Obscuration Culling on Parallel Graphics Architectures

TR95-017
May 1995

Chris Georges

Department of Computer Science
CB #3175, Sitterson Hall
UNC-Chapel Hill
Chapel Hill, NC 27599-3175

Abstract:

Obscuration culling makes use of hierarchy information to accelerate rendering of geometric models. Given some sort of object hierarchy composed of bounding volumes it is possible to avoid transforming and rendering the contents of a volume if the volume itself lies outside the view frustum or is completely invisible when rendered against the current z-buffer. Using this idea in conjunction with the exploitation of temporal coherence to assure that the z-buffer is "almost complete" can yield large speedups for scenes with high depth complexity.

This paper is intended both as an introduction to obscuration culling and as a discussion of issues related to its implementation on parallel machines in general and Pixel-Planes 5 in particular. No familiarity with Pixel-Planes is assumed, but the reader should have some understanding of z-buffering and the standard graphics pipeline. A review of this paper should enable the reader to identify situations in which obscuration culling will be effective, understand the basic sequential algorithm, and understand some of the problems and modifications required in transferring it to a parallel environment.

1. Introduction

Visibility determination is the central bottleneck of most graphics systems. Complex geometric databases challenge standard z-buffer visibility techniques [1], which require the rasterization of all elements. There will always be models too large for these brute force methods to display at interactive rates, even when implemented on the fastest graphics architectures. However, in many situations it is possible to greatly accelerate rendering using coherent aspects of the visibility computation itself. This is the goal of advanced visibility algorithms.

There are at least three types of coherence that can be exploited: object-space, image-space, and temporal. Object-space coherence refers to the tendency of objects near visible objects to be visible as well. If the database is organized such that nearby elements are easily identifiable, often one visibility calculation can suffice for many different elements. Image-space coherence is coherence in the pixel domain; often it is possible to determine visibility at many nearby pixels with a single computation. Temporal coherence usually appears in interactive viewing situations. The view position typically does not change much from frame to frame, so visibility information from the previous frame will often be useful in the next.

Other types of information can prove valuable to visibility algorithms. If part of the database is known to be static (not moving), then the spatial relationships among these elements can be analyzed and mutual visibility information stored in a preprocessing step. For example, often it is known a-priori that certain objects are invisible from certain locations (e.g. because they are in another room of a house). Models such as buildings can be naturally partitioned into disjoint regions (the "rooms," in this case) from which only limited subsets of the entire model are visible. For each region, a potentially visible set (PVS) of other regions can be precomputed. At run time, only elements of regions in the PVS of the current location need to be considered for display.

There are some situations where current visibility algorithms are ineffective [12]. Intuitively, optimizations based on visibility information are useful only in models having a high *average depth complexity*, which means from common viewpoints most lines of sight will intersect many elements of the database. For example, in a terrain database used by a flight simulator, the depth complexity along any line of sight from a typical viewpoint far above the surface will not be very large. Most of the database is visible, so no visibility algorithm will be able to reject a significant fraction of the geometry without evaluating each primitive like the standard z-buffer does. For this case, interactive display requires alternative techniques such as rendering lower-resolution versions of objects that are far away (a method known as “multi-resolution modeling” or "level-of-detail" [12], [13]).

Even if a model does have a high degree of depth complexity, algorithms based on PVS methods may not be able to exploit it. For example, an outdoor environment like a forest will have a great deal of occlusion in any given direction (due to the trunks and leaves of the trees), but partitioning it effectively is difficult because most cells will be able to see most other cells, so the resulting potentially visible sets will be too large to offer any advantage. Even indoor models like office floors and warehouses will have large open spaces though which most regions that might be considered for cell status can be seen.

Visibility algorithms based on obscuration culling offer a viable alternative to PVS techniques for these situations. The basic technique uses a spatial hierarchy in conjunction with an augmented z-buffer to avoid passing invisible geometry to the rasterization engine. This paper will first cover some relevant prior work on visibility, and then move on to describe the original obscuration culling algorithm, *hierarchical visibility* [6], and some issues that arise when parallelizing it. Next I describe a parallel implementation on UNC's Pixel-Planes 5 and report on some preliminary results. The last section will focus on possible future applications of obscuration culling.

2. Prior Work on Visibility

There have been many previous attempts to accelerate visibility determination. Most of them involve building a spatial hierarchy (a tree of objects with bounding volumes, organized such that every child object is contained within its parent's bounding volume), and then traversing it in some fashion to process visibility queries. None of them exploit all the forms of coherence in the visibility computation.

