In Proc. 1997 ACM Symposium on Interactive 3D Graphics, pp. 83-90 and 189.

Real-Time Occlusion Culling for Models with Large Occluders

Satyan Coorg Seth Teller
Computer Graphics Group
MIT Laboratory for Computer Science

Abstract

Efficiently identifying polygons that are visible from a dynamic synthetic viewpoint is an important problem in computer graphics. Typically, visibility determination is performed using the z-buffer algorithm. As this algorithm must examine every triangle in the input scene, z-buffering can consume a significant fraction of graphics processing, especially on architectures that have a low performance or software z-buffer.

One way to avoid needlessly processing invisible portions of the scene is to use an occlusion culling algorithm to discard invisible polygons early in the graphics pipeline. In this paper, we exploit the presence of large occluders in urban and architectural models to design a real-time occlusion culling algorithm. Our algorithm has the following features: it is *conservative*, i.e., it overestimates the set of visible polygons; it exploits *spatial coherence* by using a hierarchical data structure; and it exploits *temporal coherence* by reusing visibility information computed for previous viewpoints. The new algorithm significantly accelerates rendering of several complex test models.

CR Categories and Subject Descriptors: I.3.3 [Computer Graphics]: Three-Dimensional Graphics and Realism – *visible line/surface algorithms*; I.3.7 [Computer Graphics]: Computational Geometry and Object Modeling – *object hierarchies.*

Additional Keywords: Conservative visibility, temporal coherence, spatial coherence, kD-trees.

1 Introduction

Identifying visible polygons or eliminating hidden polygons is an important component of efficient scene rendering algorithms. Despite the availability of high performance z-buffer hardware, a significant fraction of graphics machines have lesser or no hardware z-buffering capabilities. Software z-buffering (e.g., on personal computers), can be a rendering bottleneck. Moreover, on many architectures, the z-test occurs after other graphics processing (e.g., shading, texture mapping), wasting computation on invisible portions of the model.

Address: 545 Technology Square, Cambridge, MA 02139
Email: {satyan,seth}@graphics.lcs.mit.edu

To appear in the ACM Symposium on Interactive 3D Graphics, Providence, Rhode Island, April 27–30, 1997

One way to address this problem is to develop occlusion culling algorithms that efficiently identify, then render, only the visible portions of the model or a tightly-bounded superset thereof. In this paper, we exploit the presence of large occluders in typical architectural or urban models to design a real-time occlusion culling algorithm.

Our algorithm is based on several ideas. First, we propose a simple (and fast) visibility test that identifies whether some region of the model is completely/partially occluded by a set of occluders. Second, we describe a cheap preprocessing step that identifies nearby large occluders for all viewpoints. Finally, a hierarchical visibility algorithm repeatedly applies the visibility test to determine the status of tree nodes in a spatial hierarchy.

1.1 Related Work

Given a 3D model and a viewpoint, exact visibility algorithms [12, 14] compute a description of the image in terms of visible polygon fragments. Once such a description is available, further processing can be restricted to involve only the visible portions of the scene. However, these techniques tend to be complex and hence difficult to use in interactive applications. Instead, the z-buffer algorithm [4], typically implemented in hardware [2], is widely used.

Given the availability of hardware z-buffers, it seems promising to *overestimate* the set of visible polygons, then use a z-buffer to render the final image. This idea of overestimated or *conservative* visibility has been exploited to design fast architectural walkthrough systems [1, 6, 10, 15]. The idea in [15] is to subdivide the input model into *cells*, roughly corresponding to rooms in a building, and use *cell-to-cell/eye-to-cell* visibility to bound exact visibility from above. Though this method eliminates most invisible polygons in architectural models, its generalization to models with less apparent cell structure (e.g., city models) appears difficult.

An approach using octree-based spatial subdivision is used in [7] to render only those polygons that lie within the viewing frustum. However, this algorithm does not exploit any occlusion properties of the model.

The hierarchical z-buffer algorithm [9] culls nodes of an octree hierarchy by using a z-pyramid to resolve visibility queries. While this is a promising approach for implementing occlusion culling with hardware support, it is difficult to realize this algorithm on current graphics architectures, as visibility queries are not supported in hardware, and simulating the z-pyramid in software entails significant overhead. Because of its reliance on image space queries, this algorithm is also susceptible to aliasing artifacts (although, with an accompanying increase in complexity, it is possible to avoid aliasing [8]).

A dynamic temporally coherent conservative visibility algorithm, described in [5], identifies *relevant* visibility events, i.e., changes in visibility that will occur in the near future. One drawback of this algorithm is that it must reconstruct visibility information for the continuous sequence of points between each discrete pair of subsequent viewpoints assumed by the moving observer.

