Accelerated Occlusion Culling using Shadow Frusta *

T. Hudson D. Manocha J. Cohen M. Lin† K. Hoff H. Zhang
Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175
{hudson,manocha,cohenj,lin,hoff,zhangh}@cs.unc.edu

Abstract:

Many applications in computer graphics and virtual environments need to render datasets with large numbers of primitives and high depth complexity at interactive rates. However, standard techniques like view frustum culling and a hardware z-buffer are unable to display datasets composed of hundred of thousands of polygons at interactive frame rates on current high-end graphics systems. We add a "conservative" visibility culling stage to the rendering pipeline, attempting to identify and avoid processing of occluded polygons. Given a moving viewpoint, the algorithm dynamically chooses a set of *occluders*. Each occluder is used to compute a *shadow frustum*, and all primitives contained within this frustum are culled. The algorithm hierarchically traverses the model, culling out parts not visible from the current viewpoint using efficient, robust, and in some cases specialized interference detection algorithms. The algorithm's performance varies with the location of the viewpoint and the depth complexity of the model. In the worst case it is *linear* in the input size with a small constant. In this paper, we demonstrate its performance on a city model composed of 500,000 polygons and possessing varying depth complexity. We are able to cull an average of 55% of the polygons that would not be culled by view-frustum culling and obtain a commensurate improvement in frame rate. The overall approach is *effective* and *scalable*, is applicable to all polygonal models, and can be easily implemented on top of view-frustum culling.

1 Introduction

Interactive display of extremely large and complex geometric data sets has long been an important problem in computer graphics. Although throughput of graphics systems has improved considerably over the years, the size and complexity of models has grown even faster. In many walkthrough and virtual environment applications, models commonly consist of millions of primitives. Rendering such models at interactive rates is a major challenge. There is a great body of literature in computer graphics and computational geometry using techniques based on visibility culling and model simplification to render such large models. In this paper, we address visibility culling.

Given a large model and a viewpoint, the goal of visibility culling and hidden surface removal algorithms is to determine the set of primitives visible from that viewpoint. No general purpose, interactive algorithms are known for *exact* visibility determination on large models composed of hundred of thousands of polygons. Current graphics hardware provides support for visibility computations on a per-pixel basis with a z-buffer. However, high-end systems are only able to render a few tens of thousands of polygons at reasonable frame rates, due to bottlenecks in their hardware before the per-pixel analyses can be carried out. As a result, many applications use the following high-level software techniques to cull away a subset of the polygons which are not visible from the current viewpoint before they are needlessly sent to the rendering hardware:

- **View-Frustum Culling:** View-frustum culling uses a traversal of spatial data structures to cull out portions of the model not lying in the current *view frustum*. The viewing volume is represented as a frustum of six planes (near plane, far plane, and four sides). At runtime the display algorithm checks whether each node of the spatial structure overlaps this frustum; only nodes partially or completely contained within the frustum are rendered.
- **Occlusion Culling:** Hidden-surface removal algorithms and occlusion culling techniques are commonly used for models with high depth complexity to further reject the portion of geometry obscured by other objects in the scene. Some of the existing techniques are based on backface culling, binary space partition trees, or partitioning the models into cells and portals.

Desiderata: In designing algorithms for occlusion culling, we recognize the importance of *generality* and *robustness*. We make no assumptions about the structure of our input models – they may be arbitrary sets of polygons with no other topological information, also known as "polygon soup". The problem of 3D occlusion culling involves the computation of some geometric relationship between two or more objects. In our case, we reduce the problem to performing overlap tests between convex objects in 2D or 3D. Based on our experience in developing two interference detection systems, I-COLLIDE [CLMP95] and RAPID [GLM96], as well as that of other authors in implementing algorithms for interference detection [HKM95, BCG+96] and intersection computation for solid modeling [For96, HHK89], we have realized robustness is an important issue in the design and implementation of interference detection algorithms. Our goal is to develop algorithms which are relatively simple, efficient, and not prone to geometric degeneracies.

In this paper, we present object-space techniques for *occlusion culling*. This involves computation of a set of polygons that are within the view frustum but are not visible from the current viewpoint. We add this conservative visibility culling stage to the

*Supported in part by a Sloan fellowship, ARO Contract P-34982-MA, NSF grant CCR-9319957, NSF grant CCR-9625217, ONR Young Investigator Award, DARPA contract DABT63-93-C-0048, NSF/ARPA Science and Technology Center for Computer Graphics & Scientific Visualization NSF Prime contract No. 8920219.

†Also with U.S. Army Research Office

To appear in the proceedings of the Thirteenth ACM Symposium on Computational Geometry, June 1997, Nice, France.

rendering pipeline in order to reduce the number of polygons sent to the graphics hardware.

As the viewpoint changes, the algorithm dynamically chooses a set of occluders. Each occluder is a convex polytope or a union of convex polytopes. For each occluder the algorithm computes a *shadow frustum* and uses fast interference detection and a hierarchical representation to find those portions of the model within the shadow frustum. The idea of shadow volumes was first introduced by Crow [Cro77] to generate shadows by creating for each object a shadow volume that the object blocks from the light source.

Main contribution: We present geometric algorithms for

1. Occluder selection using off-line and on-line techniques
2. Robust and efficient occlusion culling based on specialized interference detection algorithms, given the occluders and a hierarchical decomposition of scene geometry into bounding volumes.

The resulting algorithms have been implemented and we report their performance on a large model.

Organization: The rest of the paper is organized in the following manner. We survey related work in Section 2 and give an overview of the algorithm in Section 3. The occluder selection algorithm is presented in Section 4 and visibility culling based on occluders is described in Section 5. We present implementation details and the performance in Section 6 and analyse its complexity in Section 7.

2 Related Work

There has been significant amount of research on visibility and hidden surface removal in the fields of computer graphics and computational geometry. Many asymptotically efficient algorithms have been proposed for exact visibility and hidden surface removal [SSS74, Mul89, BDEG94, McK87]. [Dor94] provides a recent survey of object-space hidden surface removal algorithms. McKenna and Seidel [MS85] have presented an algorithm for computing optimal shadows of a convex polytope. For static models, it is possible to precompute the visibility from all the viewpoints in space based on aspect graphs [GCS91]. In the worst case, for input models composed of n polygons, this algorithm can decompose the space into $O(n^9)$ regions, making it impractical for large models. The utility of all these algorithms for complex models is currently unclear.

In the last few years a number of techniques have been proposed for efficiently computing *conservative* visibility. These can be classified into two categories, object-space and image-space algorithms. More details on this classification are presented in [SSS74]. In object space, [Cla76] proposed view-frustum culling of a hierarchy of bounding volumes. Garlick et al. [GBW90] proposed using octree-based spatial subdivision to render polygons contained in the viewing frustum.

