A Spatially and Temporally Coherent Object Space Visibility Algorithm

SATYAN COORG SETH TELLER
Synthetic Imagery Group
MIT Laboratory for Computer Science
Cambridge, MA 02139
{satyan,seth}@lcs.mit.edu

Abstract

Efficiently identifying polygons that are visible from a changing synthetic viewpoint is an important problem in computer graphics. In many complex geometric models, most parts of the model are invisible from the instantaneous viewpoint. Despite this, hidden-surface algorithms like the z-buffer or BSP tree often expend significant computation resources processing invisible portions of the model.

In this paper, we present a new approach to the visibility problem that exploits the presence of large occluders near the viewpoint to identify a *superset* of visible polygons, without touching most invisible polygons. The salient features of this algorithm are: it is *conservative*, *i.e.*, it overestimates the set of visible polygons; it exploits *spatial coherence* by using a hierarchical data structure; it exploits *temporal coherence* by reusing visibility information computed for previous viewpoints; and it is easily *parallelizable*.

Keywords: Conservative visibility, temporal coherence, spatial coherence, octrees, multi-processing.

1 Introduction

In computer graphics, identifying visible polygons or eliminating hidden polygons is an important component of efficient scene rendering algorithms. Despite the availability of the z-buffer algorithm in hardware [2], the number of polygons in many geometric models is larger than hardware alone can process at interactive frame rates. One way to address this problem is by developing algorithms that efficiently identify, then render, only the visible portions of the model. Synthetic observers in visual simulation applications typically move smoothly through the model, experiencing little change in visibility between viewpoints. There is thus ample spatial and temporal coherence to be exploited in most such applications. Algorithms that exploit the availability of the z-buffer by *overestimating* the set of visible polygons are useful, if they run in time comparable to the rendering time, and produce supersets which are only slightly larger than the true visibility.

Due to the availability of cheap memory and its simplicity, the z-buffer algorithm, typically implemented in hardware, is widely used for visibility determination. Each polygon in the model is *rasterized* into fragments, and z-values are computed for each fragment from the z-values of the polygon's vertices. This algorithm resolves visibility at each pixel by comparing the new fragment's z-value to that stored at the pixel, writing a new color value to the framebuffer only if the fragment is closer to the viewpoint than the stored value. By structuring the computation of z-values as a "slow" set-up computation per polygon and a "fast" interpolation per pixel, this algorithm makes good use of image space coherence.

However, the disadvantages of resolving visibility at this late stage are that expensive operations like shading and texturing are performed even on invisible fragments, and there is no obvious way to exploit the presence of large occluders near the observer.

Ray tracing and ray casting methods resolve visibility in object space by casting a ray through each pixel. That polygon which first intersects the ray is visible at the pixel. Spatial subdivisions have been used to exploit spatial coherence and accelerate ray tracing [12, 6]. Algorithms that exploit temporal coherence in this context have also been described [15, 6]. However, a major drawback of these algorithms is that a large number of rays must be cast to generate each image, making them impractical for current interactive display systems.

In this paper, we describe a novel algorithm that maintains a set of large occluders near the viewpoint, and uses the set to cull away occluded portions of the model. The algorithm operates entirely in object space, using fast (conservative) visibility tests. It uses an octree-based spatial subdivision, and eliminates large portions of the model without touching most invisible polygons. The algorithm works by classifying each octree node as invisible, visible, or partially occluded with respect to the set of occluders. Once the state of each octree node is known, a final traversal issues visible and partially visible polygons to the rendering hardware for per-pixel visibility determination.

The rest of the paper is organized as follows. The remainder of this section discusses existing research work in visibility algorithms. Section 2 describes the conservative visibility tests used in our algorithm. Section 3 describes the visibility algorithm, based on octrees, that efficiently detects occlusion and maintains visibility information for a moving viewpoint. Section 4 presents the algorithm's implementation and performance characteristics. Section 5 concludes and discusses future work.

1.1 Related Work

In principle, for static models, it is possible to precompute the visibility from every viewpoint, and use the precomputed values to render the model during an interactive walkthrough. The well-known aspect graph encodes a representation of exact visibility for every qualitatively distinct region of viewpoints [17, 11, 10]. One drawback of this approach is that the visible portion of the scene may have higher complexity than the input scene itself – a scene with $O(n)$ polygons can have a visible portion with descriptive complexity $O(n^2)$. Also, the large number of qualitatively distinct viewpoint regions, $O(n^9)$ in the worst case, makes this exact approach impractical for complex models.

Given the availability of fast z-buffer hardware to resolve visibility per-pixel, it seems promising to design algorithms that *overestimate* the polygons visible from any specified viewpoint. The output of such an algorithm could then be fed into the z-buffer to synthesize a final image. The overestimation guarantees that the generated image is identical to that which would be generated by rendering every polygon in the scene. The challenge, of course, is to produce a usefully tight upper bound on the visible polygons. This idea of "conservative" visibility has been exploited to design fast architectural walkthrough systems [1, 20, 8]. The idea in [20] is that the input scene can be divided into *cells*, roughly corresponding to rooms in a building, and *cell-to-cell/eye-to-cell* visibility can be used 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 seems difficult.

An approach using octree-based spatial subdivision is used in [9] to render only those polygons that lie within the viewing frustum. However, this algorithm does not exploit occlusion properties of the model, and can end up drawing many invisible polygons.

The hierarchical z-buffer algorithm [13] makes some use of *temporal* coherence by maintaining a list of polygons that are visible from the current viewpoint. For the next viewpoint, the algorithm draws polygons from this list first. The contents of the hierarchical z-buffer can then be used to *cull* invisible polygons. To exploit spatial coherence, this algorithm requires that the z-buffer support visibility queries. Such queries are not efficiently supported in most graphics hardware, and simulating the z-buffer in software involves significant overhead. Because of its reliance on image space queries, this algorithm is also susceptible to aliasing artifacts.

A temporally coherent visibility algorithm for scenes comprised only of convex objects is presented in [14]. The algorithm is complex, as it computes exact visibility, and no implementation is described.

A dynamic temporally coherent conservative visibility algorithm, described in [7], 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.

Our work represents an advance over existing efforts in the following respects. First, our algorithm exploits the availability of fast z-buffers by generating conservative (superset) visibility, greatly simplifying the algorithmics of visibility determination. Second, we use object hierarchies to avoid processing large sets of invisible polygons or their pairwise interactions. Third, our algorithm is naturally parallel, yielding an efficient implementation on multi-processors. Finally, our algorithm handles an observer moving along an arbitrary (not prespecified) path, and incorporates advanced culling features such as frustum-based culling and occlusion due to multiple connected occluders.

