Visible Zone Maintenance for Real-Time Occlusion Culling

Olaf Hall-Holt
Szymon Rusinkiewicz

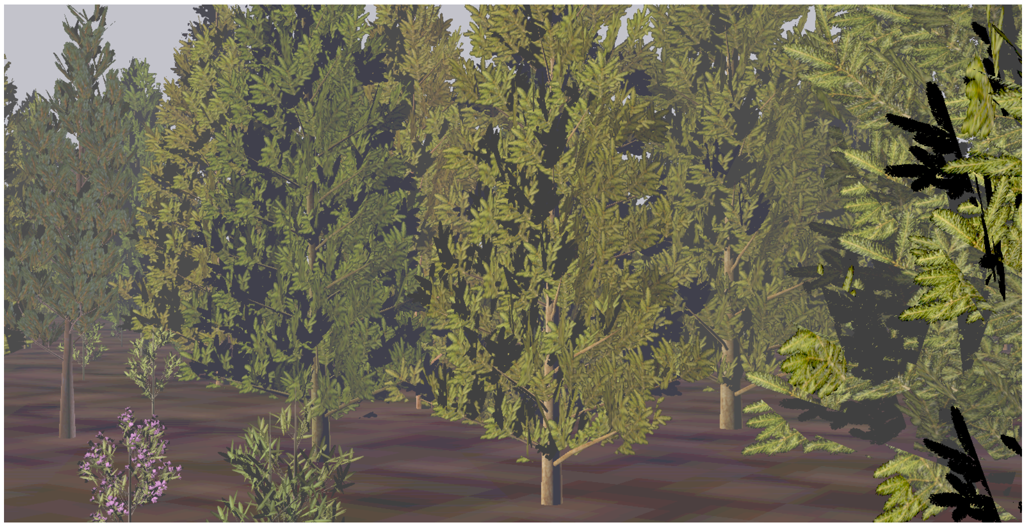


Figure 1: A frame from a flythrough of a forest scene accelerated using visible zone maintenance. The scene consists of 1,000,000 tree objects, each ranging from 2,948 to 38,744 polygons. For this frame, 817,059 of the 7,218,569,204 polygons were rendered.

Abstract

Interactive rendering of a large, dense environment can be accelerated by keeping track of the visible objects. We introduce a framework for maintaining the visible set that provides perfect occluder fusion while taking advantage of temporal coherence in the observer's position. The method is based on maintaining a *visible zone*, a spatial decomposition that supports fast visibility queries and efficient updates. We discuss visible zone maintenance in 2 and 2.5 dimensions, and present extensions to maintain conservative visibility for complex geometry. We present results from an interactive flythrough of a forest environment with one million trees and seven billion polygons.

Categories and Subject Descriptors: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism – Visible line/surface algorithms; I.3.6 [Computer Graphics]: Methodology and Techniques – Graphics data structures and data types; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling – Geometric algorithms, languages, and systems.

Keywords: Occlusion culling, visibility determination, spatial data structures

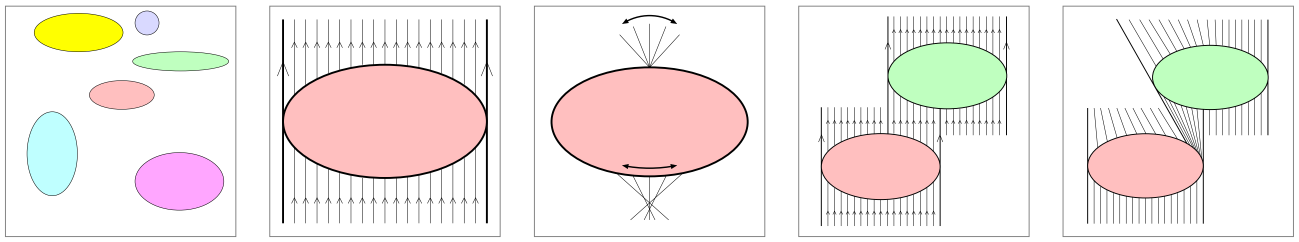
1 Introduction

A major challenge in interactive rendering of large scenes is ensuring that computational effort is concentrated on objects that are visible. For many scenes, *occlusion culling* accelerates rendering considerably by eliminating objects that are completely blocked by other objects. Fast algorithms for occlusion culling often gain their speed from frame-to-frame coherence and occluder fusion. Coherence in observer motion permits the visible set to be maintained incrementally, with computational effort expended only when objects come into or go out of view. Occluder fusion allows the algorithm to consider the aggregate effect of multiple occluders, since in many scenes it is unusual for a single object to block many others.

We present a method for maintaining the visible set of a moving observer that both takes advantage of temporal coherence and performs perfect occluder fusion. Our method is based on maintaining a *visible zone* [Anonymous 01], a quasi-spatial decomposition of the scene analogous to a topological line through the visibility complex [Pocchiola 96]. This data structure has the property that querying the visible set takes time proportional to the number of visible objects, and updating the data structure as the observer moves has cost proportional to the number of changes in the visible set, under certain conditions on object placement the scene. In addition, the visible zone has fast precomputation, can accommodate moving objects, and may be used for arbitrary scenes.

In this paper, we describe the algorithms and data structures used to maintain the visible zone in the plane, and present ways of extending the framework to 2.5-D and 3-D. In addition, we present a flexible approach to applying the visible zone framework to scenes of finely detailed geometry. In this case, it is more efficient to compute a conservative approximation to visibility, so that a superset of the actual visible objects are sent to the graphics pipeline. In many cases the occlusion characteristics of complex geometry may be well approximated by a combination of approximate occluder geometry, and simple functions expressing occlusion relationships between small sets of occluders. We show that the notions of separate geometry for occlusion relationships and approximating the extents of objects may be used to perform conservative occlusion culling in real-time, making the method suitable for flythroughs of large, complex environments.