Finally, recent algorithms proposed in [11, 13, 16] accelerate rendering by approximating sets of polygons with texture maps. Visibility is resolved only when computing the texture approximation corresponding to a set of polygons. These textures are used to render many frames, thereby reducing further visibility processing. Texture approximation is usually most effective for faraway polygons, as there is little change in their image from one viewpoint to the next. In contrast, occlusion culling can eliminate even (invisible) nearby polygons.

1.2 Algorithm Overview

We assume that the input model is a static set of convex polygons and that the number of vertices in each polygon is bounded by some constant. Our system uses a preprocessing step to merge identical input vertices. This is useful in identifying polygons that share edges. We assume no *a priori* knowledge of observer motion.

Our algorithm chooses a small set of occluders which will be later used for culling. This strategy is motivated by the observation that, in many interesting models, most of the occlusion is caused by a few polygons (from any instantaneous viewpoint). Crucially, occluders are chosen dynamically, as the viewpoint changes, so that polygons typically act as occluders for nearby viewpoints, but as occludees (culled objects) for remote or oblique viewpoints (Figure 1). Usually, the occludee is not just a single polygon, but a convex region of space (e.g., a hierarchical bounding box) containing many polygons.

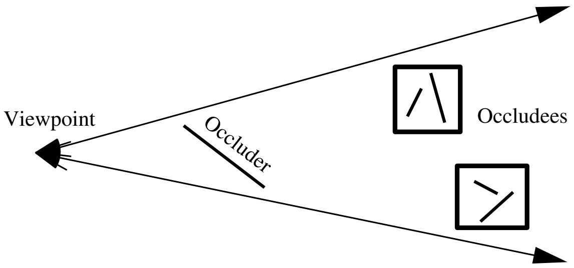


Figure 1: Occluder polygons and occludee objects for an instantaneous viewpoint.

The rest of the paper is organized as follows. Section 2 describes an algorithm to determine the conservative visibility status of an occludee with respect to a set of occluders. This test is the basis of a *visibility oracle* in the dynamic algorithm presented in Section 3. Section 4 presents the performance characteristics of our algorithm, and Section 5 concludes.

2 Conservative Visibility Testing

This section addresses the following problem: given a viewpoint, a set of convex occluders and a convex occludee, is the occludee *visible*? That is, does there exists a line segment from the viewpoint to some point on the occludee that meets no occluder? Our method of answering this query uses the notion of *supporting* and *separating* planes (Figure 2). Separating planes of two convex polyhedral objects are planes formed by an edge of one object and a vertex of the other such that the objects lie on opposite sides of the plane. Supporting planes are analogous, except that both objects lie on the same side of the plane.

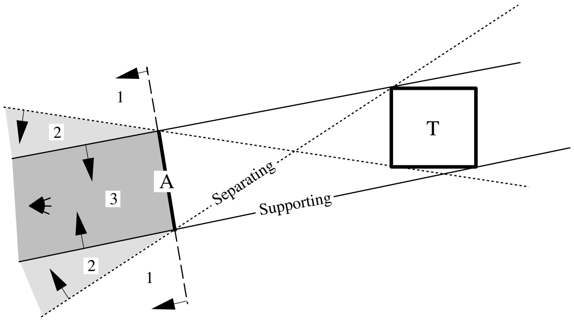


Figure 2: This figure shows occlusion in two dimensions (a planar cross section of 3D). Separating and supporting planes of an occluder A and an occludee T are shown, as is a synthetic viewpoint.

Consider the interaction between a single occluder A and an occludee T (typically a bounding box). This can be completely described in terms of the plane of the occluder and the supporting and separating planes of the occluder and occludee. First, A can occlude T only if the viewpoint lies in that half-space of A which does not contain T. This region can be divided into three qualitatively distinct regions as shown in the figure. In region 1, T is not occluded by A; in region 2, T is partially occluded by A; and in region 3, T is completely occluded by A.

The supporting and separating planes of A and T can be used to detect which of these cases holds. First, the planes are oriented toward the occluder to form half-spaces. We say that a viewpoint *satisfies* a plane iff it is inside the plane's positive half-space; this relation can be checked by performing an inner product of the viewpoint with the plane equation. Full occlusion occurs when all of the supporting planes are satisfied; that is, when the viewpoint is in the intersection of the supporting half-spaces (region 3). Partial occlusion occurs when all the oriented separating planes are satisfied, but some supporting plane is not (region 2). Otherwise, there is no occlusion (region 1).