2 Conservative Visibility

We assume that the input model is static, and is specified as a list of convex polygons. We assume 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; the resulting connectivity information is represented by a winged-edge data structure [3]. We assume no *a priori* knowledge of observer motion.

In this paper, we distinguish *occluder* polygons from *occludee* objects. This distinction is motivated by the observation that, in many models, from any instantaneous viewpoint, a few polygons cause most of the occlusion, and checking other polygons for occlusion increases the overhead of the algorithm, without identifying many more occluded polygons. Crucially, occluders are chosen dynamically, as the viewpoint changes. Polygons typically act as occluders for nearby viewpoints, and as occludees for remote or oblique viewpoints (Figure 1). Usually, an occludee is not just a single polygon, but a convex region of space (*e.g.*, a hierarchical bounding box) containing many polygons.

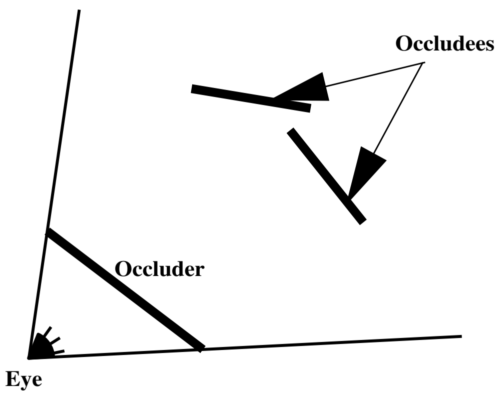

Figure 1: This figures shows occluder and occludee polygons for a viewpoint.

We present the algorithm in two parts. First, we describe algorithms for conservative visibility determination. These algorithms are then used as the basis of a visibility oracle in the dynamic algorithm presented in Section 3.

2.1 Visibility Testing

In this section, we address the following problem: given a viewpoint, a set of convex occluders and a convex occludee, is the occludee visible? That is, is there a line segment from the viewpoint to some point on the occludee that meets no occluder? A naive algorithm would perform a computation equivalent to projecting the occluders and occludee onto the image plane, then "clipping" the images of the occluders away from that of the occludee (Figure 2). The occludee is invisible if and only if it is clipped away completely by the set of occluders. This process must be repeated for each new viewpoint, making this approach inefficient in practice.

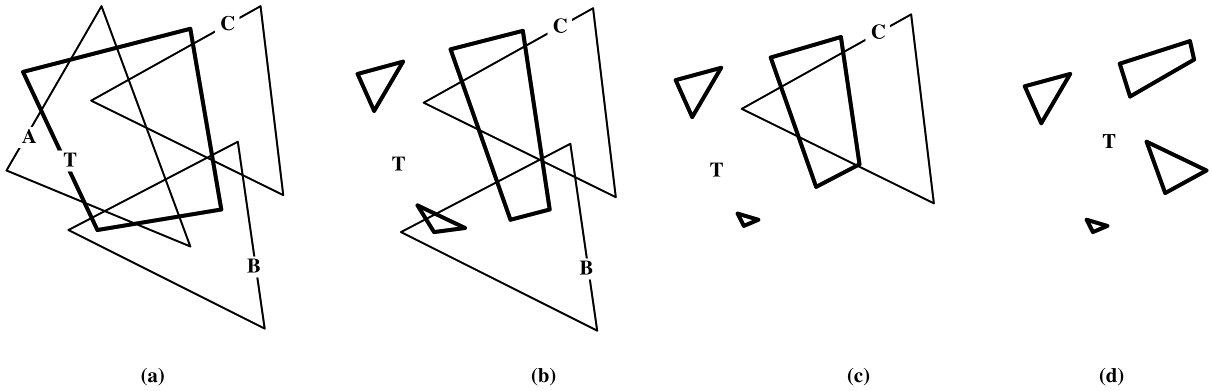

Figure 2: The figure shows a naive way of detecting occlusion. The polygons in Figure (a) are shown as they would appear in the image, *i.e.*, the reader is at the viewpoint. The occludee T is behind all the occluders A, B, and C. Figures (b)–(d) show successive clipping of the occludee to each occluder to determine its visibility.

Instead, we describe two simple (and fast) tests that conservatively guarantee the invisibility of an occludee. Our methods use the notion of *supporting* and *separating* planes. Separating planes of two convex polyhedra are planes formed by an edge of one polyhedron and a vertex of the other such that the polyhedra lie on opposite sides of the plane [18]. Supporting planes are analogous, except that both polyhedra lie on the same side of the plane.

Figure 3 shows the three ways in which a set of occluders can complete occluded an object. Figure 3-a shows the simplest case, when the occlusion is caused by a single occluder. Figure 3-b shows occlusion caused by multiple connected occluders, and Figure 3-c shows the most general case, in which occlusion occurs due to arbitrary disconnected occluders.

First, consider the interaction between a single occluder and an occludee, which can be completely described in terms of the plane of the occluder, and the supporting and separating planes of the occluder and occludee (Figure 4-a). First, the occluder A can occlude T only if the viewpoint and T are on opposite sides of the plane of polygon A. This region can be divided into three qualitatively distinct regions as shown in the figure. In region 1, T is not occluded. In region 2, T is partially occluded, and in region 3, T is completely occluded.

The supporting and separating planes of A and T can be used to detect which of these cases is occurring. First, the planes are oriented "towards" the occluder to form half-spaces (Figure 4-a). We

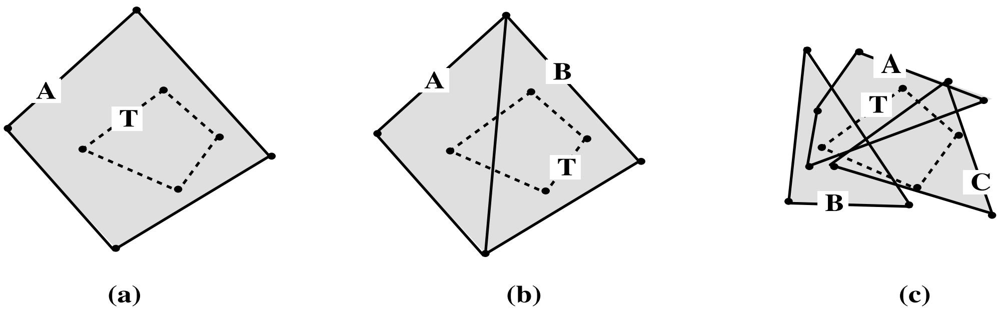

Figure 3: Occlusion caused by a single occluder (a). Occlusion caused by connected occluders (b). Occlusion caused by a set of (general) occluders (c). The polygons are shown as they would appear in the image.

