as few as one of these discrete sets of directions, or as many as all of them. We discuss this discretization of viewing directions in more detail shortly.

Our DDOs are computed, during a preprocessing stage, using an object-space hierarchical data structure as an approximation to the input. We have chosen to use an octree in our present work, although many alternative hierarchies could also be chosen, such as a k-d tree or a hierarchy of axis-aligned bounding boxes. The potential occluders that we have chosen to consider are the faces of the octree nodes.

Figure 2 is a two-dimensional illustration of the DDO approach. The grid is a discretization of the space surrounding the scene; it represents our octree nodes. The input geometry, *A* and *B*, is shown using dashed lines. For the purpose of occlusion culling, the geometry *A* can be replaced by a simpler object, shown using thick solid lines, which is a subset of the grid edges, *i.e.* the octree faces. The two figures show the same scene from different viewpoints and view directions, as indicated by the arrows. Note that the subset of grid edges that can act as occluders (in place of geometry *A*) changes as the viewpoint changes. Although geometry *A* occludes geometry *B* from each of these two viewpoints, the occluders which replace geometry *A* will help us determine only that geometry *B* is occluded in the figure on the right. The region of geometry *B* that is still visible to the viewer with respect to the occluders is contained in octree node *c*. In this example, for illustrative purposes, we did not construct a very deep octree; however, in practice, each of the octree nodes shown here, and in particular node *c*, would be further subdivided. The more refined subdivision provides additional occluders that we can use which would then indicate that geometry *B* is in fact occluded from the current viewpoint.

Each octree face is a candidate occluder only when seen from the halfspace containing the octree node to which it belongs. (Each face is considered twice, once for each of the adjacent octree nodes.) Clearly, the geometry within the octree node, and nearby nodes, can occlude the octree face only when viewed from the appropriate side. We partition the solid angle spanning the halfspace into a discrete number of *viewspace regions*, hence the name of our culling approach: *directional discretized occluders*. Figure 3 shows such a discretization of viewing directions with respect to one octree face, shown by a thick solid line. In this example, we have divided the viewing directions into twelve regions, labelled $r1$ to $r12$. The viewspace regions should not contain viewpoints located within (or nearby) the octree node itself, as this would greatly complicate the validity test. We have therefore introduced a "clipping plane" beyond which all viewpoints must be located. That is, only when the viewpoint is behind this clipping plane can the octree face be a potential occluder. See Figure 4 for an illustration.

To test whether an octree face is a valid occluder for each viewspace region, we consider the "viewspace frustum", the polyhedron obtained as a convolution of the octree face with the viewspace region. The viewspace frustum contains all possible rays from a potential viewpoint in the region to every point on the octree face. We determine if there exists a connected collection of polygons, in the geometry

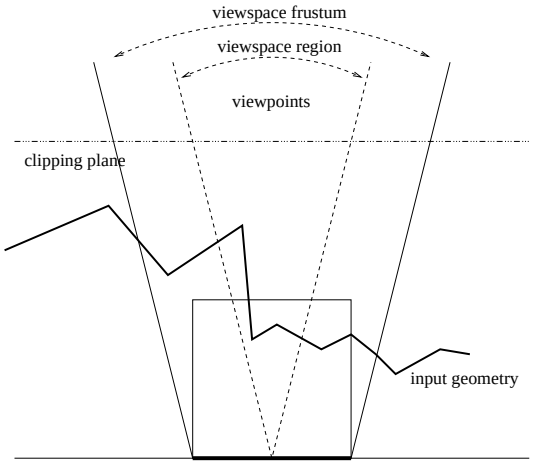

Figure 4: *Illustration of the viewspace frustum. For each face, shown as a thick solid line, of each octree node, we determine whether the octree face is a valid occluder for a discrete number of viewspace regions. To be a valid occluder, the input geometry must cut the viewspace frustum, which is the set of all possible rays from the potential viewpoints in the viewspace region to the octree face. The potential viewpoints must lie beyond the clipping plane, shown here as a horizontal line.*

within and nearby the node, that intersects every possible ray, or, equivalently, that "cuts" the viewspace frustum into two parts, one containing the face, the other containing all potential viewpoints within the region. Figure 4 shows the classification of a face of an octree node as a valid occluder for a given viewspace region. Details on this test are provided below in Section 3.2.

For each face of each node of the octree, we store an occlusion bitmask which has one bit for each of the viewspace regions. If the face is a valid occluder for viewspace region i, we set the value of bit i to one, otherwise we set the bit to zero. During the rendering process, we use the bitmask to determine whether a given octree face is a valid occluder for the current viewing parameters.