Figure 3 shows occlusion caused by two connected occluders, i.e., two occluders that share an edge. If the viewpoint is in the shaded region, T is occluded by the combined effect of A and B, even though neither occludes it alone. This case is handled by *ignoring* the supporting planes through non-silhouette shared edges (e.g., edge E), if both polygons adjacent to E partially occlude T.

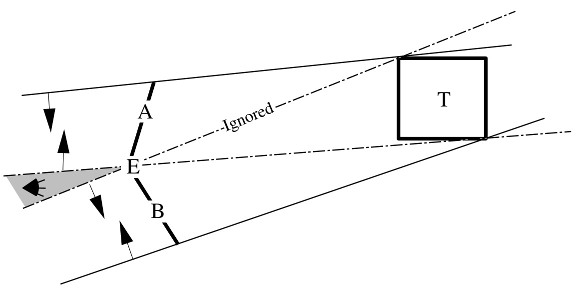


Figure 3: Occlusion by connected occluders A, B.

Note that occlusion occurs only if A and B lie on opposite sides of E, as seen from the viewpoint – intuitively, E is relevant only when it is a silhouette edge of the occluder as seen from the viewpoint. Figure 4 shows a case in which ignoring the supporting plane (dashed) through E would cause T to be classified, incorrectly, as fully occluded.

In general, a set of occluders $A_1, \ldots, A_k$ *jointly occludes* T if:

- $A_1, \ldots, A_k$ partially occlude T, and none fully occludes T;
- If two occluders A_i and A_j share an edge E, they lie on opposite sides of E as seen from the viewpoint; and

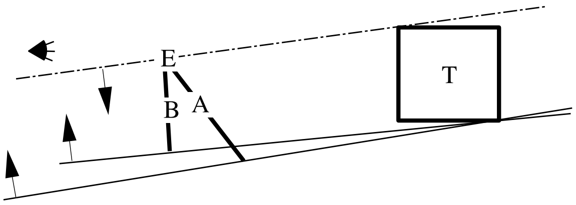


Figure 4: This figure shows that supporting planes through shared silhouette edges cannot be ignored.

- The signed distances of the viewpoint from all planes, other than those supporting common edges, are positive.

The above algorithm is simple to implement given the supporting and separating planes corresponding to a single occluder and occludee. An efficient way to compute these is described in Section 2.1.

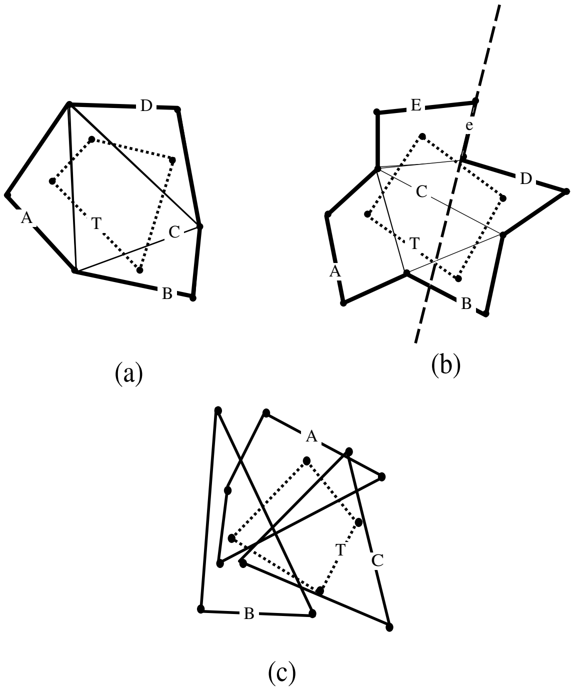


Figure 5: This figure shows occlusion in 3D, as seen from the viewpoint. Part (a) shows occlusion caused by connected occluders whose silhouette is convex in the image. Part (b) shows occlusion by connected occluders having a non-convex silhouette. Part (c) shows occlusion by a set of disconnected occluders.

Consider Figure 5, which depicts three different occlusion cases. The test described above detects occlusion in Figure 5-(a), as the supporting planes through the "internal" edges are ignored, and the occludee lies entirely within the convex silhouette in the image. The conservative visibility test may not detect occlusion if the silhouette edges form a non-convex polygon (Figure 5-(b)). In this case, the test fails as the occludee appears to "cross" the line supporting edge e in the image. Finally, the test does not detect occlusion caused by a set of disconnected occluders (Figure 5-(c)). In practice, we have found this to be a reasonable tradeoff for architectural and urban models, where most occlusion is due to large occluders acting alone or as part of a connected set. Note that our visibility test is conservative, i.e., it never misclassifies a visible entity as occluded.