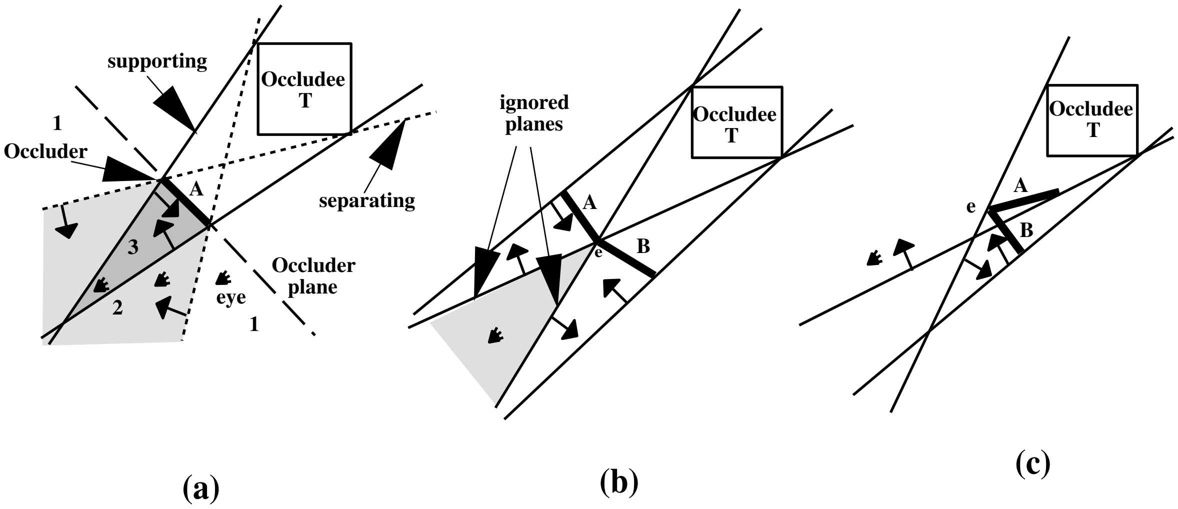

Figure 4: Figure (a)–(c) show occlusion in two dimensions. Figure (a) shows separating and supporting planes of a single occluder. Figure (b) shows supporting planes that are ignored for two connected occluders. Figure (c) shows a common edge which is also a silhouette edge; thus its supporting plane cannot be ignored.

say that a viewpoint *satisfies* a plane if it is inside the plane's positive halfspace; 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 halfspaces (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 4-b shows occlusion caused by two connected occluders, *i.e.*, two occluders that share a vertex (an edge in 3D). If the viewpoint is in the shaded region, T is occluded by both A and B, even though neither occludes it alone. Testing for occlusion by A and B in the shaded region is equivalent to checking all the supporting planes of A and B except those through e. Note that this is true 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 silhoutte edge of the occluder as seen from the viewpoint. Figure 4-c shows a case in which ignoring the supporting plane 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
- All planes other than those supporting common edges are satisfied.

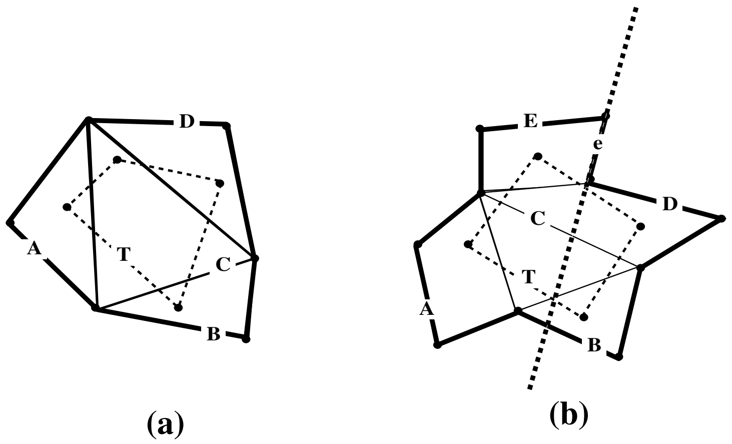

Figure 5: This figure shows occlusion in 3D, as seen from the viewpoint. Figure (a) shows occlusion caused by connected occluders whose silhouette is convex in the image. Figure (b) shows occlusion by connected occluders having a non-convex silhouette.

Figure 5-a shows occlusion in 3D by a set of connected occluders. The supporting planes through "internal" edges can be ignored, and full occlusion is caused when the viewpoint satisfies all the supporting planes through silhouette edges. Since the silhouette edges form a convex polygon in the image, the supporting planes through them are satisfied if and only if the image of the occludee lies entirely within the convex silhouette, *i.e.*, when there is full occlusion. Thus, checking for inclusion within the supporting planes through silhouette edges is an exact test for occlusion.

However, this test can fail if the silhouette edges form a non-convex polygon (Figure 5-b). Consider the silhouette edge e shown in the figure. The supporting plane of T through e is not satisfied, as T does not lie completely to the "left" of the line through e. Thus, the above test fails to conclude that T is fully occluded. Again, the test is conservative – it will never misclassify a visible or partially visible polygon as invisible. Our algorithm defines conservative visibility operationally as:

Definition 1 (Conservative Visibility) *A convex occludee is* invisible *with respect to a viewpoint and a set of occluders if either*

- *The occludee is completely occluded by a single occluder in the set, or*
- *The occludee is occluded by a set of connected occluders such that the supporting planes of their silhouette edges are satisfied.*

This definition precludes the algorithm from detecting more complex occlusions such as in Figure 3-c. This is a reasonable tradeoff for models in which most occlusion is due to large occluders acting alone or with a number of connected occluders.

The final algorithm for visibility testing is given below. It classifies the occludee as VISIBLE, INVISIBLE, or PARTIAL.

```
TestOcclusion(OccluderSet S, Occludee T, Viewpoint P)
    partial = FALSE
    for each A_i ∈ S do                         // compute occlusion relations
        compute separating and supporting planes of A_i and T
        check for full and partial occlusion by A_i
        if (A_i fully occludes T) then
            return INVISIBLE
        else if (A_i partially occludes T)
            partial = TRUE
    combine occluders in S into sets of connected occluders
    if (a connected set fully occludes T) then
        return INVISIBLE
    else if (partial) then
        return PARTIAL
    else
        return VISIBLE;
```

The two major steps in the algorithm are the computation of separating and supporting planes of occluders and the occludee, and the detection of connected sets of occluders. The rest of the algorithm involves computing dot products of planes, and is straightforward. Connected sets of occluders are detected using the connectivity information in the winged-edge data structure. Enumeration of the separating/supporting planes is described below.

First, the algorithm considers separating/supporting planes only through edges of the occluders and vertices of the occludee. Checking only these planes is an exact test for full occlusion by a single occluder, but a conservative one for partial occlusion. Second, for a convex occluder, this computation can be performed by computing separating/supporting planes through each occluder edge. Third, the occludees in our visibility queries are axial octree nodes; computing their separating/supporting planes with respect to any occluder edge can be implemented as a fast table lookup.