Several recent algorithms structure the database into *cells* or regions, and use a combination of off-line and on-line algorithms for cell-to-cell visibility and the conservative computation of the potentially visible set (PVS) of polygons [ARB90, TS91, LG95]. Such approaches have been successfully used in architectural walkthrough systems, where the division of a building into discrete rooms lends itself to a natural division of the database into cells. It is not apparent that cell-based approaches can be generalized to an arbitrary model, which may come with no structure information. Decomposing an arbitrary polygonal model into appropriate cells is rather difficult. Other algorithms for densely-occluded but somewhat less-structured models have been proposed by Yagel and Ray [YR96]. They use regular spatial subdivision to partition the model into cells and describe a 2D implementation. Some algorithms are based on binary space partition trees [FKN80, Nay92].

The hierarchical Z-buffer algorithm operates in both object-space and image-space [GKM93]. It combines spatial and temporal coherence with hierarchical structures. The algorithm exploits coherence by performing visibility queries on the Z-buffer. Currently, most graphics systems do not support this capability in hardware, and simulating the hierarchical Z-buffer in software is relatively expensive.

The work most directly related to our approach is that of Coorg and Teller [CT96, CT97]. Given two convex objects (an occluder and occludee), their early work required the construction and maintenance of a linearized portion of an aspect graph. They use this structure to track the viewpoint and determine whether one convex polytope occludes the other from a given viewpoint. This involves enumeration of all visual events and data structures for dynamic plane maintenance. In the worst case, the number of planes used to form a cell of the arrangement can be $O(m^2)$, where m is the number of vertices of the convex polytopes, though dynamic and hierarchical data structures are used in [CT96] to speed-up the computation of relevant planes. Each arrangement cell classifies all polytopes as completely, partially, or un- occluded. This approach is similar to earlier shadow computation algorithms which, given a light source and an occluder, decompose space into penumbra and umbra volumes. In their newer work they reduce the amount of coherence used and simplify the structure of the arrangement. This yields a considerable speedup, eliminating the overhead cost of maintaining complex data structures.

3 Algorithm Overview

Occlusion culling can be divided into two subproblems. First, for a given viewpoint we must select a small set of good occluders to use. Second, given good occluders, we must use them to cull away occluded portions of the model. These two problems are partially independent, and so we treat them separately in this paper.

In general, any primitive in the model to be rendered or any combinations of such primitives can be used as an occluder. In this paper, we restrict occluders to be either convex objects or those which can be expressed as union of two convex objects.

3.1 Data Structures

To perform occlusion culling, we require that the model be represented both in a spatial partition and in a spatial hierarchy. The spatial partitioning structure is constructed as part of preprocessing and used for occluder selection, while the spatial hierarchy is used for fast culling of occluded geometry. These may be separate data structures, or may be united into a single structure, based on the implementation.

The spatial partition is a discrete structure that spans space, mapping every point in the space of the model onto one of a finite number of partition *regions*. If the space of the model has volume v, and no region is a volume larger than v_r, we have at least $\frac{v}{v_r}$ regions in the partition.

The hierarchy is not a data structure concerned with the space of the model, but rather with the geometry of the model. Every piece of geometry is mapped to exactly one leaf *volume* of the hierarchy, and every volume of the hierarchy encloses all the geometry mapped to it or its children. Let the entire model consist of n pieces of geometry, and let each leaf volume contain no more than n_l geometry elements. Then, if we want a binary hierarchy (each non-leaf volume having two children), we need at least $2\frac{n}{n_l} - 1$ volumes in the hierarchy.

3.2 Occluder Selection

Finding good occluders in real time for an arbitrary model is a difficult task. Among all of those objects in the database which we might use, we must select the few which will occlude the most geometry from the current viewpoint with the least computational overhead. We make this task feasible by using a preprocess to discard all occluders which are *not* likely to be good when viewed from a given set of viewpoints. Then, at runtime, the results of this preprocess give us a list of occluders that are good over the local region of space, and we further reduce this to select those occluders that are best from the current viewpoint.

3.3 Visibility Culling

Once we have selected a few good occluders, they can be used by our culling algorithm. For each occluder we construct a *shadow frustum*: a frustum with its apex at the viewpoint, near plane determined by the occluder, and sides determined by the occluder's silhouette (as shown in Fig. 1). The space contained within this frustum is that which is not visible from the viewpoint due to the occluder. The pixels rendered for objects in this space would be discarded during depth comparison by the hardware z-buffer. Thus, model geometry which is completely contained within any of these shadow frusta (as is object A) need not be rendered.

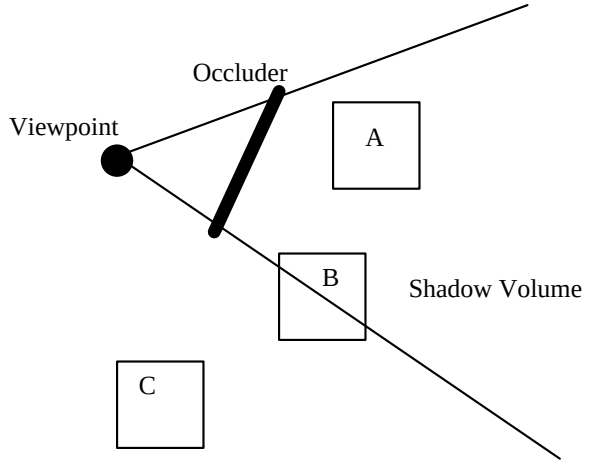


Figure 1: Relationship of bounding volumes to frusta.

4 Occluder Selection

Finding good occluders from a given viewpoint in a general unstructured model is a hard problem. By definition, a good occluder should occlude a large fraction of the geometry in the view frustum. For many viewpoints in the scene, no such good occluders may exist. In the worst case, computation of good occluders may correspond to computing the visible surface from that viewpoint using hidden surface removal algorithms. Some theoretically efficient algorithms of $O(n \log n + q)$ complexity, where q is the number of edges in the visibility map, have been proposed by [Mul89]. However, not much is known about their practical performance.

The other alternative is to preprocess the entire model to determine a set of useful occluders at every viewpoint. However, the computation of such global visibility information is more difficult than hidden surface removal from a particular viewpoint. The fastest known algorithms known for computing the effects on global visibility due to a single polyhedron with m vertices requires $O(m^6 \log m)$ time [GCS91].

Given the overall complexity of finding good occluders, we propose an approximation algorithm to find good occluders using a combination of online and offline techniques. Our algorithms work reasonably well in practice, but are not guaranteed to find good occluders all the time.