2.1 Computing Supporting and Separating Planes

In principle, it is possible to precompute all planes formed by occluder/occludee pairs. However, this would be wasteful, as only a small fraction of such planes would ever be used. Instead, we use a hybrid method that combines preprocessing and run-time table lookups to compute supporting and separating planes formed by an (arbitrary) occluder and an axial bounding box (an occludee).

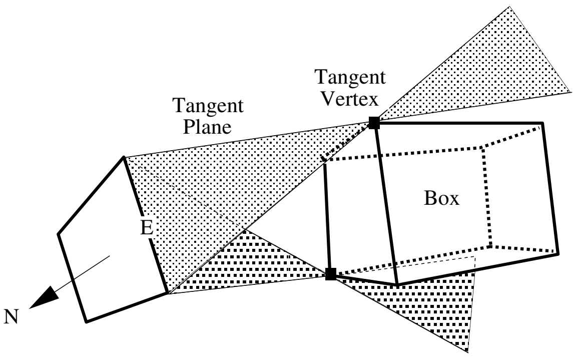


Figure 6: Tangent planes through a polygon edge E.

We restrict the separating and supporting planes of an occluder/occludee pair to the set of tangent planes formed by occluder edges and occludee vertices (Figure 6), and ignore those formed by occluder vertices and occludee edges. Assembling only these planes provides an exact test for full occlusion, and a conservative test for partial occlusion. Tangent vertices (and consequently, tangent planes) for each occluder edge are computed using the following steps:

1. Translate the occludee bounding box so that the origin is one vertex of the edge.
2. Determine the silhouette of the bounding box as seen from the origin, using a table lookup based on the box's vertices (Figure 7).
3. In the 2D projection with respect to the origin (along a direction toward the box), the occluder edge projects to a single point. Determine tangents from this point to the box's silhouette (Figure 8). As the property of tangency is retained by projection, the tangent vertices determined in 2D are the tangent vertices in 3D.

   Computing tangent vertices in 2D is a table lookup based on the location of the edge projection with respect to the silhouette edges. Due to the special properties of the silhouette edges (i.e., projection of axial edges with respect to the origin), this computation can be easily performed.

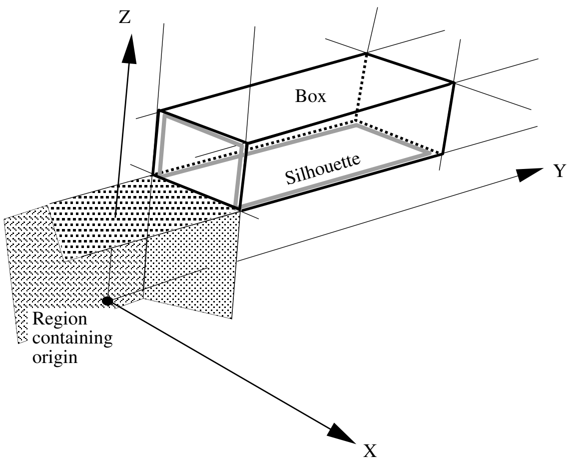


Figure 7: Silhouette edges of an axial bounding box when viewed from the origin.

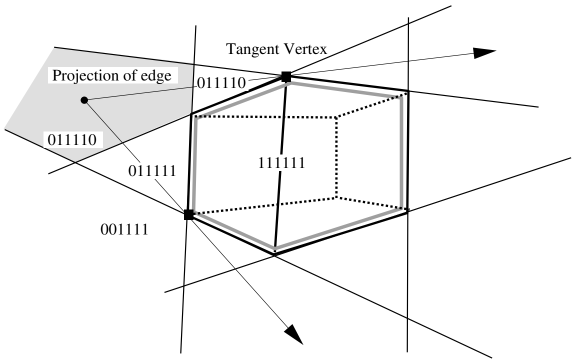


Figure 8: The projection of a box with respect to the origin (shown in 2D). Regions induced by extended silhouette edges are numbered using 6-bit codes consisting of the signs of dot products with (directed) silhouette lines. Tangent vertices are determined by the location of the occluder edge's projection.

This method speeds up computation of supporting and separating planes, a key component of the visibility test, by an order of magnitude over the naive method of computing and checking planes through each vertex of the bounding box.

```
Visible(OccluderSet S, kD-Tree Node T, Viewpoint P, PolygonSet OS)
    TestOcclusion(S, T, P);
    if (S does not occlude T) then          // all of subtree T is visible; report
        Gather(T, OS);
    else if (T is occluded) then            // all of subtree T is invisible; omit
        return;
    else if (T is a leaf) then
        Gather(T, OS);                      // conservative; report T as visible
    else
        S' = {};
        for (A_i ∈ S) do                    // determine occluder set S'
            if (A_i partially occludes T) then
                S' = S' ∪ {A_i}
        for each child T' of T do           // apply S' to subtrees of T
            Visible(S', T', P, OS)
```

Figure 9: The visibility algorithm.