The time spent in each step of *TestOcclusion* is proportional to the number of occluder edges, which is $O(k)$, where k is the number of occluders involved in the query.

3 The Visibility Algorithm

Section 2 described techniques to efficiently classify the occlusion relationships between a set of occluders and an occludee. For n polygons, individually maintaining each of the $O(n^2)$ interactions is too expensive. It is also wasteful, since most polygons do not occlude each other from the instantaneous viewpoint. Thus, it is advantageous to maintain relations only between polygons that are actually occluding or likely to occlude in the near future. This section describes our visibility algorithm, which is based an efficient hierarchical method for maintaining PARTIAL and INVISIBLE relationships.

We first turn to the problem of identifying those polygons visible from a specified viewpoint in the presence of a set of occluders. Our approach organizes all polygons in an octree [19]. A simple way to build an octree is to associate the bounding box containing all the polygons in the scene with the root node of the octree, and then recursively subdivide until some termination criterion is satisfied (*e.g.*, the number of polygons in each leaf is less than some constant or the dimensions of the octree node are less than some constant).

Given an octree, the following algorithm reports those polygons in the octree that are *not* occluded by the specified occluders. In the algorithm, *Gather*(T, OS) simply collects all polygons reachable from an octree node T and unions them to the set OS.

```
Visible(OccluderSet S, Octree Node T, Viewpoint P, ObjectSet OS)
    state = TestOcclusion(S, T, P);
    if (state == VISIBLE) then                // all of subtree T is visible; report
        Gather(T,OS);
    else if (state == INVISIBLE) then         // all of subtree T is invisible; omit
        return;
    else if (state == PARTIAL) then
        if T is a leaf then
            Gather(T,OS);                     // conservatively return all polygons 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)
```

The first step in algorithm *Visible* is to classify the top-level octree node with respect to the set of occluders. If the state of the octree node is VISIBLE, the algorithm reports all polygons in the subtree as visible *without* performing any further tests. If the state of the octree node is INVISIBLE, the algorithm simply returns, without traversing the (invisible) descendant octree nodes. The algorithm recurses on the octree node's children only if the node's state is PARTIAL. Even then, it recurses only using the (typically smaller) set S' of occluders that partially occlude the octree node, since only these occluders can occlude any descendant octree node.

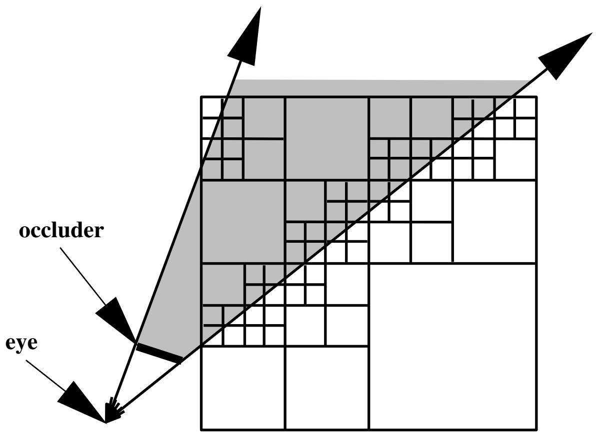

Figure 6: The octree nodes visited by the algorithm *Visible* (unshaded nodes are reported visible). For clarity of the figure, we assume the occluder is not in the octree.

Figure 6 shows the set of octree nodes visited by *Visible* with a single occluder as input. Note the avoidance of processing of large portions of the octree, if the octree node is either completely occluded or not occluded at all.

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

3.1 Extensions for a Moving Observer

A naive way of extending algorithm *Visible* for a moving observer would be to apply it independently to successive viewpoints. However, the temporally coherent nature of observer motion can be exploited to design a more efficient approach. For each viewpoint, we *cache* the occlusion relations – a list of supporting and separating planes – at each visited octree node. When the viewpoint changes, the algorithm needs only *check* existing occlusion relations, updating only those octree nodes whose visibility status has changed.

Note that this modification exploits temporal coherence in two ways. First, for nodes whose states are identical for both the previous and current viewpoint, the algorithm only checks node state (by computing inner products of the viewpoint with a list of plane equations); it does not recompute the separating/supporting planes. Second, since the algorithm's complexity depends on the number of nodes whose state has changed, it runs faster when fewer visibility changes occur in the scene – for example, when the observer moves slowly, or encounters a model region of relatively low visual complexity.

Finally, note that frustum culling [9] can be incorporated into the algorithm in a straightforward way. The algorithm achieves this by *lazily* processing octree nodes. Only octree nodes that intersect the viewing frustum are processed. For octree nodes that are outside the frustum, the set of occluders that obscure such a node are retained, but no processing is performed on either the octree node or its descendants. When a node that was previously outside is found to lie inside the viewing frustum, the set of occluders stored at the octree node is used to determine the state of the octree node and its descendants. Essentially, the node is "switched off" whenever it lies outside the instantaneous viewing frustum; in this state, a fast rejection of the octree node is performed and its occlusion status is not computed.

3.2 Parallelism

The algorithm *Visible* has natural parallelism. The parallel version of the algorithm exploits the fact that unrelated octree nodes (*i.e.*, nodes which are neither ancestors nor descendants of eachother) can be processed independently.

A naive way of parallelizing algorithm *Visible* to run on m processors would be to assign m independent subtrees of the octree to separate processors, with each processor responsible for updating the status of all octree nodes in its subtree. However, in practice, this partitioning suffers from significant load imbalance, as the work is unevenly partitioned across the processors.

To achieve load balancing, we have implemented a version of the work-stealing scheduling algorithm described in [5], which is particularly well suited to independent subtree traversal. Each tree node is packaged into a single unit of work. Processors are assigned a single unit at a time, moving to another after they have completed processing their unit.

Unfinished work resides on per-processor queues. When a processor finishes its current unit, it chooses to work on one of the current octree node's children, and deposits the other children at the *tail* of its queue (Figure 7). When the processor runs out of work, it removes the work unit at the *tail* of its work queue, if it is non-empty. Otherwise, it picks another processor at random and *steals* work from the *head* of that processor's work queue.