We pose occluder selection as an an optimization problem. Our goal is to use as few occluders as possible to keep CPU overhead down, but to occlude as many polygons as possible to keep the graphics pipeline lightly loaded. We take advantage of the empirical observation that for the data sets we are interested in, a few occluders cause most of the occlusion from most viewpoints, and using other occluders contributes little additional benefit[1].

[1] We performed experiments to confirm this. A typical result is shown in Figure 4. When we graph the fraction of polygons culled as a function of the number of occluders used, the optimum number of occluders is near the knee of the graph; in this case, approximately 8.

We use the following guiding principles in defining the optimization function to select good occluders:

1. **Solid Angle:** The viewed solid angle of a convex object is easily computable and measures the fraction of the visual field that it occupies. If we assume that the geometry of the scene is uniformly distributed about the viewpoint in all directions, the viewed solid angle is also directly proportional to the amount of geometry occluded.
2. **Depth Complexity:** We want to use occluders which occlude the maximum amount of geometry. In addition to using solid angle at runtime, we estimate the actual value of any occluder in the preprocess by random sampling. The algorithm selects some random viewpoints each partition region, constructs a shadow frustum from that viewpoint through each potentially good occluder, and determines the number of objects contained in the frustum. The average of several samples is a direct estimate of the value of the occluder.
3. **Coherence:** An occluder that does well or poorly at occluding for some frame will likely perform similarly for the next few frames. Similarly, objects that lie near a good occluder in screen space are themselves likely to be good occluders. The algorithm keeps track of the geometry culled by each occluder at each frame.

These criteria are integrated into our algorithm for both the online and offline processing of the geometry database. Rather than consider every polygon in the model as a potential occluder from every viewpoint, we use auxiliary data structures to quickly reduce the set of potential occluders to a manageable quantity. We can then further refine this set of potentially good occluders based on the exact viewing parameters of a particular frame.

4.1 Preprocess

The goal of the preprocess is to associate a set of potentially good occluders with every viewpoint in the model. We begin by constructing a spatial partition which divides the model into *regions*. Each region will store a list of potentially good occluders. At runtime, we determine which region contains the current viewpoint and use this region's associated list as a set of potentially good occluders. Thus, the region sizes control the granularity with which we sample the model's geometry for occlusion properties. Using small regions in our partition may shorten the list of potentially good occluders at each region, but this increases the number of regions in the data structure, thereby increasing its total memory usage.

Having constructed our spatial partition, we consider every potential occluder in the model[2]. For each potential occluder, we compute the set of viewpoints from which its viewed solid angle exceeds the threshold θ (a user-defined value). We use θ as the cutoff for consideration as an occluder. Objects whose viewed solid angles are less than the threshold cover only a small amount of screen space and, under the assumption that geometry is evenly distributed, are unlikely to occlude much geometry. For a single-sided ellipse, this set of viewpoints forms an ellipsoid and for a sphere, a concentric sphere. More complex occluders will have significantly more complex sets, which we approximate with the above. Each region of the partition which intersects this set of viewpoints adds the potential occluder to its list of potentially good occluders.

This solid-angle approximation is the first criterion of the runtime analysis we detail later. It is an *indirect* measure of the effectiveness of the occluder. We also pre-sample the second criterion, the actual amount of geometry behind the occluder, during the preprocess and store it in the region with the reference to the potentially good occluder. To do this, we choose a small number of viewpoints in the region and calculate the shadow frustum

[2] A separate preprocess may filter out small objects or simplify highly detailed objects to equivalently-occluding versions. Thus, the set of potential occluders need not be the same as the set of objects to be rendered.

cast by the occluder from each viewpoint. If we determine what fraction of the model geometry is actually occluded from each of these viewpoints and average our results together over all the viewpoints sampled, we have a *direct* estimate of the occluder's efficacy. Our selection of sample viewpoints be either random or guided by results on choosing optimal sampling patterns for antialiasing.

4.2 Runtime Computation

At every frame, we find the region of the spatial partition which contains the viewpoint. That region has an associated list of potentially good occluders obtained from the preprocess. The list is first narrowed by performing preliminary view-frustum culling to determine which potential occluders lie within the field of view. These potential occluders are then sorted based on our optimization function. The n_{occ} occluders which produce the highest value of the function are used as that particular frame's occluders.

Throughout the process we exploit coherence. The algorithm exploits temporal coherence of occlusion. When the viewpoint only moves a little between frames, the value of a given occluder only changes a little. Using coherence, we improve our expected sorting performance to linear time. Similarly, due to spatial coherence and the fact we use convex occluders, the algorithm easily tracks the silhouette of each occluder during the overlap tests.

5 Visibility Culling Using Occluders

In order to use occluders, we must have constructed a hierarchy of *bounding volumes* that contains the entire model. Having selected n_{occ} good occluders, we now proceed to perform occlusion culling with them. The critical requirement is to do so efficiently.

We begin by constructing a shadow frustum for each occluder. Taking the viewpoint as the apex of a frustum, we define the near plane of the frustum as a plane passing through the farthest point of the occluder's silhouette and whose normal points in the direction from that point towards the viewpoint. Each side of the frustum is a plane containing the viewpoint and an edge (two adjacent vertices) of the object's silhouette. This frustum contains the space which the occluder occludes from the viewpoint. Any geometry completely contained in the frustum is occluded and need not be rendered.

Using these shadow frusta requires an inorder traversal of the hierarchy. In fact, we incorporate view-frustum culling, shadow-frustum culling, and rendering into a single traversal of the hierarchy. We begin by marking all shadow frusta as active. As we encounter each volume during our traversal of the hierarchy, we first test for interference with the view frustum. If the volume is outside the view frustum, we are done with that sub-tree. Otherwise, we test for interference with all of the active shadow frusta. If the volume is entirely within any active frustum, the algorithm stops the current branch of the traversal without rendering any of the geometry in the volume. If the volume does not intersect any active frustum, we stop the current branch of the traversal and render all the geometry contained in the volume. Only if the volume partially overlaps some of the active frusta need we continue the traversal with that volume's children.

5.1 Overlap Tests

The algorithm determines the portions of the model occluded by a shadow frustum using interference tests with the bounding volume hierarchy. As the algorithm traverses the hierarchy, the intersection test between each bounding volume and the shadow frustum becomes a time-critical operation which directly influences the overall performance of occlusion culling. As shown in Figure 1, we need to quickly decide if a bounding volume is: (A) completely inside the shadow frustum, (B) partially overlapping the shadow frustum, or (C) completely outside the shadow frustum.