3 The Visibility Algorithm

Applying the visibility test to every possible visual interaction (which could consume $O(n^2)$ resources, for n polygons) is expensive. Instead, we use two techniques to reduce the number of visibility tests performed. First, the algorithm operates on a hierarchical data-structure: a kD-tree [3] organizing all model polygons. Second, only a small, dynamically maintained set of occluders near the viewpoint is used to determine occlusion (Section 3.1).

Given a kD-tree, the visibility algorithm (Figure 9) reports those polygons in the kD-tree that are *not* occluded by the specified occluders. In the algorithm, $Gather(T, OS)$ simply collects all polygons reachable from a kD-tree node T and unions them to the set OS. $TestOcclusion(S, T, P)$ determines the visibility status of T with respect to occluders in the set S when viewed from P, using the algorithm described in Section 2.

The visibility algorithm recursively applies the conservative visibility test to determine the visibility status of each kD-tree node. First, the conservative visibility test is applied to determine whether the current node is invisible. If so, the algorithm returns without performing any further work – the entire subtree rooted at the current node is occluded. If it fails, the algorithm recurses with those occluders that partially occlude the kD-tree node, since only these occluders can occlude any descendant kD-tree node.

Complexity

The complexity of this algorithm is $O(kv)$, where v is the number of kD-tree nodes visited and k is the number of occluders. In the worst-case, this complexity could be $O(kn)$ if the algorithm tests all kD-tree nodes against all occluders. In practice, the complexity is lower, as only a fraction of the kD-tree nodes are tested against each occluder (see Section 4 for further details).

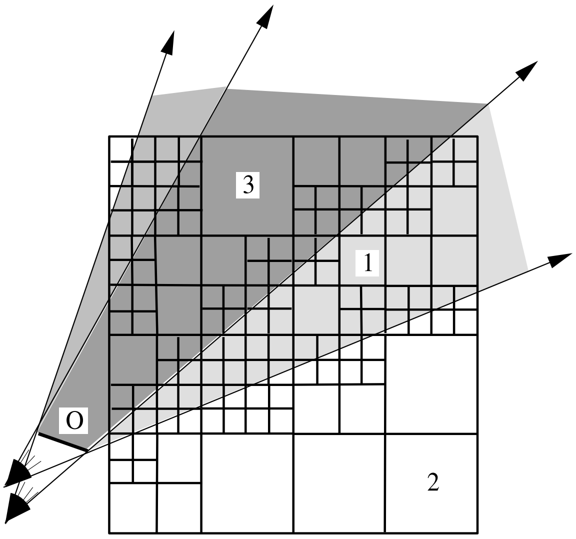


Figure 10: kD-tree nodes classified with respect to a single occluder (O) and two consecutive viewpoints. For clarity, subdivision in the kD-tree is assumed to occur at the center of each kD-tree node.

Spatial Coherence

Figure 10 shows kD-tree nodes visited during visibility classification of a hierarchy with respect to a single occluder and two consecutive viewpoints. Note that using the hierarchy exploits spatial coherence by avoiding further testing of nodes that are completely occluded (e.g., node 1) or that are completely visible (e.g., node 2).

Temporal Coherence

For a moving observer, the algorithm *caches* the occlusion relations – a list of supporting and separating planes – at each visited kD-tree node. When the viewpoint changes, the algorithm need only *check* existing occlusion relations, and update those kD-tree nodes whose visibility status has changed. For example, in Figure 10, nodes in the dark region (e.g., node 3) are checked against the occluder in both the traversals. For such nodes, the separating and supporting planes that are needed to determine their visibility status are computed only at the first viewpoint and reused later. The cache entries corresponding to a node are maintained until the node is no longer involved in the visibility calculation (i.e., until an ancestor node becomes fully occluded or fully visible).

Frustum Culling

View frustum culling can be incorporated into the algorithm in a straightforward way, by first checking whether the kD-tree node is inside the viewing frustum before invoking the visibility algorithm. However, any supporting and separating planes computed for nodes outside the viewing frustum are still retained in the cache so that they can be reused later.