This technique has several advantages with respect to load balancing. First, it ensures that a single

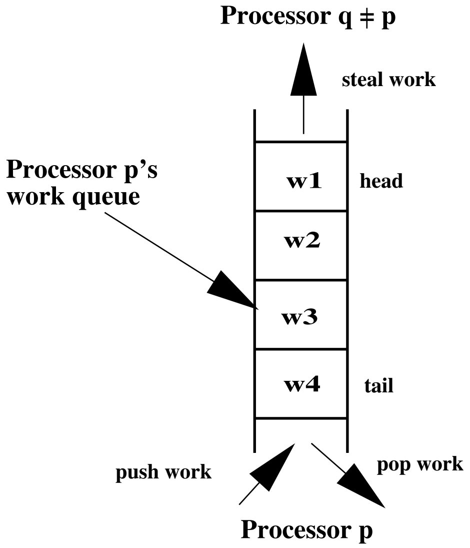

Figure 7: The per-processor load balancing work queue.

processor works in a depth first fashion, leaving sibling octree nodes to be stolen. Second, a *steal* by another processor from the head of the queue yields a large "chunk" of work; such nodes are likely to be at higher levels of the octree. In this way, work is distributed evenly across the processors avoiding both congestion at a single queue or starving of processors. Using this technique, we have been able to achieve good load balance among the processors (Section 4).

3.3 Dynamic Occluder Selection

The algorithm *Visible* described earlier maintains the state of an octree with respect to a *fixed* set of occluders. As the viewpoint moves, it is crucial to update the occluder set to contain those polygons that are "large" in the image, and therefore likely to occlude substantial portions of the model. This section describes the updating heuristics in our system; intuitively, small polygons are discarded from the occluder set, whereas polygons that loom large, if not present, are added.

A simple metric for the usefulness of a polygon as an occluder is its subtended solid angle. A reasonable estimate of this is the quantity

$$\frac{\vec{A} \cdot \hat{D}}{|\vec{D}|^2}$$

(called *area-angle*) where A represents the (directed) area of the occluder and $\vec{D}$ represents the vector from the viewpoint to the center of the occluder (Figure 8-a). The area-angle metric captures several properties of the subtended solid angle of the polygon, making it a useful approximation. First, larger polygons have a larger area-angle. Second, 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.

One simple way of using the area-angle metric would be to statically precompute the set of occluders with maximum area-angle for every possible viewpoint. However, this approach involves significant preprocessing time and storage. Instead, our system employs a combination of static preprocessing and dynamic occluder selection.

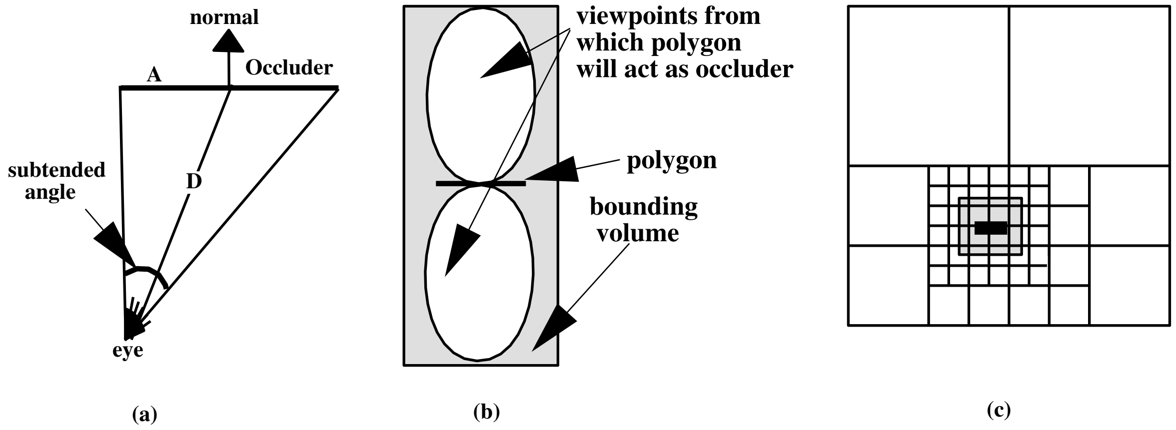

Figure 8: The subtended angle for a 2D occluder, and the components of the area-angle metric (a). The region of space in which the occluder is significant, i.e., where area-angle > threshold (b). Static insertion of an approximate occluder volume into the octree (c).

The idea is to associate each octree leaf with those occluders that are likely to be most effective from viewpoints in the leaf. First, a minimum threshold value for area-angle is chosen. The set of viewpoints such that the area-angle of a given polygon is greater than the threshold value is some region in 3-dimensional space (Figure 8-b). This viewpoint region is approximated by an axial bounding volume. The polygon is then associated with all octree leaves that intersect this volume (Figure 8-c). This process is performed for each input polygon, after which each octree leaf contains a list of (pointers to) occluders.

As interactive model viewing begins, the algorithm locates the octree leaf which contains the initial viewpoint. This octree leaf indicates a set of *potential* occluders; k *actual* occluders are chosen among them by computing area-angle for each occluder, and picking those k occluders with maximum area-angle (k is a parameter to the real-time system). When the viewpoint moves, the area-angles of both *potential* and *actual* occluders are updated; a *potential* occluder becomes an *actual* occluder if its area-angle value becomes one of the k largest, and vice-versa. When the viewpoint crosses into a new octree leaf, the set of *potential* occluders is reinitialized to the set associated with the newly entered leaf.

4 Implementation and Results

We have implemented the algorithms described in this paper inside the SGI OpenInventor toolkit, to run on a SGI Onyx workstation with four 250 MHz R4400 processors and 512 MB of physical memory. Excluding toolkits, the entire system comprises about 5000 lines of C++ code. The timings presented in this section reflect only time spent in the visibility algorithm, and exclude time required to draw the visible polygons.

We studied the performance of the algorithm on the following models: the fifth floor of Berkeley's Soda Hall building (Soda); a city from Viewpoint DataLabs (City); and a number of tetrahedra on a plane (Tetra). Though we described the algorithm in terms of octrees, the actual implementation used kD-trees [4], a similar hierarchical data structure. The number of occluders to be maintained dynamically is a program input parameter; we set it at 32 to achieve interactive frame rates, and are experimenting with heuristics for choosing it automatically. Figures 9, 10, and 11 show screen snapshots from the system in action.