We begin by surveying a number of approaches to determining visibility. In Section 3, we review the structure of a visible zone, and describe how it is updated as the observer moves. Next, we



(a)
A sample scene in 2-D.

(b)
A ray loop contains two families of rays (called "forward" and "backward" rays), plus two tangent rays.

(c)
In a ray loop, the rays may freely vary in both orientation and position on the object.

(d)
Two consistent ray loops. The forward rays on one object coincide with the backward rays on the other.

(e)
A second set of consistent ray loops, obtained from (d) by turning a tangent ray.

Figure 2: Ray loops.

present extensions of the algorithm to partial occlusion, 2.5-D, and separate geometry for blocking and visibility. We present results from a flythrough of a forest scene with one million trees and seven billion polygons. We conclude by describing other possible applications of the visible zone framework, including one method for generalizing to the full 3-D case.

2 Background: Visibility Algorithms

A variety of algorithms have been proposed for determining the portions of a scene visible from one or more camera viewpoints [Sutherland 74, Dorward 94]. These methods may be classified according to several criteria, including image vs. object space, static vs. moving scene, general vs. restricted scenes, and spatial vs. directional techniques.

Image space algorithms, such as *z*-buffering, determine visibility on a pixel-by-pixel basis, after primitives have been scan-converted. Because of its simplicity and the fact that it always resolves visibility exactly, *z*-buffering is widely implemented in graphics hardware. *Object space* algorithms, in contrast, operate before scan conversion, and thus tend to be more efficient at culling larger sections of the scene. Algorithms such as hierarchical *z*-buffering [Greene 93] are hybrids: they operate in both image and object space. For highest efficiency, however, such systems would require specialized hardware support. For the rest of this paper we will assume that hardware *z*-buffering is used to resolve exact visibility, with *conservative* object-space occlusion culling used to reduce the amount of geometry sent to the graphics pipeline.

For static scenes and a single viewpoint, the problem of accelerating visibility determination has been widely studied in the context of ray tracing [Arvo 89]. For most interactive applications, however, it is necessary to allow the observer position to move, requiring the visibility to be updated at each point in time. One way to do this is to subdivide space into a number of regions, and precompute the visible set from each [Cohen-Or 98]. If the preprocessing step is undesirable, or if objects within the scene are allowed to move, visibility determination must rely on keeping some spatial data structure up to date. In order to make this practical, systems that take advantage of *temporal continuity* in the observer motion have been proposed [Bern 94, Coorg 96].

Specialized algorithms exist that perform fast occlusion culling in the presence of several kinds of scene structure. Many architectural walkthrough systems, for example, take advantage of the structure of indoor scenes by subdividing them into *cells* and *portals* [Teller 91, Funkhouser 92]. Other systems are optimized for scenes in which certain objects frequently block large portions of the scene at once, and determine which objects are blocked by these "large occluders" [Coorg 97]. The efficiency of this strategy may be improved by occluder fusion [Schaufler 00, Durand 00].

Among object space visibility algorithms, a second classification can be applied to distinguish between algorithms that operate primarily by means of spatial subdivisions, and those whose focus is the ray space. Spatial subdivision algorithms typically have compact storage requirements, and are often easy to describe and visualize. Ray space algorithms may have larger storage requirements since they operate in the four dimensional space of lines, but they can also be faster and more directly focused on desired outputs, since a camera takes a sample of rays, rather than points.

The zone-based visibility algorithm used in this paper is unusual in that the data structure is simultaneously a spatial subdivision, and a subset of the ray space. As such, it inherits many of the benefits of both types of approaches. The algorithm has compact storage, fast updates to the data structure when the observer or objects move, and returns the visible set in time proportional to the number of visible objects.

3 Visible Zone Maintenance in 2-D

The visible zone structure arises out of the study of the visibility complex, introduced in [Pocchiola 96]. The visibility complex encodes information about visibility in a scene of non-overlapping objects, and has multiple applications [Durand 97]. Our approach is to identify a particular linear substructure of the visibility complex, and show how it can be maintained as the observer moves. This section is devoted to the two dimensional case, which is the basis for all the visibility algorithms discussed in this paper.

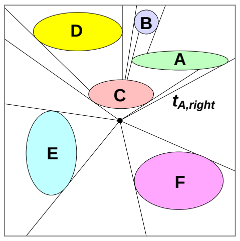
Our maintenance algorithm is based on the Kinetic Data Structure (KDS) framework [Basch 99], in which moving objects are assumed to have known short-term flight plans. The data structure we use has two main characteristics:

1. Deriving the set of objects currently visible from the observer is easy; and
2. These updates are efficient because the structure is only updated when the visibility changes.

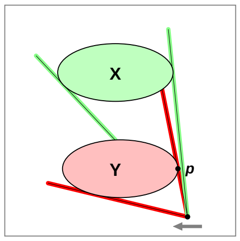
In order for updates to be efficient, the data structure must include more information than just the currently-visible set. This additional information allows rapid identification of newly visible objects.

The visible zone has the following properties in two dimensions:

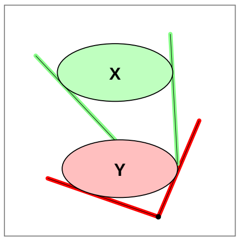
- size linear in the number of objects in the scene.
- directly gives the visible set of the observer.
- local updates are possible by making small changes to the structure when the observer's motion has invalidated a local consistency check.