3.1 Dynamic Occluder Selection

The algorithm as described maintains the state of each kD-tree node with respect to a fixed set of occluders. As the viewpoint moves, it is crucial to update this occluder set to contain those polygons that are "large" in the image, and therefore likely to occlude substantial portions of the model. Likewise, polygons that become small in apparent size should be ejected from the set.

A simple metric for the occlusion potential of a polygon from a given position is the solid angle it subtends at that position. A reasonable estimate of solid-angle is the quantity

$$\frac{-A(\vec{N}\cdot\vec{V})}{||\vec{D}||^2}$$

where A represents the area of the occluder, $\vec{N}$ represents the normal, $\vec{V}$ represents the viewing direction, and $\vec{D}$ represents the vector from the viewpoint to the center of the occluder (Figure 11). This "area-angle" metric captures several properties of the subtended solid angle of the polygon, making it a useful approximation. First, larger polygons have larger area-angle. Second, the area-angle falls as the square of the distance from the viewpoint, as does subtended angle. Third, maximum area-angle occurs when the viewing direction D is "head-on" with the occluder, and falls with the dot product as the occluder is viewed obliquely. While this metric differs from the solid-angle in that it does not consider the actual shape of the occluder, it is much simpler to compute, and serves as a useful heuristic to identify large occluders near the viewpoint.

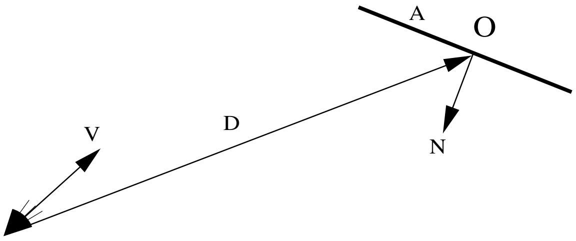


Figure 11: Parameters in the area-angle metric.

Given a discretization of all possible viewing directions, the occluder preprocessing algorithm uses the area-angle metric to associate each kD-tree leaf and viewing direction with those k occluders likely to be most effective from viewpoints in that leaf. The number of occluders k is a parameter supplied to the preprocessing stage.

When interactive model viewing begins, the algorithm locates the kD-tree leaf which contains the initial viewpoint. The algorithm then uses the set of k occluders associated with the kD-tree leaf and discrete viewing direction closest to the current viewing direction. When the viewpoint moves outside the kD-tree cell or the viewing direction changes substantially, the set of occluders is modified to correspond to the current viewing position and direction. Note that if the observer is moving smoothly, the set of occluders effective from the current viewpoint is likely to be similar to the set of occluders effective from the next viewpoint, and little or no changes need be made to the occluder set.

3.2 Detail Objects

Much of the polygon complexity in urban and architectural models arises due to the presence of small "detail" objects (e.g., furniture in a building, foliage in a city). As they are also part of the kD-tree hierarchy, these objects may be culled by the hierarchical culling algorithm. Objects that are not culled are subjected to additional tests. Each such object is tested against occluders that are "near" the object (Figure 12). These tests can cull objects even if there are no large occluders near the viewpoint.

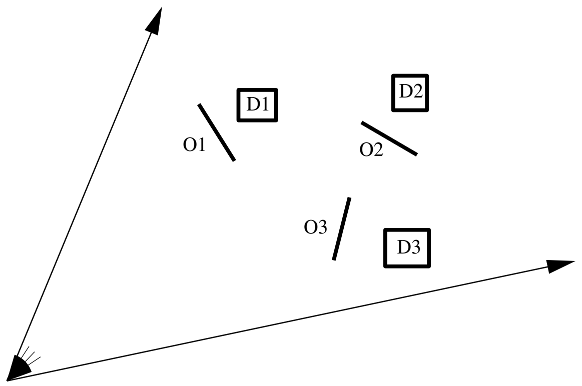


Figure 12: Detail objects $D1, D2, D3$ and occluders used to determine their visibility.

Occluders that are near detail objects can be determined as follows. First, as detail objects are usually limited in spatial extent, they can be usefully approximated as a single point (e.g., the center of the object). Second, the problem of determining if the detail object is visible from the viewpoint is identical to the problem of determining if the viewpoint is visible from the detail object (i.e., visibility between two points is *reflexive*). Thus, occluders that are large when viewed from the center of the object (in a viewing direction toward the viewpoint) are good candidates to test. These are identified and stored for each object using the data structure computed in Section 3.1. Of course, once these occluders are determined, it is necessary to perform the visibility test on the entire detail object using separating and supporting planes, instead of just on its midpoint.