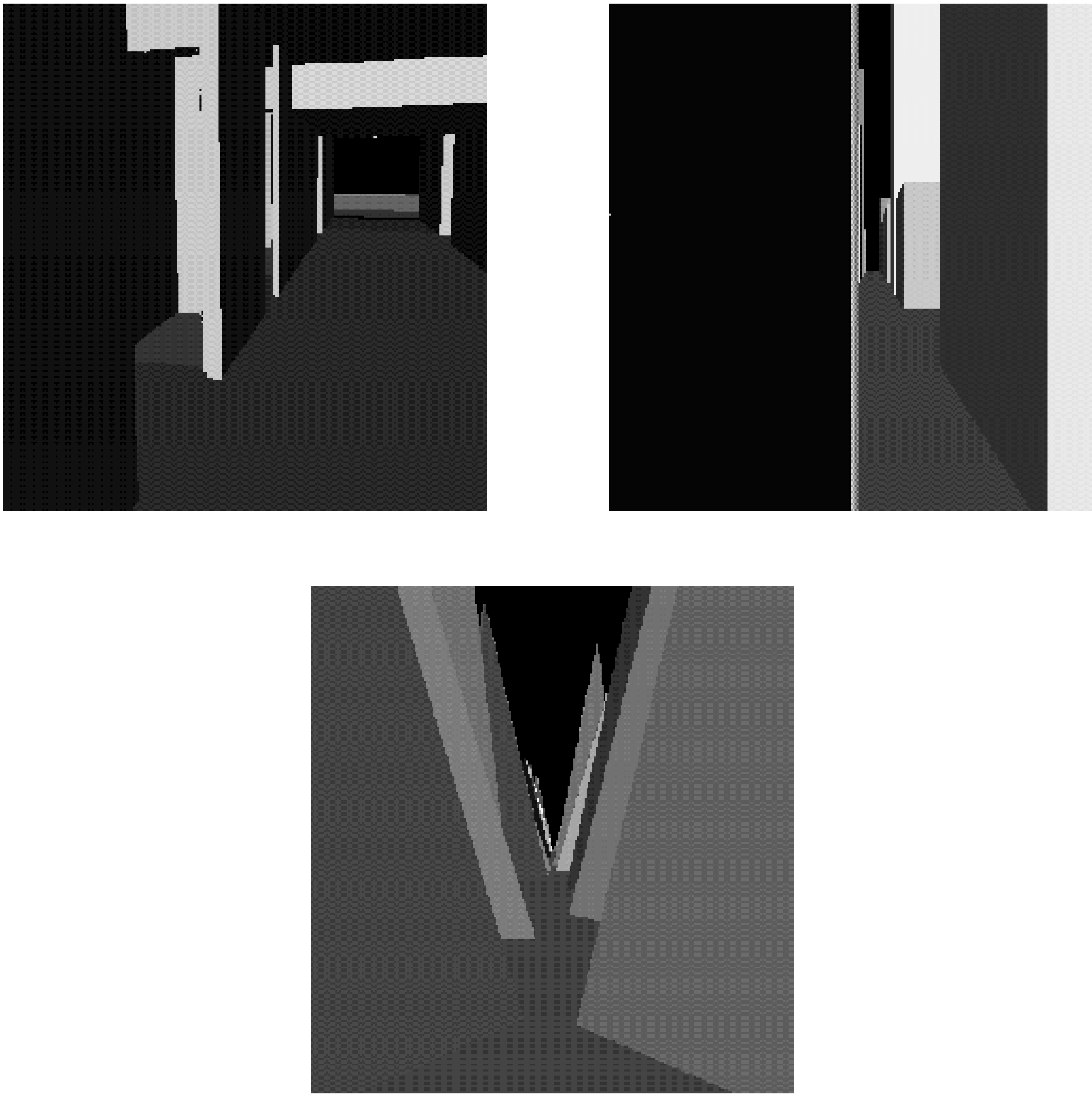

Figure 9: The three models used in our experiments: Soda, City and Tetra.

4.1 Input and Initialization

Scene	Polygon References		Construction	Initialization
	Initial	Final	(msec)	(msec)
Soda	1685	2971	160	510
City	2857	4722	270	480
Tetra	3601	4974	260	740

Table 1: The column *Final* denotes the total number of polygon references after octree construction. *Construction* shows the time spent in octree construction, and *Initialization* shows the time spent initializing the algorithm to the first viewpoint.

Table 1 shows some characteristics of the models, which contain only major occluding features such as exterior walls. The table also shows time spent in octree construction and initialization (including occluder area-angle preprocessing).

4.2 Comparison to Exact Visibility

Scene	Occ	Single Occluder			Multiple Occluders		
	(msec)	Rel	Vis (msec)	Cull	Rel	Vis (msec)	Cull
Soda	4.7	40	20	0.73	38	18	0.75
City	4.5	36	23	0.35	34	22	0.38
Tetra	4.6	57	32	0.69	56	32	0.69

Table 2: In the table, *Occ* shows the time spent on occluder processing at every viewpoint. *Rel* shows the number of octree nodes tested per occluder. *Vis* shows the time spent in the algorithm *Visible* for each change in viewpoint. *Cull* shows the efficacy of the culling algorithm as a fraction of (exact) invisible polygons, as reported by the hardware z-buffer – that is, as a fraction of the number of polygons that would be culled by a "perfect" visibility algorithm.

Table 2 illustrates the efficacy and efficiency of two conservative visibility tests. The observer has a full spherical field of view, and moves smoothly through the model at walking speeds in model units. Using z-buffer hardware, we computed the exact set of visible polygons from each viewpoint (and its complement, the exact set of invisible polygons). The values shown in the table reflect averages over many random walks, and tabulate the algorithm's effectiveness (35 - 75%) in identifying truly invisible polygons – that is, those that did not contribute any pixels to the rendered image. The algorithm culls a significant fraction of invisible polygons, even for scenes like City and Tetra, where the cell/portal technique of [20, 8] is less effective due to lack of obvious cell structure, and the presence of non-axial polygons.

The relatively lower performance of the algorithm on the City model is due to the existence of viewpoints from which most occluders are far away, and from which no single or connected occluders substantially occlude the model. However, for many models our algorithm will combine well with techniques that visually approximate remote polygons (e.g. by substituting textures [16]).

An interesting observation is that the algorithm *Visible* performs better in terms of *both* culling efficacy and efficiency when it uses the (more complex) connected occluder visibility test. This is due