Deciding how many viewspace regions to create is an interesting question. If we have only a few large regions, the viewspace frusta are also very large and the likelihood that the nearby geometry will completely cut the frustum goes down. On the other hand, having very small regions is also problematic in that the preprocessing time will increase proportionally with the number of regions that we must check for occlusion. Since we are using bitmasks to store the DDO information for each face of each octree node, we have experimented with 32 and 64 viewspace regions, for quickly accessing the information and for efficient memory usage.

```
Algorithm OctreeConstruction(octree node c)
 1. for each face f of c
 2.     for each viewspace region r of face f
 3.         Look-up viewspace frustum v;
 4.         if (geometry cuts v) then
 5.             Set appropriate bit of f's bitmask to 1;
 6.         else
 7.             Set appropriate bit of f's bitmask to 0;
 8. if (# polygons in c > threshold) then
 9.     Subdivide c into 8 children;
10.     Partition c's polygons into the 8 new nodes;
11.     for each new child c' of node c
12.         OctreeConstruction(c');
```

Figure 5: *Pseudo-code for the octree construction algorithm. As the octree is built, we test each face of each node to determine if it is a valid occluder for each of the viewspace regions. For each node c of the octree, we maintain an occlusion bitmask to record the occlusion information.*

3.1. DDO Construction

The DDO construction process builds an octree data structure based on the input model's geometry. At the same time, the process determines for each of the six faces of each of the octree nodes whether it can be used as an occluder for the discrete set of viewspace regions.

The octree is initialized by computing the axis-aligned bounding box of the input geometry, defining the root node, and assigning each input polygon to the root node. The octree construction is a recursive process which works on a single node, referred to as the *current node*, at any one time.

Figure 5 contains the pseudo-code for the octree and DDO construction algorithm. Initially, the algorithm begins with the root node of the octree as input. For each face f of the current node, we consider each of the viewspace regions r which are defined for that face. Given the face f and viewspace region r, we consider the viewspace frustum v, which is the convolution of f and r. (These frusta are precomputed and stored in a look-up table.) We then must test whether geometry within (and nearby) the current node cuts v into the two regions described above: one containing the face f and the other containing the set of potential viewpoints within r. If v is cut, we set the corresponding bit in the bitmask to be one, otherwise we set it to zero. Once we have performed these tests for all of the faces of the octree node, we consider the number of polygons assigned to the current node. If the number is not greater than a predefined threshold, the current node is considered a leaf of the octree. Otherwise, the current node is spatially subdivided into eight children, the polygons of the current node are partitioned among the eight children, and the *OctreeConstruction* process is recursively called for each of the eight children of the current node.

The occlusion information computed in the preprocessing phase can be used directly by a run-time rendering process or saved to a DDO file which can be read and used at a later time. The DDO file contains a description of the octree hierarchy and for each face, it stores the occlusion bitmask which indicates the viewspace regions for which the face is a valid occluder.

Figures 9 and 10 illustrate the results of our DDO preprocessing step for an automobile model containing 162K polygons. In each of the figures on the left, the valid DDOs are shown, in one case with the model, drawn in red wireframe, and the other without. The right images in Figures 9 and 10 show the final 2D image-space hierarchies for the current viewpoint.

3.2. DDO Validation

For each viewspace region of an octree face, we need to determine whether the octree face is a valid occluder for the set of all potential viewpoints within the viewspace region. Recall that the set of potential viewpoints cannot reside within the octree node itself, they must reside beyond a clipping plane, as illustrated in Figure 4.

To test the validity of a potential occluder, we consider the viewspace frustum, which contains all of the possible rays from a potential viewpoint in a viewspace region to every point on the octree face. We need to determine if there exists a connected collection of polygons, in the geometry within and nearby the octree node, that intersects every possible ray.

Figure 6 contains the pseudo-code for our current DDO validation algorithm. The algorithm begins by finding an unvisited seed polygon, s, located within the octree node. A first-in-first-out queue Q is used to maintain the list of visited polygons; it is initialized to be the empty set. Polygon s is then marked as visited and inserted into Q. While Q is non-empty, we remove the polygon p from the head of Q, and check if p intersects the octree face. If it does, we conservatively conclude that the polygons within the octree cell cannot serve as a valid occluder and we terminate the test. While it is still possible that the geometry would cut the viewspace frustum as desired, verifying this result becomes extremely complicated and inefficient. A similar test is performed to check polygon p and the clipping plane.