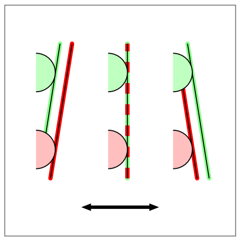
(a)
The initial visible zone of the scene in Figure 2a, formed using a radial decomposition.



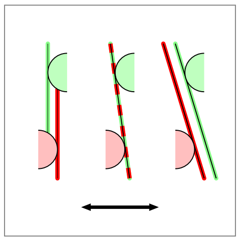
(b)
As the observer moves, a topological change to the visibility zone becomes necessary.



(c)
After the change to the zone, object *X* is no longer visible.



(d)
An elementary step of type "RR." A mirror image of this configuration results in an "LL" elementary step.



(e)
An elementary step of type "LR." A mirror image of this configuration results in an "RL" elementary step.

Figure 3: Visible zones.

We now describe the visible zone, and present a discrete data structure that captures its topological properties. We describe how the data structure is generated initially, and how it is updated as the observer moves.

3.1 Ray Loops and the Visible Zone

Consider a two-dimensional scene consisting of non-overlapping convex objects, as in Figure 2a. Associated with each object, we define a structure called a *ray loop*, as follows:

Let *A* be one of the convex objects of the scene, and let $t_{A,left}$ and $t_{A,right}$ be two rays that are tangent to *A* and that lie locally to the left and right of *A*, respectively. The points of tangency are $pt_{A,left}$ and $pt_{A,right}$. Consider a continuous path along the upper side of the boundary of *A*, from $pt_{A,left}$ to $pt_{A,right}$. The path may double back on itself, but must not go completely around *A*. We associate an outward-pointing ray with each point along this path, with the condition that the orientations of the rays vary continuously. These are called *forward rays* of *A*. The object reached at the other end of a forward ray is called its forward object. Similarly, a continuous family of inward-pointing rays whose endpoints are on the lower side of *A* are *backward rays* of *A*, and the objects from which these rays originate are called the back objects. We define a ray loop to be the union of two tangent rays and the forward and backward rays between them (Figure 2b).

Traversing a ray loop is like a description of a continuously sequence of rays, with one endpoint of the rays always on the edge of the object. The rays are allowed to move around the object freely, and their orientation may vary continuously (Figure 2c). The conditions placed on the ray loop are that it must go around the object exactly once, and become tangent to the object exactly twice.

The ray loops of two objects *A* and *B* can fit together and are called *consistent* if any forward rays of *A* that land on *B* are also backward rays of *B*, and vice versa (see Figure 2d). A consistent union of the ray loops of all objects in the scene is called a *visible zone*, and is the fundamental structure that underlies the visibility algorithms considered in this paper. Actually, as we will see later, in order to determine visibility it is only necessary to consider the topological structure of the visible zone. For example, in Figure 2e we show a second set of consistent ray loops obtained by turning one of the tangent rays. We have also (implicitly) turned the nearby rays in their ray loops, so that all the ray loops remained consistent. Topologically, however, this structure is the same as that in Figure 2d: the same objects are still connected by tangent rays, and the order of tangent rays, forward rays, and backward rays on each loop has not changed. Because of this, we will no longer draw the interior rays in our visibility zone diagrams, and will draw only the tangent rays. For the same reason, the data structure used to actually represent the visible zone (presented in the next section) only needs to include the tangent rays.

3.2 Initializing the Visible Zone

In order to generate an initial consistent visible zone, we first add two extra objects to the scene, representing the observer and the outside bounding box. The initial visible zone consists of all rays pointing directly away from the point observer, the tangent rays of which form a radial spatial subdivision. The result of performing this for the scene of Figure 2a is shown in Figure 3a. As stated earlier, we only show the tangent rays of the zone. Generating this initial zone is takes time $O(n \log n)$ in the number of objects in the scene.

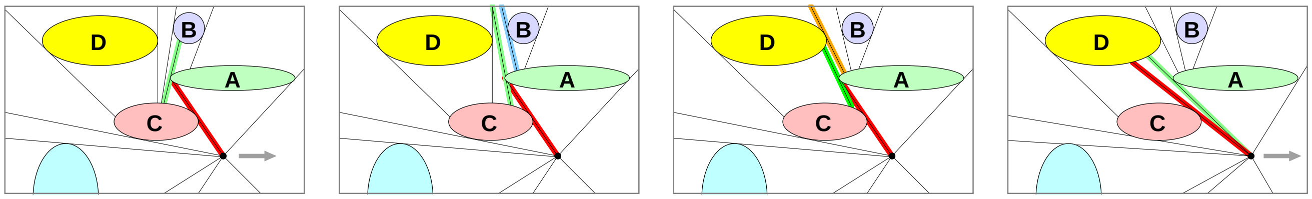
As mentioned above, in order to represent the topological structure of the visible zone, it is sufficient to keep track of the tangent rays and their order along the boundary of an object. The data structure for each tangent ray *t* consists of the following:

1. the object to which *t* is tangent, as well as the forward and back objects at which the ray ends;
2. the type of the tangent, whether locally left or right of their tangent object;
3. the neighboring tangent rays along the ray loops of the forward, tangent, and back objects of *t*.

The data structure for each object describes the geometry of the object, and has pointers to the left and right tangent rays of the object. The total size of the tangent and object data structures is thus proportional to the number of objects in the scene.