Consider the case when the occluder is a convex polytope. The shadow frustum must therefore be a convex volume. A number of efficient algorithms have been proposed in geometry and robotics literature for collision detection between convex polytopes. These are based on Minkowski sums [GJK88], closest features computation based on external Voronoi regions [LC91] and linear programming [Sei90]. All of them have been implemented and work reasonably well in practice. However, these algorithms are not directly applicable to our requirements for two reasons:

- **Limitation:** All these algorithms only check whether or not two objects are overlapping. They can not differentiate between partial overlap of two objects and complete containment of one object inside another, which is required for our algorithm.
- **Robustness:** Although the basic idea in all these algorithms is simple, their practical implementation are prone to robustness problems due to floating point arithmetic and geometric degeneracies.

In the geometry literature, other linear time algorithms have been proposed to compute intersections between convex polytopes [Cha89]. Unlike collision detection algorithms, they can unambiguously distinguish the three distinct cases shown in Figure 1. However, not much is known about their performance on real-world models.

5.2 General Algorithm

We project the silhouette of the occluder and the occludee onto the image plane. Each projection is a convex polygon and we refer to them as the occluder polygon (A) and the occludee polygon (B). Based on this projection, we reduce the problem to a 2D overlap test between two convex polygons. To check whether B is totally or partially contained inside A, the algorithm initially checks whether they are overlapping or not. Our algorithm uses a modified Cyrus-Beck clipping algorithm [FDHF90], which quickly determines if two polygons intersect and robustly computes an edge and a common point (call it O) contained in the intersection of two polygons if they are overlapping. There are also other robust implementations available for computing the intersection of two planar polygons.

To check whether B is totally contained inside A or not, we need to check whether any of their edges intersect. Given O, we use a sweep-line approach to check whether or not B is totally contained inside A. Therefore, the number of edge pairs we need to test for overlap is linear in the number of edges of the two polygons. In terms of robustness, this algorithm only requires a robust edge-edge overlap test.

5.3 Specialized Overlap Tests

Although the hierarchy of volumes may consist of arbitrarily-shaped convex volumes, this is rarely the case. Our work is a companion algorithm to view frustum culling, which normally uses hierarchies based on octrees, k-d trees, sphere-trees, OBB-Trees, R-trees etc. In all of these each node of the hierarchy is a sphere or a rectangular box. The rectangular box may be axis-aligned (an axis-aligned bounding box, or AABB) or be arbitrarily oriented (OBB).

In this section, we present robust and specialized overlap tests between a shadow frustum and an AABB or a OBB. Their overall running time is linear in the number of faces of the shadow frustum. However, we have also optimized the constant terms and overall operation count. The performance of the occlusion culling algorithm is dominated by these overlap tests.

The naive approach to exact interference detection requires an enormous amount of computation in the worst case: 54 "inside-outside" half-plane tests and 108 edge-face intersection tests (see Table 1). In the best case, it trivially rejects a box in 8 half-plane tests (dot products). We would like to maintain an exact test while drastically reducing the worst case cost and improving the best case cost. Our approach combines improved box-plane overlap tests and a fast edge-box intersection test using a parallel slabs representation for the bounding volume[Gre94, KK86].

Our box-plane overlap test uses the box polygon's normal to quickly find the box vertices which are closest to and farthest

Interference Test	Dot Products	Scalar-Vector Mult/Divide	Vector Add/Sub	Cross Products
Naive with far plane	270 (8)	216 (0)	1080 (0)	432 (0)
Naive without far plane	216 (8)	168 (0)	600 (0)	336 (0)
Specialized with far plane	294 (4)	144 (0)	144 (0)	0 (0)
Specialized without far plane	240 (4)	144 (0)	132 (0)	0 (0)

Table 1: Worst (and Best) case computational cost for the overlap tests

from the frustum's plane. Using these extremal points we need to check each plane against no more than two vertices, rather than eight. Oriented bounding boxes impose an overhead of only three additional dot products per plane tested at this stage.

We improve the worst case dramatically by speeding up expensive edge intersection tests. We can easily speed up the twelve tests between the frustum's edges and a box by representing our bounding volumes as three sets of slabs. This lets us avoid the edge-face test against each face individually by performing only a single edge-"solid" intersection test. Previously we required six expensive edge-face tests to test an edge against an entire box, but now we require only three edge-slab tests that completely eliminate the more expensive cross-product calculations.

Improving the twelve box-edge/frustum intersection tests proceeds similarly, using techniques from the ray-tracing literature. We represent the frustum as a set of planes forming a convex polyhedron and perform the edge test against the entire frustum at once. This also removes the expensive cross-product calculations entirely since we only have to perform six edge-plane intersection tests.

We further improve the performance by removing the far plane; this avoids having to perform the far-plane edge/box intersection tests and the view-frustum containment tests entirely (an infinite frustum never be contained in the bounding box). The improvements resulting from the new overlap test are shown in Table 1. Far planes are useful for view frusta, but are undesirable for shadow frusta.

6 Implementation and Performance

6.1 Data Structures

We have implemented the algorithms described in this paper. They scale well (efficiently and robustly) to large models. Our implementation has been interfaced within an unoptimized graphics viewer written using OpenGL. The viewer runs on top of UNC's view-frustum culling library and uses either exact or conservative versions of our specialized interference detection algorithm. We assume that *any* large model viewer will use view-frustum culling; results reported ignoring it are meaningless.

In our implementations, we also used the area-angle approximation presented by Coorg and Teller[CT96]. This is

$$-\frac{a\vec{\mathbf{N}}\cdot\vec{\mathbf{V}}}{|\vec{\mathbf{V}}|^2}$$

where a is the area of the polygon (in object space), $\vec{\mathbf{N}}$ the polygon's normal vector, and $\vec{\mathbf{V}}$ the vector from the viewpoint to the center of the polygon. This gives a good approximation of the subtended solid angle of the polygon.

The results reported in this paper were obtained using area-angle as a goodness criterion; we have seen better results using the full three-part goodness criterion (solid angle, preprocess estimation, and coherence), but have not yet tested it on the large model.

We use a uniform division of the bounding box of the entire database into rectangular voxels as our spatial partition for occluder selection, and a tree of axis-aligned or oriented bounding boxes as our hierarchy for view-frustum culling and occlusion culling. We could alternatively use an octree, BSP tree, or other hierarchical spatial partition to contain the polygons of the model, thus using a single data structure to serve both purposes. However, in our experience that approach has two drawbacks:

1. Most spatial partition algorithms subdivide the polygons (or other primitives), such that each polygon lies in exactly one region. Polygon count is typically increased by a factor of two by this type of subdivision. Any increase in polygon count can be expected to slow rendering, which is the process we are fundamentally trying to speed up.
2. The two data structures capture two different properties of the model which may not be spatially similar. The spatial partition *should be* dense, i.e. have the smallest volumes, in regions of the model where the best set of occluders to change frequently. The bounding volume hierarchy needs to capture the geometry of the model as closely as possible for best results in culling. Occlusion culling as a field would benefit from further investigation of the tradeoffs in the number of spatial data structures used by algorithms.