4 Results

We have implemented the algorithm described above, as well as tools for visualizing its operation. For models consisting of a few thousand large occluders (and hundreds of additional detail objects), the algorithm maintains interactive rates on an SGI Onyx™ workstation (with a 250 MHz R4400 processor, 512 MB of main memory and an SGI RealityEngine2™ graphics pipeline), and culls a significant fraction of all models on average. Figure 13 (color plate) shows snapshots from several interactive walkthroughs.

We studied the performance of the algorithm on two models: the fifth floor of Berkeley's Soda Hall building with furniture (Soda) and a city from Viewpoint Datalabs (City). In our implementation, the kD-tree is constructed by splitting the current kD-tree node alternately in each of the three dimensions, and choosing a splitting plane that results in a roughly balanced partitioning of the polygons. The kD-tree is of height 8 (around 250 nodes) and the number of occluders is fixed at 32 per direction (for 8 directions) in these experiments. Initialization and kD-tree construction used about 5 seconds of CPU time. The results reported below are averaged over the viewpoints visited during smooth "figure 8" walkthroughs of the models.

Table 1 shows the efficacy of the culling algorithm. Occlusion culling reduces the rendering load (in terms of polygons drawn) by a factor of six to eight over frustum culling. The speedup obtained due to this reduction of rendering load depends on the performance of the hardware graphics pipeline – the lower the performance, the higher the benefit of occlusion culling. Table 2 shows the time spent in culling and drawing these two models; we report these times for the Onyx as well as an SGI Indigo^2 Elan™ workstation (with a 200 MHz R4400 processor), which has less powerful graphics hardware. Note that the algorithm reduces total rendering time by a factor of 2 on the Onyx, and by a factor of 5 on the Elan.

Scene	Polygons	Frustum	Occlusion
Soda	134,832	19.7	2.6
City	108,841	36.9	5.6

Table 1: The column Frustum shows the percentage of polygons drawn after only view frustum culling, and the column Occlusion shows the percentage of polygons drawn after frustum *and* occlusion culling.

Scene	Frustum			Occlusion		
	Cull	Draw	Total	Cull	Draw	Total
Onyx						
Soda	12	83	95	27	10	37
City	11	102	113	29	16	45
Elan						
Soda	13	435	448	32	57	89
City	12	482	494	34	77	101

Table 2: The columns labeled Frustum show culling and drawing times after only view frustum culling. The columns labeled Occlusion show culling and drawing times after frustum *and* occlusion culling. All times are reported in milliseconds.

4.1 Spatial and Temporal Coherence

The visibility algorithm maintains (and tests) only 27–35 kD-tree nodes per frame (30–40% of the kD-tree nodes in viewing frustum), reflecting the spatial coherence exploited by the algorithm. Its use of temporal coherence is indicated in Table 3, which shows the time spent in the algorithm *Visible* (excluding frustum culling) as the speed of the observer is varied. In this experiment, the observer moves along the "figure 8" path with different speeds. The algorithm spends lesser time for more slow moving observers, reflecting the temporal coherence exploited by the algorithm.

Scene	Time in *Visible* (msec)						
	0.25x	0.5x	1x	2x	4x	8x	16x
Soda	17	18	21	25	32	45	56
City	19	21	24	29	35	45	55

Table 3: Time spent in visibility processing for an observer moving at increasing speed. The speeds are given as multiples of the slowest speed, which corresponds to "walking" speed in Tables 1 and 2.

5 Conclusion

This paper describes an efficient occlusion culling algorithm that exploits the presence of large occluders in urban and architectural models, and culls a significant fraction of two test scenes. The algorithm is conservative in that it uses only simple object-space tests to detect occlusion. By organizing the polygons in a kD-tree, it exploits spatial coherence. By caching occlusion relations and large occluders across viewpoints, it exploits temporal coherence in the motion of the observer.

In future work, it would be interesting to apply techniques developed in this paper for models that do not contain large occluders (e.g., CAD models of airplanes or submarines). One promising approach is to construct "fictitious" occluders that conservatively approximate the occlusion caused by a large mesh of triangles. Also, we are investigating strategies for adaptively choosing k, the size of the occluder set dynamically maintained by the visibility algorithm.

Another interesting area of future research is integration of occlusion culling techniques with texture based approximation [11, 13, 16]. Finally, occlusion culling techniques presented here (especially those based on table lookup) may be amenable to hardware implementation, yielding further speedups.

References

[1] Airey, J. M., Rohlf, J. H., and Brooks, Jr., F. P. Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments. *ACM Siggraph Special Issue on 1990 Symposium on Interactive 3D Graphics 24*, 2 (1990), 41–50.

[2] Akeley, K. RealityEngine Graphics. *SIGGRAPH '93 Conference Proceedings* (1993), 109–116.