We may now use this data structure to obtain the set of objects visible to the observer. This consists of simply sweeping through the observer's ray loop. For example, suppose we begin with the ray $t_{A,right}$ and sweep counterclockwise. The ray $t_{A,right}$ has a pointer to the object *A*, so we may add *A* to the visible set. The data structure for $t_{A,right}$ also contains a pointer to the neighboring tangent along *A*, which is $t_{C,right}$. Thus, we know that *C* is visible, because the back object of $t_{C,right}$ is the observer object. Similarly, we can continue the sweep to obtain all objects visible from the observer. The cost of this sweep is proportional to the number of visible objects. In particular, in this example objects *B* and *D* are not touched during the sweep.

We have defined a visible zone as a set of ray loops in ray space, and represented these rays with a data structure that looks similar to a spatial subdivision. The set of tangent rays, considered as segments in the plane, is a quasi-spatial decomposition, where the segments are allowed to cross to a limited extent (provided that they end on the same object). There exists another data structure that can be used to capture the topology of a set of ray loops, called a pseudo-triangulation. This latter representation is truly a spatial subdivision, but its connection to the space of rays, and the associated update algorithms, is not as intuitive, and so for clarity we focus on tangent rays in this paper. Either representation for a set



(a)
As the observer moves, the tangent ray $t_{C,right}$ becomes tangent to *A*. In order to perform an elementary step, the ray $t_{A,left}$ must be rotated.

(b)
In order to rotate $t_{A,left}$, we perform an "LL" elementary step on $t_{A,left}$ and $t_{B,left}$.

(c)
We continue rotating $t_{A,left}$, and perform an "RL" elementary step on $t_{A,left}$ and $t_{D,right}$.

(d)
We are finally able to complete the "LR" elementary step on $t_{C,right}$ and $t_{A,left}$. The observer's ray loop now includes the object *D*.

Figure 4: A recursive cascade of elementary steps.

of consistent ray loops provides a web of visibility information connecting all the objects of the scene together.

3.3 Maintaining the Visible Set

Since we can always determine the visible set of the observer given a visible zone, the goal of maintaining visibility requires keeping the zone consistent as the observer moves. We could, of course, recompute the zone every time by performing a radial sweep. Alternatively, we could update the zone when the observer moves so that it always has the form of a radial decomposition. These options, however, would require computation time $O(n \log n)$ or $O(n)$, respectively, whenever the observer moved. Instead, we adopt the strategy of changing the visible zone only when necessary, and of making the minimum required changes that still leave the zone consistent. Let us examine what happens as the observer begins to move.

First, we require that the tangent rays connected to the observer move with the observer. Abstractly, each of these tangent rays may be thought of as sliding along the boundary of its tangent object, thus remaining at all times tangent. Computationally, of course, we do not actually keep track of this sliding; we only maintain the identity of the object along which it slides. The topology of the ray loops is not forced to change until one of these tangent rays becomes tangent to a second object. See Figures 3b and 3c, where the right tangent ray $t_{X,right}$ becomes tangent to object Y at a point p along the boundary of Y, at the same time as $t_{Y,right}$ becomes tangent to X. Immediately after coming together, the two tangent rays separate again, as shown in Figure 3c. This operation to change the topology of the ray loops is called an *elementary step*. There are four types of elementary steps: the two shown in Figures 3d and 3e and their mirror images.

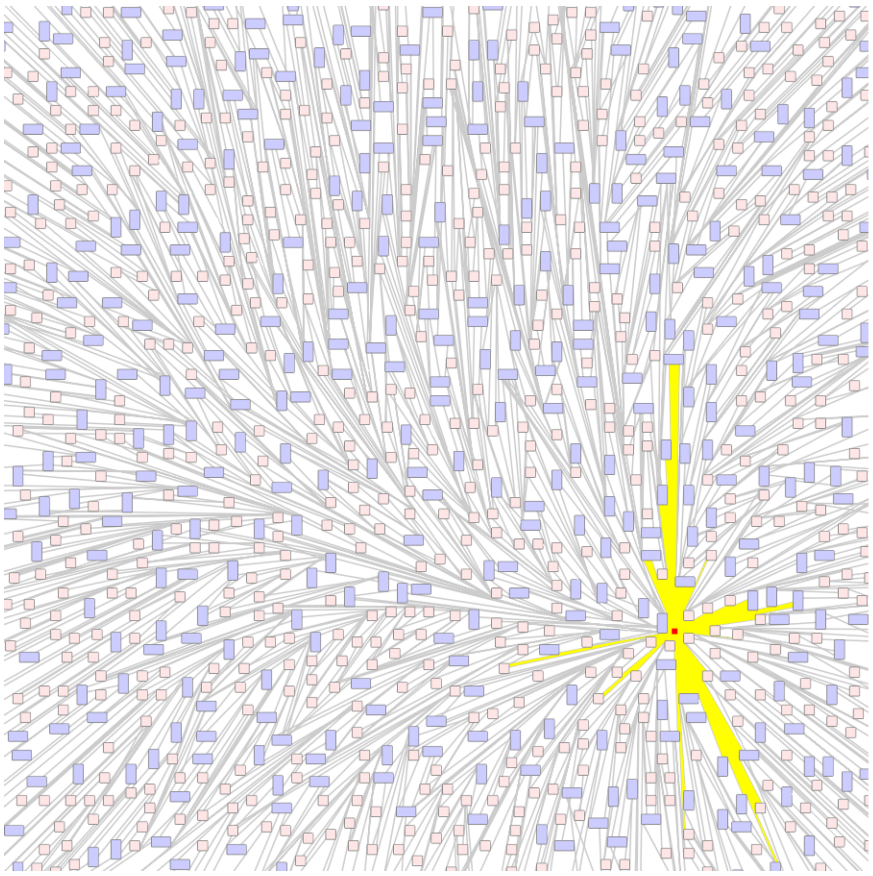
The algorithm to maintain the visible set of a moving observer in the plane proceeds as follows. First, we initialize all the tangent rays in a radial decomposition, as described above. For each tangent ray connected to the moving observer, we compute the time when the tangent ray will first become tangent to another object. These event times are stored in an event queue, so as to process these events in order as they occur.