If the polygon passes each of the previous two checks, we consider each of the edges e of p. If the current edge e is within the viewspace frustum, we consider the neighbor of polygon p along edge e. If there is no neighbor along edge e, then we cannot cut the viewspace frustum and therefore we must restart our test with another seed polygon in the octree node. If there is a neighboring polygon p' and it is unmarked, we mark it as visited and insert it into Q. If the current edge e is not within the viewspace frustum, we mark it as a boundary edge. Once we have tested all of the edges of polygon p in this manner, we continue removing polygons from Q until it is empty or we have returned a negative result (invalid occluder). If we reach the point that Q is empty, we then test if the visited geometry will cut the viewspace frustum. To do this, we project the boundary edges onto a plane through the center of the octree face, which is perpendicular to the axis of the viewspace frustum. If the projected boundary contains the center of the occluder, we know that the octree face is a valid occluder, otherwise it is not.

```
Algorithm DDO Validation(octree node c)
 1. if (no unmarked polygon s in c) then
 2.     goto Step 20;
 3. Initialize queue Q to be empty;
 4. Mark polygon s and insert into Q;
 5. while (Q != ∅)
 6.     Extract polygon p from head of Q;
 7.     if (p intersects octree face) then
 8.         return invalid occluder;
 9.     if (p intersects clipping plane) then
10.         return invalid occluder;
11.     for each edge e of p
12.         if (e intersects viewspace frustum) then
13.             if (∃ neighbor polygon p' along e) then
14.                 if (p' is not marked) then
15.                     Mark p' and insert into Q;
16.             else
17.                 return to Step 1;
18.         else
19.             Mark e as a boundary edge;
20. if (projection of boundary edges contains
                                  center of occluder) then
21.     return valid occluder;
22. else
23.     return invalid occluder;
```

Figure 6: *Pseudo-code for the DDO validation algorithm. To determine if an octree face is a valid occluder for a given viewspace region, we check if there exists a connected collection of polygons, in the geometry within and nearby the octree node, that intersects every possible ray, or, equivalently, that cuts the viewspace frustum into two parts, one containing the octree face, and the other containing all potential viewpoints within the viewspace region.*

It is important to note that the faces of the octree act as *conservative occluders*, *i.e.*, they occlude a subset of what is occluded by the actual input geometry. This property guarantees that culling the geometry occluded by the octree faces will not introduce visibility artifacts in the rendered scene. On the other hand, the fact that only a subset of the occluded geometry will be culled is usually a small price to pay for the computational speed-up obtained.

```
Algorithm OctreeTraversal(octree node c)
1.  if (c is outside view frustum) then
2.      return;
3.  if (c is occluded by occlusion map) then
4.      return;
5.  if (c is a leaf node) then
6.      Render c's geometry;
7.  else
8.      /* in front-to-back order */
9.      for each child c' of c
10.         OctreeTraversal(c');
11. Insert c's occluders into occlusion map;
```

Figure 7: *Pseudo-code for the octree traversal algorithm. As indicated in step 8, the children of node c must be visited in front-to-back order.*

4. Occlusion Culling using DDOs

Once the DDO preprocessing step is completed, our goal is to render as little of the model's geometry as possible while producing an image on the screen which is identical to the image we would get if we simply rendered all of the geometry.

4.1. Octree Traversal Algorithm

To visualize the input models, for a given viewpoint and view direction, we begin by traversing our octree in a top-down, front-to-back order. To correctly produce the final image, it is vital to traverse the octree in this manner. During the run-time visualization of the input geometry, we utilize a two-dimensional image-space hierarchy, described below, to maintain the set of valid occluders seen up to the current point of the traversal algorithm. Figure 7 highlights the pseudo-code used by our *OctreeTraversal* algorithm.

During the traversal, each node of the octree is first tested to see if the node lies outside the view frustum. If it does, we no longer need to consider the subtree rooted at this node of the octree, since it is not visible to the viewer. If the node intersects the view frustum, we then determine (see below) whether it is occluded using the current set of valid occluders added up to this point of the traversal. If the node is occluded, then all of the polygons of its subtree cannot be seen from the current viewpoint, and we need not consider this node any further. If the node is not occluded and it is a leaf, the polygons that are assigned to this node are rendered, and the valid occluders, as determined by the occlusion bitmask for the current node, are added to the image-space hierarchy. If the node is not a leaf, we recursively call the *OctreeTraversal* algorithm for each of the children of the current node, in front-to-back order, based upon the current viewpoint. For efficiency, we use a look-up-table to quickly determine this order. Upon completion of the recursion for each of the non-leaf nodes, the valid occluders for those nodes can be added to the image-space hierarchy.