6.2 Comparison with Other Algorithms

Our specialized shadow frustum/bounding box interference test has been compared against efficient algorithms and implementations for collision detection between convex polytopes [CLMP95, GJK88]. Our overlap test is at least two times faster and more robust than efficient implementations of these algorithms. Notice that some of the earlier algorithms for collision detection between convex polytopes utilize temporal and spatial coherence. Since the shadow frusta may change between frames, our algorithm uses no coherence in performing the overlap tests.

6.3 Performance on a Large Model

To test our algorithm in environments of varying high depth complexity, we built a composite model of more than 500,000 polygons by fusing together a model of central London, a model of the Aztec ceremonial center at Tenochtitlan, and a partially-developed region of hills in England. See plates I - IV at the end of the paper. The input model came to us as a collection of polygons with no adjacency or any other structure. We used uniform grids for occluder computation and a bounding volume hierarchy based on axis-aligned bounding boxes for view frusta and occlusion culling.

On our SGI Onyx RealityEngine 2, along a 2748-frame path through the central regions of this model we saw up to 80% of the model that passed view-frustum culling occluded (with roughly 40% of total polygons occluded on average). Our measured average speedup in rendering was 55%. (The theoretical maximum is 65%, with the discrepancy due to the overhead of managing the occluders and their shadow frusta.)

Occlusion culling is fundamentally a viewpoint-dependent optimization; Figures 2 and 3 show the frame time and rendered polygon count for each frame in our long path through the city model. On some frames occlusion culling gives extremely good results, while on others it does not obtain significant culling, depending on the depth complexity.

Occlusion culling yields good results with surprisingly few occluders. We rapidly see decreasing returns beyond five or six occluders active. Figure 4 shows the results of one experiment

Method	Polygons per Frame	Time per Frame (us)	Frames per Second
View-Frustum Culling	49196	276	3.6
View-Frustum plus Occlusion Culling	29749	178	5.6

Table 2: Comparison of average statistics over 2748-frame path down streets of the 500,000-polygon city model. Our implementation improves average frame rate by 55%.

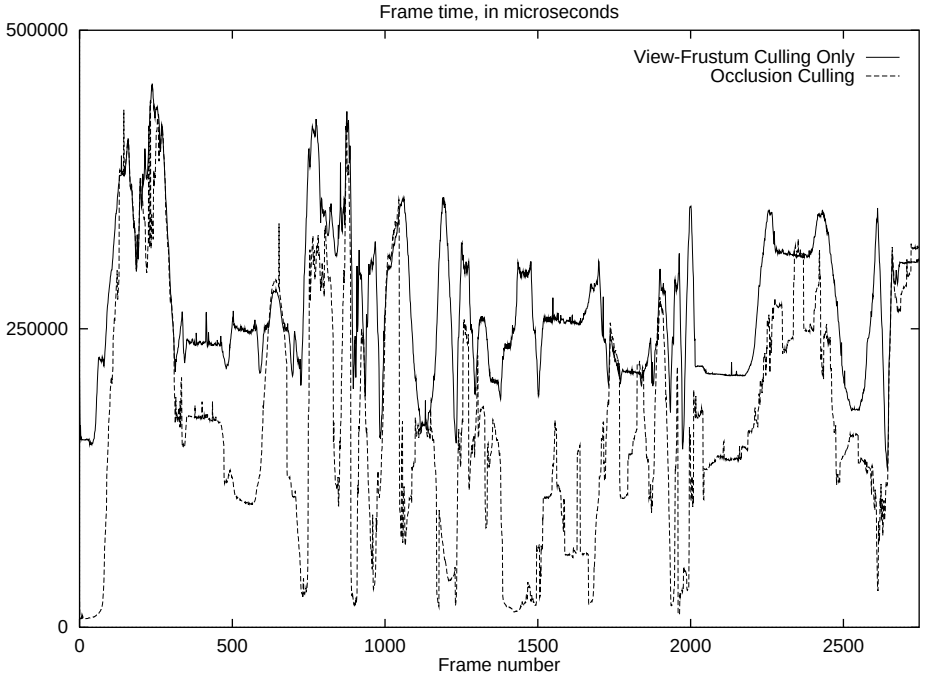


Figure 2: Time per frame, in microseconds, for each frame in the 2748-frame path through the large city model. Some viewpoints are good for occlusion culling, and at many of these points our implementation is seen to yield speedup.

with the city model, and we have seen similar results for other paths and other models.

6.4 Generality and Robustness

Generality and robustness have been main concerns in the design and implementation of our algorithm. The algorithm and implementation are applicable to all unstructured polygonal models. The algorithm requires no adjacency or connectivity information between the polygons. Only the silhouette tracking algorithm used for shadow frusta computation requires connectivity information. On unstructured models, we treat each convex polygon as a separate occluder.

Our specialized algorithms for overlap tests between the shadow frusta and bounding boxes are robust and not prone to degeneracies. They do not need to check for non-generic conditions such as parallel faces or edges. These are not special cases for the test and do not need to be handled separately. As a series of comparisons between linear combinations, the test is numerically stable. Our current implementation uses IEEE 64-bit floating point arithmetic instead of exact arithmetic for these overlap tests. Speed is an important concern in our system. We realize that floating point arithmetic can be a potential source of problem (though we have not experienced any so far), especially when one face of the box is just touching the shadow frusta. Many recent papers by Fortune, van Wijk, Clarkson, Boissonnat et al, have demonstrated that clever use of floating point arithmetic can yield very effective and yet correct implementations of geometric primitives. Many libraries like LEDA [MN95] have implementations of such algorithms for line segment intersections. As a result, it should be possible to have fast correct implementations of our specialized overlap test. For the general algorithm, we reduce the problem to 2D overlap tests between two convex polygons and use robust public domain implementations (available in Graphics Gems).

7 Algorithm Analysis

The overall asymptotic running time of the algorithm is at most linear in n, where n is the total number of polygon in the model. The worst-case running times for the various phases of the algorithm are:

Occluder Selection: We select the n_{occ} active occluders by sorting a list of n_{pot} potentially good occluders. In practice, n_{occ} is typically a small constant (between five and ten), and n_{pot} is a small fraction of the input primitives. The total complexity of this phase is $O(n_{occ}n_{pot})$ or $O(n_{pot}log(n_{pot}))$.