When an event involving some ray t_1 occurs, we first identify the object A to which it has become tangent. In order for t_1 to continue sliding, it must undergo an elementary step with the appropriate (left or right) tangent ray of A. Let t_2 be this tangent ray of A. If t_2 can slide around A without changing the objects to which it connects, and reach t_1, then an elementary step for t_1 and t_2 is applied directly. However, as illustrated in Figure 4, it may occur that t_2 becomes tangent to a third object B as it slides around A, before it reaches t_1. The same procedure is then recursively invoked, where the appropriate tangent ray of B is turned toward t_2, until t_2 is involved in an elementary step, and continues on its way. Figure 4 illustrates such a cascade of elementary steps, where $t_1 = t_{A,right}$ and $t_2 = t_{A,left}$. In the course of this series of elementary steps, most ray loops outside the immediate neighborhood are not touched, and thus remain consistent with each other. Figure 5a shows an example of a visible zone for a larger scene.

As part of this algorithm, it is often necessary to compute the next object T that a tangent ray t will run into as it moves. A consistent set of ray loops always embeds this information locally, as follows. Let B and C be the forward and back objects of t, respectively. Let S_{fwd} be the set of objects to which the forward rays of C connect, and let S_{back} be the set of objects to which the back rays of B connect. It can be shown that T is always an element of S_{fwd} or S_{back}. Intuitively, as t approaches the boundary of T, some of the rays in the ray loop of T will begin to intersect t. Due to the restriction on intersection of rays in ray loops, these rays must then connect to the same forward or back object as t. Note that it is not necessarily the case that a tangent ray of T reaches B or C; it is only the rays in between two tangent rays that are guaranteed to make the connection. This computation thus involves traversing the ray loops of the forward or back objects, and testing the related objects for possible tangency.

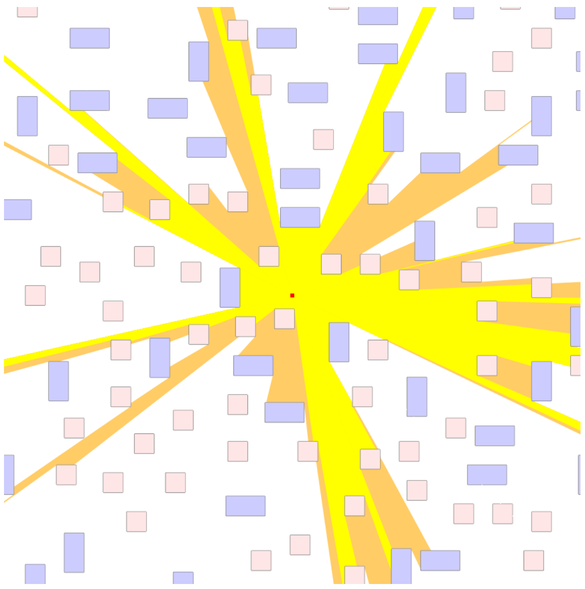
In summary, the algorithm for maintaining consistent ray loops involves recursively turning tangent rays through elementary steps, until the global consistency of the ray loops is restored. Thus the visible zone is similar to a Delaunay triangulation, in that local consistency checks are sufficient to restore a global property. The recursive character of this update algorithm means that some updates may take considerably longer than others, and there is a question as to whether these updates can ever loop around and thus prevent the recursion from terminating. The proof of correctness in the general case is outside the scope of this paper (see [Anonymous 01] for a high level description), but intuitively is based on showing that these updates always move the tangent rays closer to their "rest positions" in a radial decomposition. In particular, it can be shown that the above algorithm for turning a tangent ray around the boundary of an object will always succeed in producing a consistent set of ray loops, provided that the tangent is turned toward the place where it would be in the radial decomposition corresponding to the current location of the observer.

The theoretical complexity of this update algorithm depends on the occlusion characteristics of the scene. If the scene is densely occluded, objects do not line up in long, gently curving rows, and the observer moves along a pseudo-algebraic curve of constant degree, then the amortized complexity of these updates can be shown to be optimal, proportional to the number of changes in the visible set. This algorithm may not be so efficient when the objects are separated by large spaces, or when they are lined up, but the complexity is never worse in the amortized sense than maintaining a radial spatial decomposition. For details, see [Anonymous 01].



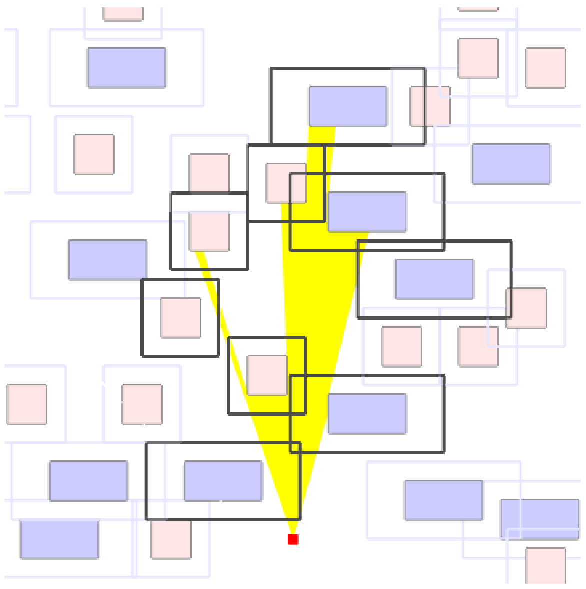
(a)

The visible zone is only updated as needed, in the vicinity of the observer. Note that the rays far away from the observer point towards the center of the scene (i.e., the point around which the initial radial sweep was performed), rather than towards the observer's current position.