4.2. Occlusion Maps

For each viewpoint, we maintain the current set of valid occluders in a two-dimensional image-space hierarchical data structure, which we also refer to as an *occlusion map*. Notice that for the purpose of determining occlusion, we can use an image plane that is parallel to one of the coordinate planes [5]. Which of the coordinate planes to use is determined by the current viewing parameters. In the current implementation, we choose the plane which maximizes the area of projection based upon the current viewpoint and view direction. Projecting occluders onto the image plane produces axis-aligned rectangles. Visibility queries therefore reduce to checking an axis-aligned rectangle (bounding the projection of the current node) against a collection of axis-aligned rectangles.

Testing whether a node of the octree is occluded by the current occlusion map begins by projecting the vertices of the node onto the image plane. The bounding box of the node's projection is then compared against the occluders currently stored in the occlusion map. If the bounding box is completely contained within the union of occluders, then it is not visible from the current viewpoint.

The problem of maintaining and querying a collection of axis-aligned rectangles has been well-studied in the field of computational geometry [10]. In the current implementation, we have tested three data structures for this task: a traditional quadtree, a uniform quadtree, and a hierarchy of uniform grids. The quadtree represents the union of rectangles exactly, while the other two data structures are conservative discretized approximations. While these discretizations may hinder our approach in that we will not conclude that a node is occluded when in fact it is, the efficiencies gained when querying and maintaining these data structures greatly outweighs any such hindrances.

5. Implementation and Analysis

We have implemented our method entirely in software (C++) and tested it upon an IBM RS/6000 595 with a GXT800 graphics adapter. We have found that our method is especially effective when the scene is comprised of many small polygons, for example when approximating curved surfaces such as the body of a automobile. Our method replaces the finely tessellated surface with a simpler collection of octree faces during the DDO preprocessing stage, allowing the run-time culling algorithm to perform at higher frame rates. Complex data sets of this sort are commonplace in the manufacturing industry where scanned models are often oversampled.

We have run the majority of our tests using an automobile model, containing 162K polygons, comprised of the

body, engine, and interior sections. Figure 8 demonstrates the speed-up obtained by using our DDOs for occlusion culling. In the best case, we see a speed-up of two times, and in the worst case, for a few frames of our inspection of the model, the DDO method is actually slower than not using occlusion culling at all. This occurred when the camera was very close to the automobile and essentially all of the occluders were invalidated. Thus, the DDO approach had the overhead of performing the visibility queries, which did not cull any of the octree nodes, and still had to render (essentially) the same geometry as not using occlusion culling. On average, the speed-up for this inspection was 1.4. We expect that for larger models with higher depth complexity, our method will provide better results. An improved implementation and additional experiments are currently underway. The preprocessing requirement for our method is not inexpensive. For the automobile data set, the DDO construction algorithm required over one hour of processing time.

5.1. Additional Uses of the DDOs

Once the DDO preprocessing step is completed, numerous approaches can be taken for utilizing the DDOs. The computed DDOs could, in fact, be used by many existing visibility culling approaches that pre-select large occluders or that pre-render occluders to compute an occlusion map. For example, the recent Hierarchical Occlusion Maps (HOM) technique of Zhang, *et. al* [13] renders a subset of the input geometry to construct their occlusion maps. To speed up this procedure, the HOM method preselects these subsets of geometry for various viewpoint locations, and then uses a simple look-up-table to determine which geometry should be rendered to construct the HOMs based upon the current viewpoint. However, one problem that remains is that the complex input geometry is still being used for this purpose. By utilizing the results of our preprocessing step, the DDO file, the same look-up-table could be used to quickly determine which geometry should be rendered, but in this case, the geometry is now much simpler and faster to render.

Other visibility culling approaches which could benefit from having the DDO information are the techniques of Coorg and Teller [3] and Hudson, *et. al* [8]. Each of these techniques finds hidden geometry by using a small subset of the input geometry as occluders and computing the *shadow frusta*. For each of the occluders, they compute the region of the scene which is in the shadow of the polygon with respect to the current viewpoint. Once again, these methods suffer from having to use the input geometry for their choice of occluders. This is problematic in that if they only select a few occluders, the amount of culling is not significant. On the other hand, if many occluders are selected, the shadow frusta computations become too time consuming. Another issue that must be resolved in these approaches is the problem of selecting which of the input polygons are good occluders for the given viewpoint. By utilizing the preprocessing step of our DDO approach, these issues are efficiently resolved.