Shadow Frusta Computation: Constructing shadow frusta based on silhouette tracking for these occluders requires time linear in the number of edges per occluder, which is a small constant k, for a phase complexity of $O(kn_{occ})$.

Visibility Culling: We use the n_{occ} shadow frusta in the same manner as view frusta; each view-frustum cull is worst-case linear in the size of the model. The complexity of this phase is $O(n_{occ}n)$.

Since n_{occ} is a small constant, and $n_{pot} << n$, our algorithm is in the worst case linear: $O(n)$. The spatial hierarchy constructed for occluder use keeps the constant in this term very small. We also use spatial and temporal coherence in resorting the list of potentially good occluders by using insertion sort. Since the ordering of this list only changes slightly between frames, that phase has an expected time linear in the number of potentially good occluders n_{pot}.

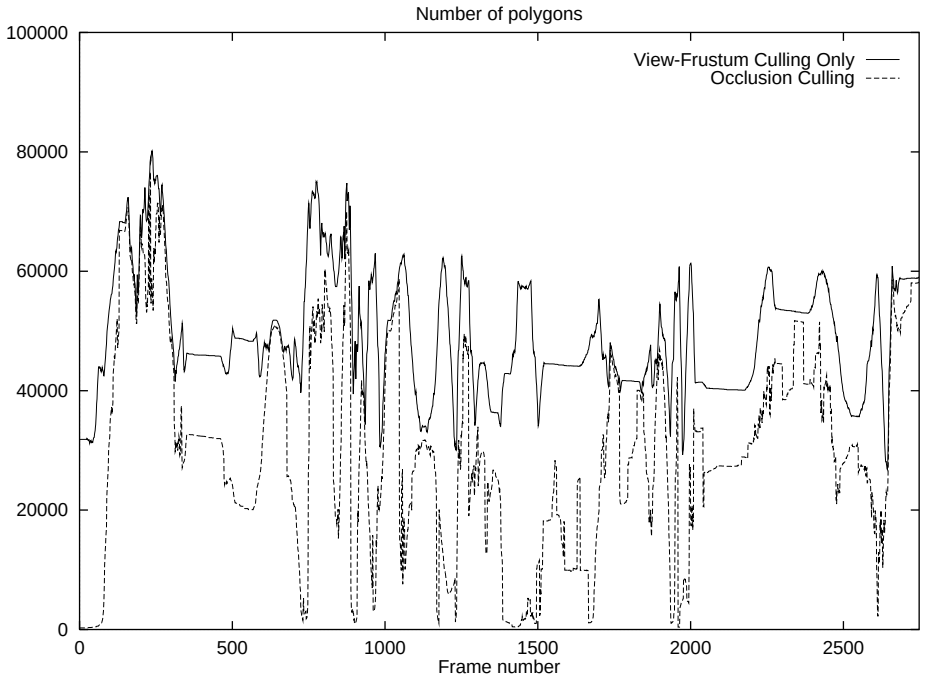


Figure 3: Polygons rendered per frame along a 2748-frame path through the large city model. Some viewpoints are good for occlusion culling, and at many of these points our implementation is seen to reduce the number of polygons rendered, which translates into less bus bandwidth consumed.

7.1 Main Features

Our main goals in developing these algorithms and systems have been robustness, applicability to large unstructured models, and scalability. None of the earlier conservative visibility algorithms has been applied to large and unstructured models composed of hundred of thousands of polygons. As a result, it is somewhat difficult to make exact comparisons in terms of robustness and efficiency.

Some of the main features of our approach are:

- **Robustness**: Since we use specialized interference detection algorithms for determining occluder-occludee relationships, our algorithm is not prone to robustness problems and geometric degeneracies.
- **Efficiency:** The additional CPU overhead due to occlusion culling varies with the viewpoint and is a linear function of the number of occluders chosen at that particular frame. In most cases, we use five to ten occluders. We use efficient and specialized algorithms for overlap tests between a shadow frustum and a hierarchy of bounding volumes. As a result, our algorithm performs interactively on large models composed of hundreds of thousands of polygons.
- **Generality:** The algorithm is applicable to all polygonal and unstructured models. In many CAD applications, the input models come as "polygon soup," with no structure or adjacency information available for constructing winged-edge or similar data structures. The presence of structure increases the effectiveness of our approach, but the results reported herein were obtained on unstructured data sets.
- **Ease of Implementation:** The implementation requires only simple data structures plus a general view frustum culling library. Since the occlusion culling algorithm sees the model as composed of simple bounding volumes (like spheres, rectangular boxes etc.), we use simple and easily-specialized overlap tests.

7.2 Ongoing Work

Often geometry is not occluded by any single convex occluder, but is occluded by the combination of several occluders. This is made more difficult to take advantage of in an object-space solution by the fact that the union of several convex occluders is not necessarily convex. We have developed and are implementing extensions to our interference detection algorithms to combine occluders in object space.

In [ZMHH97], Zhang et al. have successfully used the graphics pipeline to merge these occluders in the image space and form occlusion maps. Based on that they have proposed a two-pass algorithm. In the first pass, the algorithm constructs an occlusion map hierarchy and in the second pass used to cull away portions of the model not visible from the current viewpoint. This algorithm is able to cull away a higher percentage of the model than our object space approaches, but requires a high performance graphics pipeline. On the other hand, the algorithm presented in this paper does not use the graphics pipeline for visibility culling and should thus be useful for low-end hardware or personal computers.

8 Acknowledgements

Some of the models used in this work were originally constructed by:

- Dr. Vasilis Bourdakos of the Centre for Advanced Studies in Architecture, Bath University.
- Bob Galbraith Computer Graphics, East Longmeadow, MA.

References

[ARB90] J. Airey, J. Rohlf, and F. Brooks. Towards image realism with interactive update rates in complex virtual building environments. In *Symposium on Interactive 3D Graphics*, pages 41–50, 1990.

[BCG+96] G. Barequet, B. Chazelle, L. Guibas, J. Mitchell, and A. Tal. Boxtree: A hierarchical representation of surfaces in 3d. In *Proc. of Eurographics'96*, 1996.

[BDEG94] M. Bern, D. Dobkin, D. Eppstein, and R. Grossman. Visibility with a moving point of view. *Algorithmica*, 11:360–78, 1994.

[Cha89] B. Chazelle. An optimal algorithm for intersecting three-dimensional convex polyhedra. *Proc. 30th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 586–591, 1989.