(b)

Visibility determination with partial occlusion. The rectangular objects are completely opaque, and the square objects are partially transparent, such that the observer can see through one but not two of them in a row. The yellow polygon represents the directly-visible portion of the scene, and the orange polygon represents the extent of the visibility when partial occlusion is taken into account.



(c)

For conservative visibility determination, each object is given both an (outer) bounding box and an (inner) occluding box. The visibility polygon is computed based on the inner boxes, and all objects whose outer bounding boxes intersect it are assumed to be visible. This figure also shows that we may perform frustum culling by sweeping the observer's ray loop through a limited angle.

Figure 5: Visible zone maintenance 2-D and some extensions.

4 Extensions to the Visible Zone Algorithm

The visibility zone in 2-D is a flexible structure, and may be used for a variety of tasks beyond simply determining visibility in 2-D. In fact, the structure can answer queries about visibility along any ray or set of rays, as long as the ray loops are turned to point along the direction of the query. Here we describe two such extensions, allowing for objects that occlude only partially and 2.5-D scenes. We also describe how to reduce the cost of using the visible zone for objects with high geometric complexity by maintaining conservative, rather than exact, visibility.

4.1 Partial Occlusion

In many cases, objects in the scene must be treated as occluding other objects only partially. One obvious cause for this is objects that are partially transparent (i.e., have non-unit alpha), but in some cases even fully opaque objects might be modeled as not occluding completely. For example, tracking the exact occlusion characteristics of a model of a tree with dense foliage would require the visibility algorithm to keep track of a large amount of geometry. Instead, it would more efficient for the visible zone to use only a rough approximation to the full model, with the understanding that the model does not block the observer's view completely.

In order to handle the open areas in such a model, several objects can be aggregated at a high level. For example, although three trees in a row may individually have many interior open areas, it may be the case that a ray passing though open areas in two of them will not pass through any of the open areas in the third. Such high level occlusion relationships could be precomputed (perhaps using techniques from [Durand 00]) to be used at run time. In general, a function can be defined that expresses whether or where a given sequence of occluders in a row block visibility. We approximate such a function by associating a "partial occlusion" value with each object, depending on the object model and the distance from the viewer. Partial occlusion values are combined along lines of sight to estimate whether a given sequence of occluders is likely to occlude the geometry behind them.

In order to generate an "extended visibility polygon" (including the effects of partial occlusion) for a single viewer position, we start with a single ray from the observer's ray loop. We extend the ray outwards, until the desired stopping condition is met. In order to find the objects the ray hits, we query the visible zone; this may require rotating certain tangent rays until they are aligned with the observer, which in turn may require elementary steps and/or recursive updates to the structure. Once we have found the visibility along one ray, we sweep the ray around the observer, keeping the information about what is visible up to date by querying and updating the zone. Once we have swept completely around the observer, we have visited all the objects that are visible (Figure 5b), and have "straightened out" the zone in a larger region than just the simple visibility polygon.

Once we have generated the extended visibility polygon for a single observer position, we may keep it up to date as the observer moves using the standard KDS approach of computing event times and placing events in the queue. Alternatively, we can maintain only the simple visibility, and recompute the extended visibility polygon once per frame. Depending on the opacities in the scene and the speed of observer motion, either of these approaches may be more efficient; we implement the latter.

4.2 Visibility in 2.5-D

Maintaining visibility in 2.5-D is, in a sense, very similar to the partial occlusion case. We find the closest visible objects using the standard algorithm, then extend rays past the directly-visible objects to see whether anything is visible behind them. In order to make this efficient, we assume that the height of the scene is bounded (i.e., the scene has a "ceiling" and "floor"). We then extend rays from the observer that are tangent to the upper and lower edges of the initial occluder. If these rays hit any other objects, we mark those objects as visible and update the ray along which we search to be tangent to the upper or lower edge of the object we hit. Once the upper ray hits the ceiling and the lower ray hits the floor, we stop the search. The process is illustrated in Figure 6.

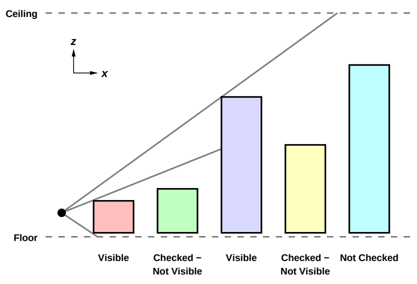


Figure 6: Visibility determination in 2.5-D.