[3] Bentley, J. Multidimensional binary search trees used for associative searching. *Communications of the ACM 18* (1975), 509–517.

[4] Catmull, E. E. *A Subdivision Algorithm for Computer Display of Curved Surfaces.* PhD thesis, University of Utah, Dec. 1974.

[5] Coorg, S., and Teller, S. Temporally Coherent Conservative Visibility. In *Proc. 12^{th} Annual ACM Symposium on Computational Geometry* (1996), pp. 78–87.

[6] Funkhouser, T., Séquin, C., and Teller, S. Management of Large Amounts of Data in Interactive Building Walkthroughs. In *Proc. 1992 Workshop on Interactive 3D Graphics* (1992), pp. 11–20.

[7] Garlick, B., Baum, D. R., and Winget, J. M. Interactive Viewing of Large Geometric Databases Using Multiprocessor Graphics Workstations. *Siggraph '90 Course Notes (Parallel Algorithms and Architectures for 3D Image Generation)* (1990).

[8] Greene, N., and Kass, M. Error-Bounded Antialiased Rendering of Complex Environments. In *SIGGRAPH '94 Conference Proceedings* (1994), pp. 59–66.

[9] Greene, N., Kass, M., and Miller, G. Hierarchical Z-Buffer Visibility. In *SIGGRAPH '93 Conference Proceedings* (1993), pp. 231–240.

[10] Luebke, D., and Georges, C. Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets. In *Proc. 1995 Symposium on Interactive 3D Graphics* (1995), pp. 105–106.

[11] Maciel, P. W. C., and Shirley, P. Visual Navigation of Large Environments Using Textured Clusters. In *Proc. 1995 Symposium on Interactive 3D Graphics* (1995), pp. 95–102.

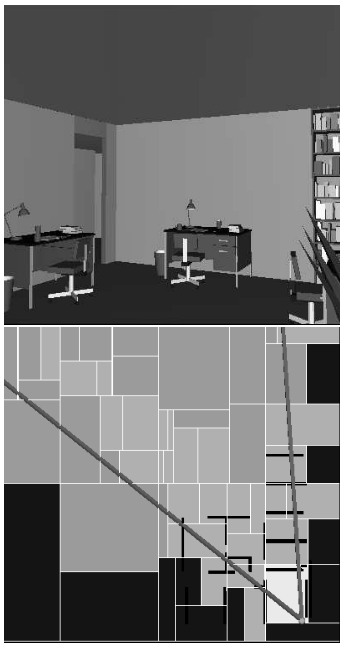
[12] Naylor, B. F. Partitioning Tree Image Representation and Generation from 3D geometric models. In *Proc. Graphics Interface '92* (1992), pp. 201–211.

[13] Shade, J., Lischinski, D., Salesin, D., DeRose, T., and Snyder, J. Hierarchical Image Caching for Accelerated Walkthroughs of Complex Environments. In *SIGGRAPH '96 Conference Proceedings* (1996), pp. 75–82.

[14] Sutherland, I. E., Sproull, R. F., and Schumacker, R. A. A Characterization of Ten Hidden-Surface Algorithms. *Computing Surveys 6*, 1 (1974), 1–55.

[15] Teller, S., and Séquin, C. H. Visibility Preprocessing for Interactive Walkthroughs. *SIGGRAPH '91 Conference Proceedings* (1991), 61–69.

[16] Xiong, R. A Stratified Rendering Algorithm for Virtual Walkthroughs of Large Environments. Masters Thesis, EECS Department, MIT, May 1996.



White:	Node borders
Yellow:	Current leaf
Black:	Occluders
Dark Blue:	Outside frustum
Green:	Culled
Light Blue:	Drawn

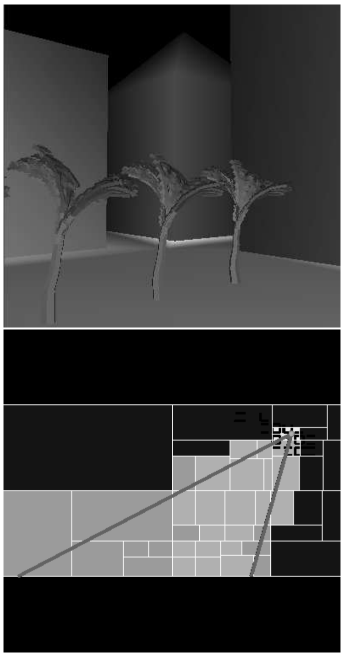


Figure 13: Frames from walkthroughs of Soda (left) and City (right) with culling visualization.