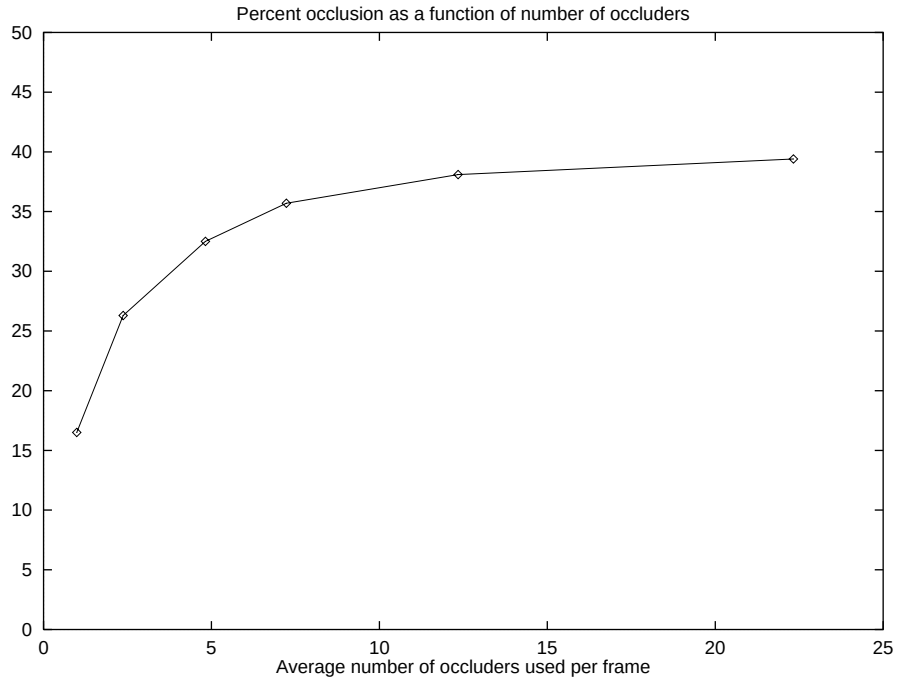


Figure 4: Average percentage of polygons within view frustum culled by occlusion culling as a function of average number of occluders active, along a 2748-frame path through the large city model. This supports the observation that for any given viewpoint a small number of occluders provide most of the occlusion.

[Cla76] J.H. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, 1976.

[Cla88] K. L. Clarkson. Applications of random sampling in computational geometry, II. In *Proc. 4th Annu. ACM Sympos. Comput. Geom.*, pages 1–11, 1988.

[CLMP95] J. Cohen, M. Lin, D. Manocha, and M. Ponamgi. I-collide: An interactive and exact collision detection system for large-scale environments. In *Proc. of ACM Interactive 3D Graphics Conference*, pages 189–196, 1995.

[Cro77] F. C. Crow. Shadow algorithms for computer graphics. *ACM Computer Graphics*, 11(3):242–248, 1977.

[CT96] S. Coorg and S. Teller. Temporally coherent conservative visibility. In *Proc. of 12th ACM Symposium on Computational Geometry*, 1996.

[CT97] S. Coorg and S. Teller. Real-time occlusion culling for models with large occluders. In *Proc. of ACM Symposium on Interactive 3D Graphics*, 1997.

[Dor94] S. E. Dorward. A survey of object-space hidden surface removal. *Internat. J. Comput. Geom. Appl.*, 4:325–362, 1994.

[FDHF90] J. Foley, A. Van Dam, J. Hughes, and S. Feiner. *Computer Graphics: Principles and Practice.* Addison Wesley, Reading, Mass., 1990.

[FKN80] H. Fuchs, Z. Kedem, and B. Naylor. On visible surface generation by a priori tree structures. *Proc. of ACM Siggraph*, 14(3):124–133, 1980.

[For96] S. Fortune. Robustness issues in geometric algorithms. In M.C. Lin and D. Manocha, editors, *Applied Computational Geometry*, pages 9–14. Springer-Verlag, 1996.

[GBW90] B. Garlick, D. Baum, and J. Winget. Interactive viewing of large geometric databases using multiprocessor graphics workstations. *Siggraph'90 course notes: Parallel Algorithms and Architectures for 3D Image Generation*, 1990.

[GCS91] Z. Gigus, J. Canny, and R. Seidel. Efficiently computing and representing aspect graphs of polyhedral objects. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(6):542–551, 1991.

[GJK88] E. G. Gilbert, D. W. Johnson, and S. S. Keerthi. A fast procedure for computing the distance between objects in three-dimensional space. *IEEE J. Robotics and Automation*, vol RA-4:193–203, 1988.

[GKM93] N. Greene, M. Kass, and G. Miller. Hierarchical z-buffer visibility. In *Proc. of ACM Siggraph*, pages 231–238, 1993.

[GLM96] S. Gottschalk, M. Lin, and D. Manocha. Obb-tree: A hierarchical structure for rapid interference detection. In *Proc. of ACM Siggraph'96*, pages 171–180, 1996.

[Gre94] N. Greene. Detecting intersection of a rectangular solid and a convex polyhedron. In *Graphics Gems IV*, pages 74–82. Academic Press, 1994.

[HHK89] C. Hoffmann, J. Hopcroft, and M. Karasick. Robust set operations on polyhedral solids. *IEEE Computer Graphics and Applications*, 9(6):50–59, 1989.

[HKM95] M. Held, J.T. Klosowski, and J.S.B. Mitchell. Evaluation of collision detection methods for virtual reality fly-throughs. In *Canadian Conference on Computational Geometry*, 1995.

[KK86] T. Kat and J. Kajiya. Ray tracing complex scenes. *Computer Graphics*, pages 269–278, 1986.

[LC91] M.C. Lin and John F. Canny. Efficient algorithms for incremental distance computation. In *IEEE Conference on Robotics and Automation*, pages 1008–1014, 1991.

[LG95] D. Luebke and C. Georges. Portals and mirrors: Simple, fast evaluation of potentially visible sets. In *ACM Interactive 3D Graphics Conference*, Monterey, CA, 1995.

[McK87] M. McKenna. Worst-case optimal hidden-surface removal. *ACM Trans. Graph.*, 6:19–28, 1987.

[MN95] K. Mehlhorn and S. Näher. LEDA: a platform for combinatorial and geometric computing. *Commun. ACM*, 38:96–102, 1995.

[MS85] M. McKenna and R. Seidel. Finding the optimal shadows of a convex polytope. In *Proc. 1st Annu. ACM Sympos. Comput. Geom.*, pages 24–28, 1985.

[Mul89] K. Mulmuley. An efficient algorithm for hidden surface removal. *Computer Graphics*, 23(3):379–388, 1989.

[Nay92] B. Naylor. Interactive solid geometry via partitioning trees. In *Proc. of Graphics Interface*, pages 11–18, 1992.