4.3 Conservative Occlusion Culling

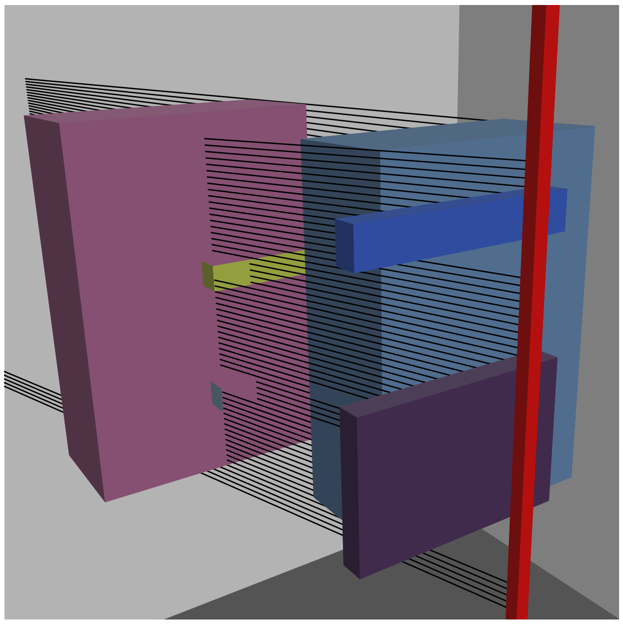
Having the capability to represent non-opaque objects, as described above, will often reduce the complexity of the geometry that must be maintained in the visible zone. In order to further reduce this complexity, we may take advantage of the fact that we will use a hardware *z*-buffer to resolve exact visibility and compute a conservative approximation to the visible set. As long as a superset of the correct visible set is marked "visible," we are sure that any object that should be displayed will, in fact, be displayed.

The algorithm to perform conservative approximation of visibility is based on assigning inner and outer bounds to each object. Any simplification that lies completely inside the object may be used for occlusion: any ray that intersects this box is assumed to be stopped. Thus, this simplified geometry occludes everything behind it. Similarly, any simplification that completely encloses the original object may be used for determining visibility: the object is assumed visible if and only if any part of the outer bounding geometry is unoccluded. This strategy may incorrectly indicate that an object is visible when it is not, but it will never fail to find an object that belongs to the visible set. In our implementation, we use the simplest possible objects for the inner and outer bounds: axis-aligned boxes (Figure 5c).

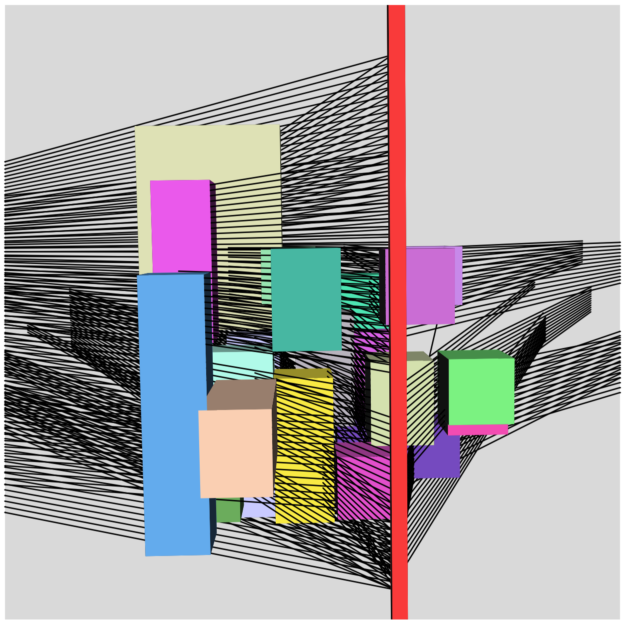
5 Implementation in a Flythrough Application

We have implemented a flythrough system that uses visible zone maintenance to perform occlusion culling. The system incorporates the 2.5-D version of zone maintenance, can use both bounding (outer) and occluder (inner) boxes, and allows for partial occlusion with user-specified per-object opacity.

We present results from an interactive flythrough of a large forest scene (see Figure 1 and the accompanying video). Each tree model contained, on average, approximately 7,000 texture-mapped polygons, but for the purposes of visibility determination was represented as just an outer bounding box and an inner occluder box with some opacity. Instancing was used to control the total storage requirements of the scene – only five distinct tree models were used, and rotated and scaled copies were instantiated to make up the scene. Some statistics from the flythrough are presented in Table 1.



(a)
Grates corresponding to the near left edge of the blue object and the near right edge of the purple object.



(b)
Grates for a collection of objects.

Figure 7: The structure for determining visibility in 3-D keeps track of "grates" which are the analogues of 2-D tangent rays. Similarly, a perspective column (shown in red) replaces the perspective point (i.e., the observer). Although in these figures the grates are shown as discrete lines, in reality they are represented continuously.

Table 1: Statistics about the forest scene flythrough.

Number of objects in scene	1,000,000
Total polygons in scene	7,218,569,204
Avg objects visible per frame	30
Avg visibility computation time	2.5 ms. per frame
Time to generate initial zone	44 sec.

All measurements were made on a PC with 1 GHz Intel Pentium III Xeon processor and NVidia GeForce 2 graphics.

6 A Visible Zone in 3-D

In order to apply the visible zone to fully three-dimensional scenes, we have implemented an approach based on maintaining the visible zone in all horizontal planes parallel to the *xy*-plane simultaneously. As an initial object representation, we use axis-aligned bounding boxes, which have the advantage that their cross section in any horizontal plane is always the same. The representation of the observer is also changed from a point to a vertical line, in order to intersect all the horizontal planes. We generalize the notion of a tangent ray to a set of vertically aligned tangent rays, which are constrained to lie in a plane orthogonal to the *xy*-plane. Thus instead of maintaining a single left and right tangent ray for each object, we instead maintain a continuous set of left tangent rays, and a continous set of right tangent rays, each constrained to lie in a vertical plane. An example of such a set of tangent rays is shown in Figure 7a. The updates to this structure are directly analogous to the two-dimensional case, where the sets of tangent rays slide around an object until they are involved in elementary steps. A snapshot of a test scene for this structure is shown in Figure 7b.