6. Conclusion and Future Work

We have presented an occlusion culling technique for accelerating the rendering of high depth complexity scenes. The main contribution of our work are our Directional Discretized Occluders (DDOs). In a preprocessing stage, we approximated the input model with a hierarchical data structure and computed simple view-dependent polygonal occluders, our DDOs, to replace the complex input geometry in subsequent visibility queries.

Our method has several advantages which allow it to perform conservative visibility queries efficiently and it does not require any special graphics hardware. A major benefit of our approach is that the preprocessing step can also be used within the framework of other visibility culling methods which need to pre-select or pre-render occluders.

Future research areas include improving the DDO construction algorithm. Currently, the preprocessing stage can be fairly time consuming. We are investigating alternative construction schemes, including some image-based methods for accelerating the DDO validation stage of our algorithm.

7. Acknowledgements

We would like to thank David Luebke and Jai Menon for useful discussions during the initial stages of this work.

References

1. D. Bartz, M. Meißner, and T. Hüttner. OpenGL-assisted occlusion culling for large polygonal models *Computer & Graphics*, 23(5):667–679 , 1999.

2. D. Cohen-Or, G. Fibich, D. Halperin, and E. Zadicario. Conservative visibility and strong occlusion for viewspace partitioning of densely occluded scenes. *Computer Graphics Forum*, 17(3):243–253, 1998.

3. S. Coorg and S. Teller. Real-time occlusion culling for models with large occluders. In *Proc. 1997 Sympos. Interactive 3D Graphics*, 1997.

4. S. Coorg and S. Teller. Temporally coherent conservative visibility. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 78–87, 1996.

5. J. E. Goodman and J. O'Rourke, Editors. *Handbook of Discrete and Computational Geometry*. CRC Press LLC, Boca Raton, FL, 1997.

6. N. Greene. Hierarchical polygon tiling with coverage masks. In *SIGGRAPH 96 Conference Proceedings*, pages 65–74, Aug. 1996.

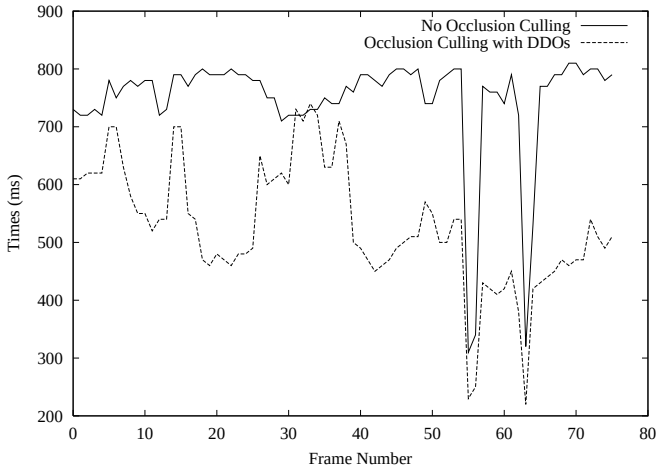

Figure 8: *This plot demonstrates the speed-up obtained by using our DDOs for occlusion culling. For an automobile model containing 162K polygons, we see a speed-up of two times in the best case. For a few frames of our inspection of the model, the DDO method is actually slower than not using occlusion culling at all. This occurred when the camera was very close to the automobile and essentially all of the occluders were invalidated.*

7. N. Greene, M. Kass, and G. Miller. Hierarchical Z-buffer visibility. In *SIGGRAPH 93 Conference Proceedings*, pages 231–240, Aug. 1993.

8. T. Hudson, D. Manocha, J. Cohen, M. Lin, K. Hoff, and H. Zhang. Accelerated occlusion culling using shadow frusta. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 1–10, 1997.

9. J. Klosowski and C. Silva. Rendering on a Budget: A Framework for Time-Critical Rendering. In *Proc. IEEE Visualization*, pages 115–122, 1999.

10. F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 3rd. ed., Oct. 1990.

11. C. Saona-Vázquez, I. Navazo, and P. Brunet. The visibility octree: a data structure for 3D navigation. *Computer & Graphics*, 23(5):635–643, 1999.

12. S. J. Teller and C. H. Séquin. Visibility preprocessing for interactive walkthroughs. In Thomas W. Sederberg, editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 61–69, July 1991.