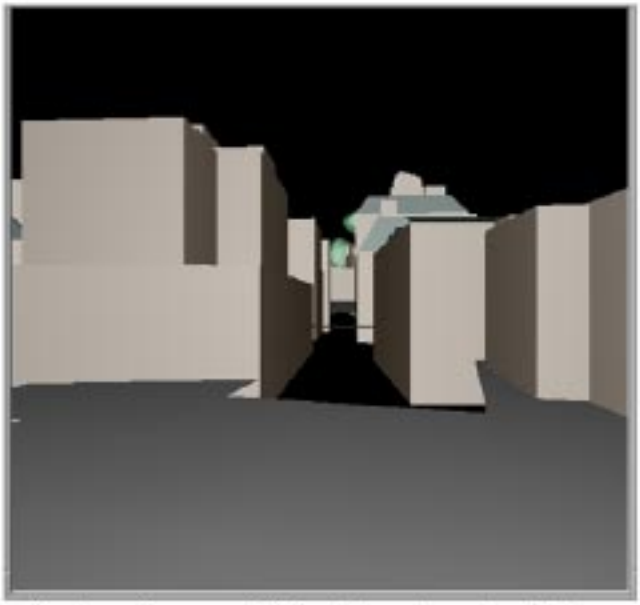
[Sei90] R. Seidel. Linear programming and convex hulls made easy. In *Proc. 6th Ann. ACM Conf. on Computational Geometry*, pages 211–215, Berkeley, California, 1990.

[SSS74] I. Sutherland, R. Sproull, and R. Schumaker. A characterization of ten hidden-surface algorithms. *Computing Surveys*, 6(1):1–55, 1974.

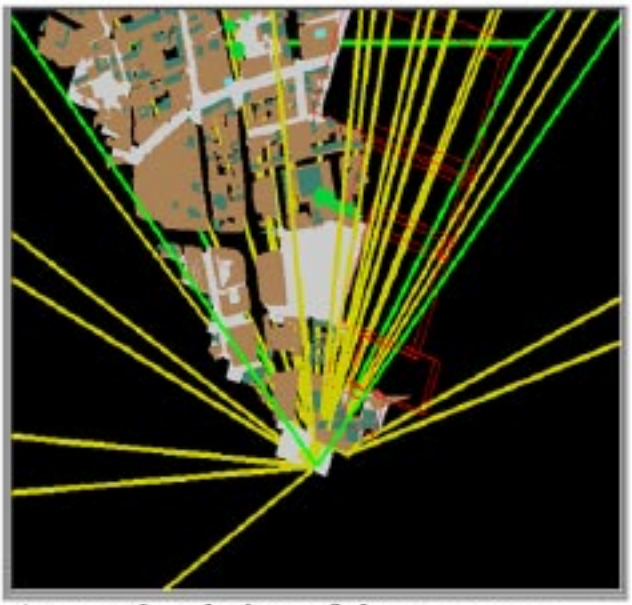
[TS91] S. Teller and C.H. Sequin. Visibility preprocessing for interactive walkthroughs. In *Proc. of ACM Siggraph*, pages 61–69, 1991.

[YR96] R. Yagel and W. Ray. Visibility computations for efficient walkthrough of complex environments. *Presence*, 5(1):1–16, 1996.

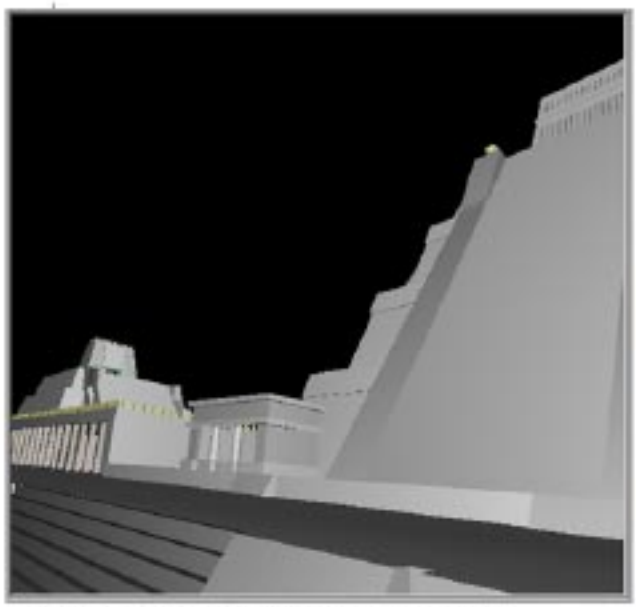
[ZMHH97] H. Zhang, D. Manocha, T. Hudson, and K. Hoff. Visibility culling using hierarchical occlusion maps. Technical Report TR97-004, Department of Computer Science, University of North Carolina, 1997. To Appear in Proc. of ACM Siggraph'97.



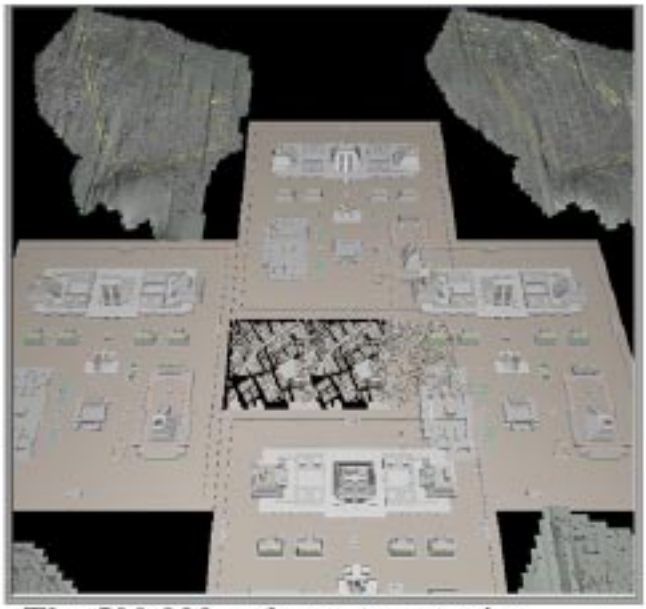
A view from within the streets of the London model. Note how all the outlying portions of the model are occluded by buildings on streets near the viewpoint.



An overhead view of the same area, showing the view frustum (green), the shadow volumes (yellow) of the occluders chosen by our algorithm, and the regions (red) culled away as a result.



A close-up of the Aztec portion of the composite model, showing the great range of scales - from tiny detail polygons to huge blocks.



The 500,000-polygon composite model used in some of our tests, built out of London, Tenochtitlan, and central England.