In order to recover visibility in directions not parallel to the *xy*-axis, the same approach to extending the two dimensional visible zone can be applied in this parallel structure. For example, given a non-horizontal ray query from a point along the observer line, imagine moving a bead along the ray, always keeping track of the object between the bead and the observer line. The sets of ray tangents can be brought into alignment with the observer line in the neighborhood of the bead, so the effect is exactly like traversing a radial decomposition of the scene centered at the observer line. To compute the visible set for a given view frustum, a plane sweep can be applied in the same manner, straightening sets of tangent rays as necessary to provide a radial decomposition the neighborhood of the visible region.

7 Conclusions and Future Work

We have presented a method for efficiently maintaining the visible set of a 2-D scene, with extensions to 2.5-D. The visibility zone data structure requires little precomputation, takes space proportional to the number of objects in the scene, and requires updates only when visibility changes. We have shown how to extend the structure to accommodate partially-occluding objects and to maintain (conservative) approximate visibility for geometrically-complex objects.

One feature that could be added to our application is the ability to handle the scene hierarchically, permitting multiple levels of detail for each model. This would improve the effectiveness of the conservative visibility determination, since the simple inner and outer boxes could be used for distant objects, and more accurate inner and outer simplifications for nearby objects.

The visible zone framework could be used for a variety of other applications involving visibility and related queries. For example, the data structure could be used to answer queries about the potentially-visible set in a walkthrough application, allowing for prefetching of data from secondary storage [Funkhouser 92], working set management [Funkhouser 96], and scheduling of computationally-expensive precomputation (e.g., precomputation of soft shadows). Another potential application of the visible zone would be to accelerate ray tracing. Although the structure would be of greatest benefit for accelerating primary rays, it might also be possible to accelerate the tracing of secondary rays if they were traced in coherent bundles [Pharr 97].

References

[**Anonymous 01**] Anonymous. “Kinetic Visible Set Maintenance in the Plane,” Submitted for publication.

[**Arvo 89**] Arvo, J. and Kirk, D. “A Survey of Ray Tracing Acceleration Techniques,” *An Introduction to Ray Tracing*, Glassner, A. S. ed., Academic Press, 1989.

[**Basch 99**] Basch, J., Guibas, L., and Herschberger, J. “Data Structures for Mobile Data,” *Journal of Algorithms*, Vol. 31, No. 1, 1999.

[**Bern 94**] Bern, M., Dobkin, D., Eppstein, D., and Grossman, R. “Visibility with a Moving Point of View,” *Algorithmica*, Vol. 11, No. 4, 1994.

[**Cohen-Or 98**] Cohen-Or, D., Fibich, D., Halperin, D., and Zadicario, E. “Conservative Visibility and Strong Occlusion for Visibility Partitioning of Densely Occluded Scenes,” *Proc. Eurographics*, 1998.

[**Coorg 96**] Coorg, S. and Teller, S. “Temporally Coherent Conservative Visibility,” *Proc. Symposium on Computational Geometry*, 1996.

[**Coorg 97**] Coorg, S. and Teller, S. “Real-Time Occlusion Culling for Models with Large Occluders,” *Proc. Symposium on Interactive 3D Graphics*, 1997.

[**Dorward 94**] Dorward, S. “A Survey of Object-Space Hidden Surface Removal,” *International Journal of Computational Geometry and Applications*, Vol. 4, No. 3, 1994.

[**Durand 97**] Durand, F., Drettakis, G., and Puech, C. “The Visibility Skeleton: A Powerful and Efficient Multi-Purpose Global Visibility Tool,” *Proc. SIGGRAPH*, 1997.

[**Durand 00**] Durand, F., Drettakis, G., Thollot, J., and Puech, C. “Conservative Visibility Preprocessing Using Extended Projections,” *Proc. SIGGRAPH*, 2000.

[**Funkhouser 92**] Funkhouser, T., Séquin, C., and Teller, S. “Management of Large Amounts of Data in Interactive Building Walkthroughs,” *Proc. Symposium on Interactive 3D Graphics*, 1992.

[**Funkhouser 96**] Funkhouser, T. “Database Management for Interactive Display of Large Architectural Models,” *Graphics Interface*, 1996.

[**Greene 93**] Greene, N., Kass, M., and Miller, G. “Hierarchical Z-buffer Visibility,” *Proc. SIGGRAPH*, 1993.

[**Pharr 97**] Pharr, M., Kolb, C., Gershbein, R., and Hanrahan, P. “Rendering Complex Scenes with Memory-Coherent Ray Tracing,” *Proc. SIGGRAPH*, 1997.

[**Pocchiola 96**] Pocchiola, M. and Vegter, G. “Topologically Sweeping Visibility Complexes via Pseudotriangulations,” *Discrete & Computational Geometry*, Vol. 16, No. 4, 1996.

[**Schaufler 00**] Schaufler, G., Dorsey, J., Decoret, X., and Sillion, F. “Conservative Volumetric Visibility with Occluder Fusion,” *Proc. SIGGRAPH*, 2000.

[**Sutherland 74**] Sutherland, I., Sproull, R., and Shumacker, R. “A Characterization of Ten Hidden-Surface Algorithms,” *ACM Computing Surveys*, Vol. 6, No. 1, 1974.

[**Teller 91**] Teller, S. and Séquin, C. “Visibility Preprocessing for Interactive Walkthroughs,” *Proc. SIGGRAPH*, 1991.