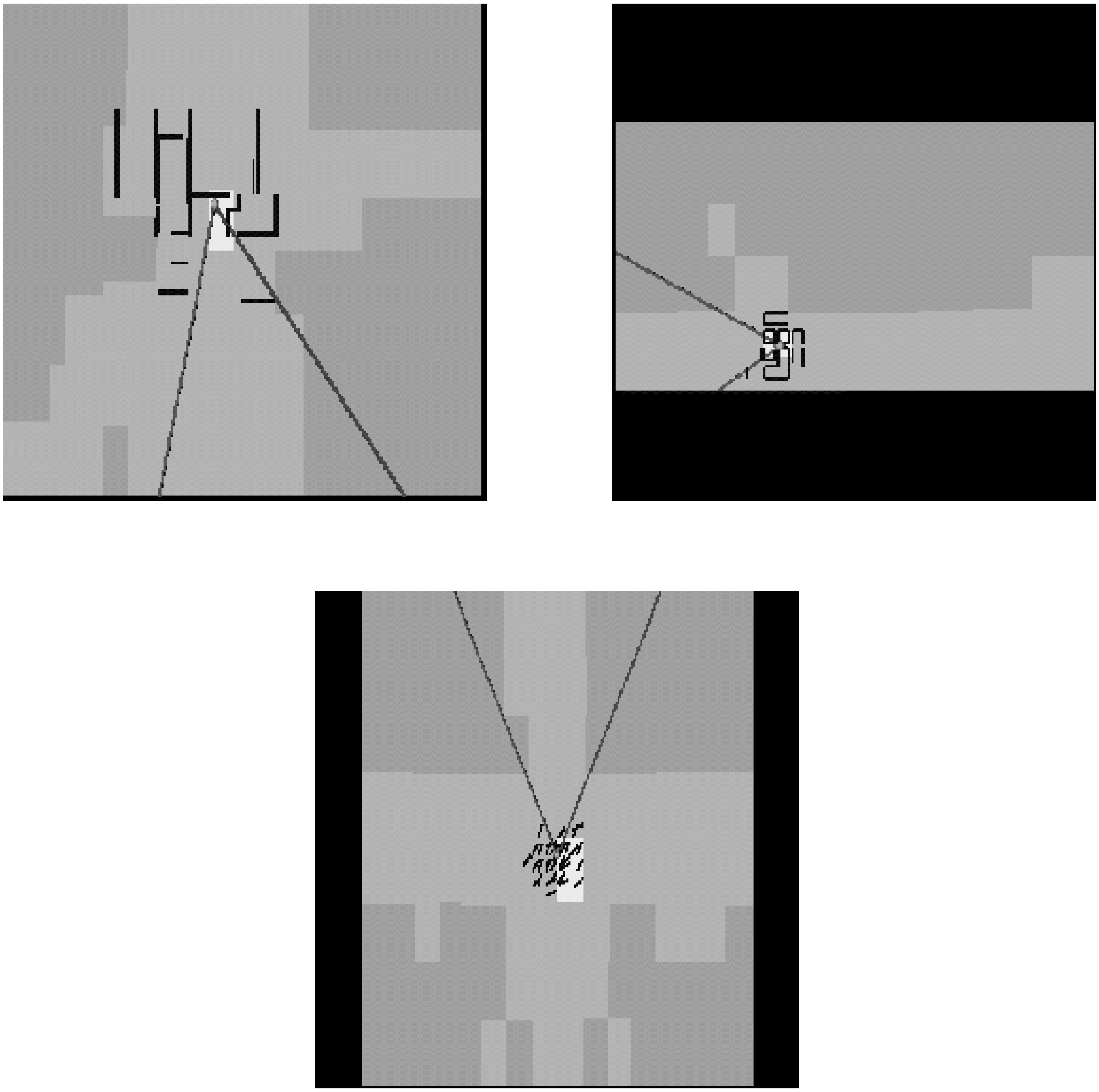

Figure 10: Position-based culling performed by our system. Top views of the three models are shown for the same viewpoints as in the previous figure. Green signifies the model portions classified as INVISIBLE; cyan signifies the model portions classified as PARTIAL or VISIBLE. Black lines denote the active set of occluders. The octree leaf containing the current viewpoint is shown in yellow.

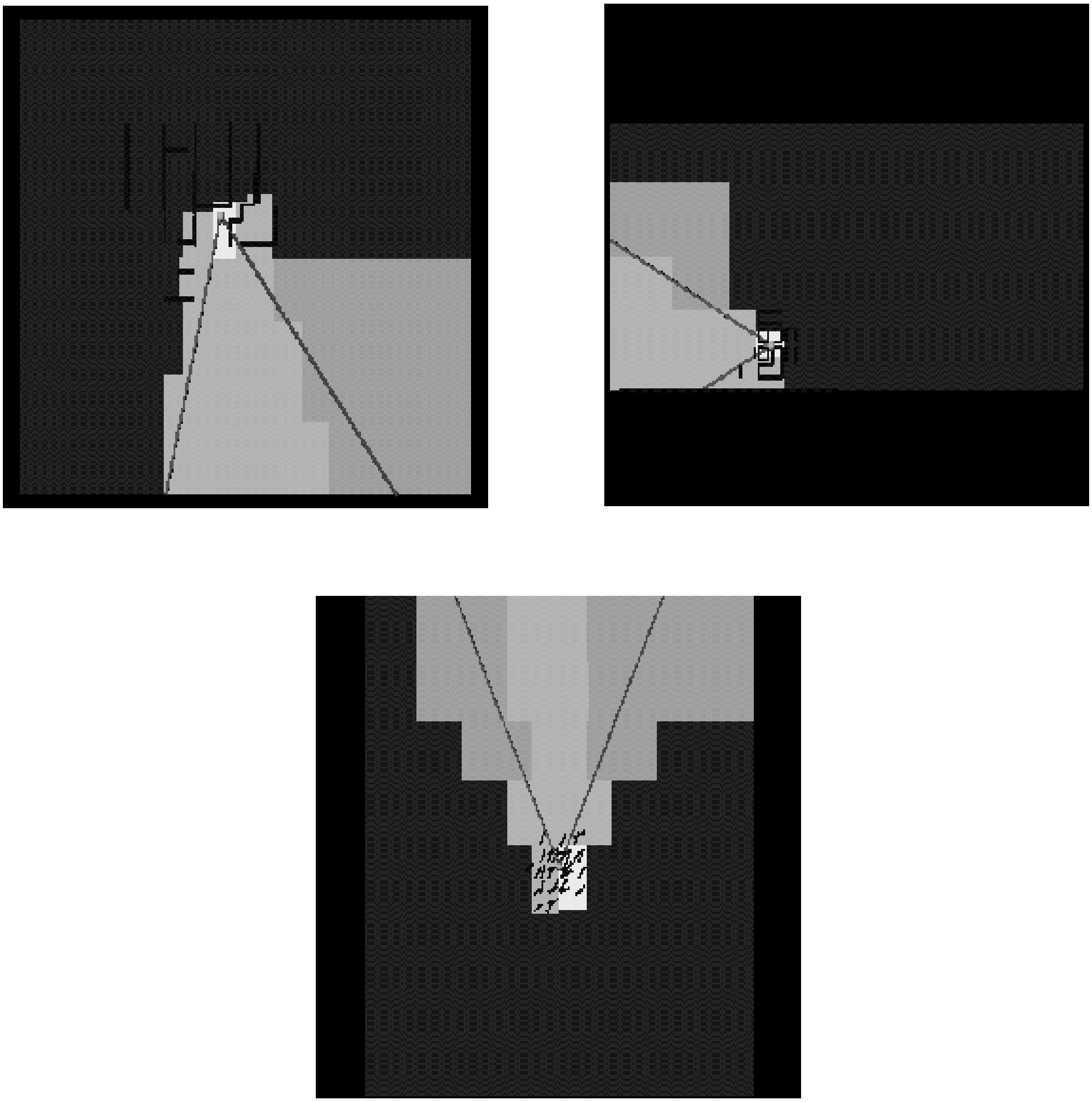

Figure 11: Frustum-based culling performed by the system. Blue shows portions of the model which are found to be completely outside the viewing frustum, and whose occlusion status therefore need not be computed.

to octree nodes being classified as invisible at higher levels than before, avoiding the processing of descendant nodes.