13. H. Zhang, D. Manocha, T. Hudson, and K. E. Hoff III. Visibility culling using hierarchical occlusion maps. In *SIGGRAPH 97 Conference Proceedings*, pages 77–88, Aug. 1997.

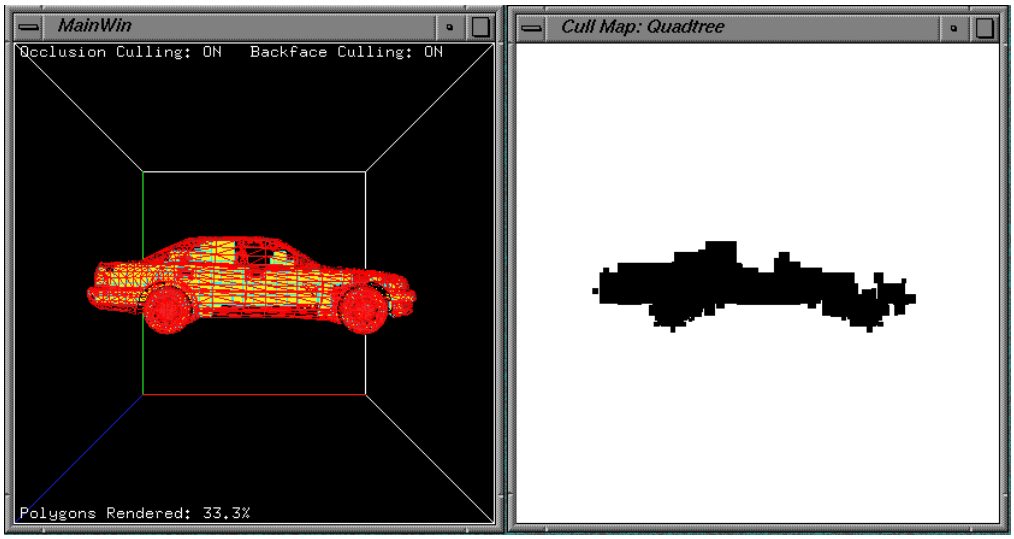

Figure 9: *Illustration of three-dimensional DDOs computed for an automobile model. In the left figure, the model is drawn in red wireframe to allow the valid occluders for this viewpoint to be seen. In the right figure, we show the corresponding two-dimensional occlusion map.*

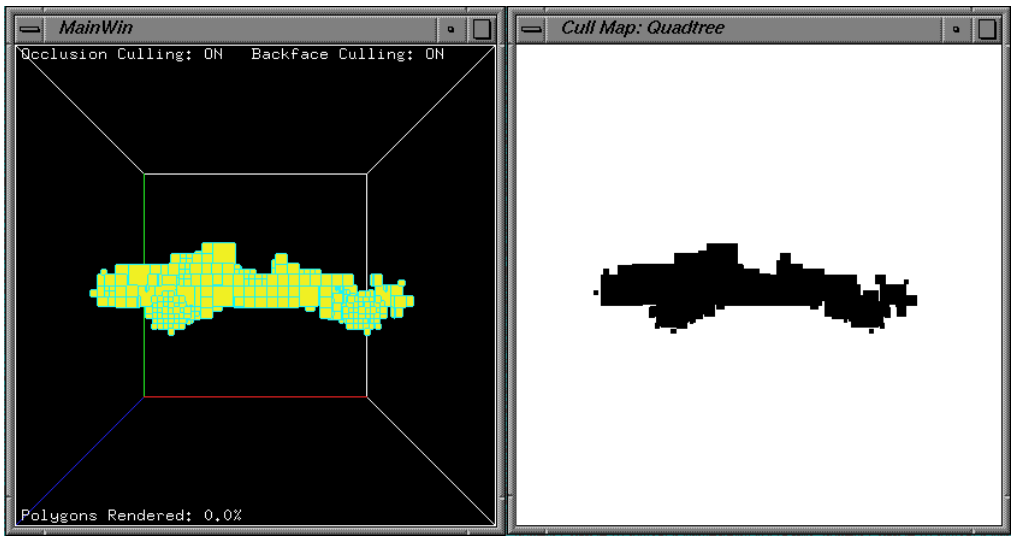

Figure 10: *Illustration of three-dimensional DDOs computed for an automobile model. In the left figure, the model is removed to allow the valid occluders for this viewpoint to be clearly seen. Again, in the right figure, we show the corresponding two-dimensional occlusion map.*