Incorporating frustum based culling to the algorithm reduced execution time by almost a factor of four by avoiding the processing of octree nodes outside the viewing frustum.

4.3 Coherence

	Time in *Visible* (msec)				
Scene	1	2	4	8	16
Soda	17	23	30	43	61
City	20	28	41	55	76
Tetra	26	40	59	86	112

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 the "walking" speed in the previous experiment.

Table 3 shows the time spent in the algorithm *Visible* as the speed of the observer is varied. In this experiment, the observer moves along a fixed straight-line path with different speeds. The table shows that the algorithm spends less time for slower speeds of the observer, reflecting the temporal coherence exploited by the algorithm.

4.4 Parallel Implementation

Scene	Time (msec) Speedup		
	1 proc.	2 proc.	4 proc.
Soda	17	13	8
	1	1.3	2.13
City	20	14	9
	1	1.43	2.22
Tetra	26	18	11
	1	1.55	2.36

Table 4: This table shows speedups obtained using the parallel version of algorithm *Visible*. The times and speedups are reported for 1, 2, and 4 processors.

Table 4 shows parallel performance of algorithm *Visible*, for an observer moving as in the previous experiment. The speedups obtained are significant, though not optimal. After extensive instrumenting, we have determined that even though the load is fairly well balanced (within a few percent of perfect load balance), executing fundamental operations takes longer on more than one processor, due to contention for the memory bus. The data cache size of the processor is fairly small (16KB), and is insufficient to cache data across successive traversals of the octree[1]. We expect that the algorithm will be faster on a machine with larger data caches.

[1] This cache flushing is not due to "ping-ponging" of data between processors. An *affinity* parameter in the load balancing algorithm ensures that octree nodes are likely to be assigned to the same processor as for the previous viewpoint.

5 Conclusions and Future Work

This paper described a parallel, conservative visibility algorithm that exploits spatial and temporal coherence during the motion of an observer through a general polyhedral model. Our results show that temporal and spatial coherence can be exploited to design visibility algorithm that is efficient enough to support viewing actual scenes at interactive frame rates. Moreover, our system can handle scenes with far more polygons than could exact object space visibility algorithms. Casting the algorithm as one of conservative visibility maintenance greatly simplified its design and implementation, and reduced its time and storage complexities, while maintaining the crucial correctness property that the polygons output by the algorithm form the correct image after z-buffering.

Our implementation relies on user-supplied parameters to achieve real-time performance. The maximum octree depth and the number of dynamically maintained occluder polygons should be determined automatically. Also, we plan to extend the visibility algorithm with *prediction* of observer motion a few frames in advance. This information can be used to bound the set of visible polygons for future viewpoint regions before the observer arrives, avoiding synchronous visibility determination.

Two types of occlusive scenes that are not handled well by our algorithm are those with many connected small polygons (such as complex CAD meshes), and those with many disconnected polygons (such as jungle scenes). For the first scene type, we propose to identify occlusive polygon sets, and replace them with large, simplified "virtual" occluders, whose occlusion properties conservatively approximate those of the set. For the second scene type, we have designed a method based on linear programming to quickly detect occlusion in some special circumstances, and are studying the effectiveness of this technique for more general models.

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. *Computer Graphics (Proc. Siggraph '93)* (1993), 109–116.

[3] Baumgart, B. G. A Polyhedron Representation for Computer Vision. In *Proc. AFIPS Natl. Comput. Conf.* (1975), vol. 44, pp. 589–596.

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

[5] Blumofe, R. D. Scheduling Multithreaded Computations by Work Stealing. In *35th Annual Symposium on Foundations of Computer Science* (Santa Fe, New Mexico, 20–22 Nov. 1994), IEEE, pp. 356–368.

[6] Chapman, J., Calvert, T. W., and Dill, J. Exploiting Temporal Coherence in Ray Tracing. In *Proceedings of Graphics Interface '90* (May 1990), pp. 196–204.

[7] Coorg, S., and Teller, S. Temporally Coherent Conservative Visibility. To appear in *Twelfth Annual ACM Symposium on Computational Geometry* (1996).

[8] 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.

[9] 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).

[10] Gigus, Z., Canny, J., and Seidel, R. Efficiently Computing and Representing Aspect Graphs of Polyhedral Objects. *IEEE Transactions on Pattern Analysis and Machine Intelligence 13*, 6 (1991), 542–551.

[11] Gigus, Z., and Malik, J. Computing the Aspect Graph for Line Drawings of Polyhedral Objects. *IEEE Transactions on Pattern Analysis and Machine Intelligence 12*, 2 (1990), 113–122.

[12] Glassner, A. S. Space Subdivision for Fast Ray Tracing. *IEEE Computer Graphics and Applications 4*, 10 (1984), 15–22.

[13] Greene, N., Kass, M., and Miller, G. Hierarchical Z-Buffer Visibility. In *Computer Graphics Proceedings, Annual Conference Series, 1993* (1993), pp. 231–240.

[14] Hubschman, H., and Zucker, S. W. Frame to Frame Coherence and the Hidden Surface Computation: Constraints for a Convex World. *ACM Trans. on Graphics (USA) 1* (Apr. 1982), 129–162.

[15] Jevans, D. A. Object space temporal coherence for ray tracing. In *Proceedings of Graphics Interface '92* (May 1992), pp. 176–183.

[16] Maciel, P. W. C., and Shirley, P. Visual Navigation of Large Environments Using Textured Clusters. In *1995 Symposium on Interactive 3D Graphics* (Apr. 1995), P. Hanrahan and J. Winget, Eds., ACM SIGGRAPH, pp. 95–102. ISBN 0-89791-736-7.

[17] Plantinga, W., and Dyer, C. Visibility, Occlusion, and the Aspect Graph. *Int. J. Computer Vision 5*, 2 (1990), 137–160.

[18] Preparata, F. P., and Shamos, M. I. *Computational Geometry: an Introduction.* Springer-Verlag, 1985.

[19] Samet, H. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS.* Addison-Wesley, 1990.

[20] Teller, S., and Séquin, C. H. Visibility Preprocessing for Interactive Walkthroughs. *Computer Graphics (Proc. Siggraph '91) 25*, 4 (1991), 61–69.