Much research has been devoted to the topic of accelerating ray-tracing renderers [14]. These algorithms typically use spatial hierarchies to speed ray intersection queries. Some of them even make use of temporal coherence by using precomputed object paths to construct space-time bounding volumes. Nevertheless ray-tracing by nature operates on a per pixel basis and thus cannot take advantage of image-space coherence.

Standard z-buffer techniques already use image-space coherence in the rasterization process. A primitive's depth values are interpolated from pixel to pixel across the screen as it is drawn. Most of the PVS methods were designed to be used in conjunction with a z-buffer rendering engine, so they have some claim to utilizing image-space coherence.

The first of the PVS algorithms was developed by Jones [2], who subdivides a model by hand and stores the openings that connect adjacent cells as *portals*. The database is rendered by starting at the cell containing the viewer and passing it and associated primitives to the graphics pipeline. Then adjacent cells are traversed in depth-first fashion. To decide whether to traverse a portal, it must first be clipped to the intersection of all portals along the path from the current cell. If the result is empty, the portal is not followed. This can be thought of as "lazy" evaluation of the potentially visible set, because it is not computed until needed at render time.

By contrast Airey [3] precomputes the PVS's, estimating them using random sampling methods. Teller [4] achieves more efficient and exact visibility determination using computational geometry techniques. He solves the portal-to-portal visibility problem exactly by determining the existence of sightlines through multiple portals from anywhere in a region. He dynamically culls these statically-determined PVS's to a portal view-frustum, as Jones did.

All of these methods run into trouble when the environment does not partition naturally (like an architectural model). Furthermore, Airey's and Teller's systems are optimized to handle primitives that are axis-aligned (i.e. parallel to a coordinate plane), and datasets with a lot of non-axial primitives make both algorithms much less efficient (due to more complex subdivisions and thus more complicated cell-to-cell visibility calculations). There are models that fall into both of these categories but are still densely occluded (like the aforementioned forest), and thus appropriate for visibility techniques. This is the niche that is filled by obscuration culling.

3. Description of Algorithm

The fundamental idea behind obscuration culling is simple: if the bounding volume of an object is completely invisible, then the object itself must be invisible and need not be rendered. Since the bounding volume is comprised of a small constant number of primitives, rendering it is inexpensive compared to the cost of rendering the entire object.

3.1 Choosing and Constructing a Spatial Hierarchy

Obscuration culling requires an object-space hierarchy, so the first step is to construct one. The basic process is recursive: take the current cell, 1) decide if it needs to be subdivided, and if so 2) create two or more new cells within the current one, 3) redistribute the current cell's primitives among the new cells and 4) repeat the process for each new child cell. There are many options at each step, each resulting in a different type of hierarchy. The subdivision criteria is most commonly based on number of primitives in the cell since the goal is usually a hierarchy with a fairly uniform number of primitives in the leaf cells. Subdivision itself can be done any number of ways. If the hierarchy is being constructed by hand usually the child cells are logical subdivisions of the current one (rooms in a house, furniture in a room). A more automatic method is to split the cell along an arbitrary plane that divides the primitives in the cell roughly in half--the resulting structure is known as a BSP tree [5]. The arbitrary splitting plane allows BSP trees to be inherently balanced, but for purposes of determining and testing against boundaries it is often more efficient to constrain cells to be axially aligned. In an "octree," every cell is uniformly subdivided into eight child "cubes" defined by the three axis-aligned planes that bisect the parent.

Primitives are redistributed by giving all those that fall entirely within one child to that child. Primitives that straddle child boundaries can be handled several ways: they can be associated with the parent cell, they can be split and the parts passed to the children, or they can be passed to one child and the child's boundaries expanded to contain it. Every method has some disadvantages. Associating them with a parent cell associates them with the parent cell's visibility which is inefficient for tiny primitives that happen to straddle boundaries. Splitting can greatly increase the total primitive count, and passing to one child may expand the child's bounding volume so much as to be no longer tight enough to be effective.

The original obscuration culling algorithm, "Hierarchical Z-Buffer Visibility," [6] uses an octree subdivision because of its simplicity and efficiency, along with a hybrid redistribution strategy. Primitives that intersect a cube's dividing planes are associated with the cube itself unless the primitive is deemed "small" compared to the cube, in which case it is passed to all children it overlaps and marked after rendering so it is only drawn once.

3.2 Image-Space Hierarchy

Hierarchical visibility uses the hierarchical concept in the image domain as well the spatial one. In standard z-buffering, a depth, or "z-value," is kept for each pixel. This algorithm keeps additional z-buffers at increasingly lower resolutions (all the way down to a single pixel) in a structure called a "z-pyramid." Each level is 4 times smaller than the lower one, and each z-value represents the furthest (maximum) of the 4 pixel z's it covers in the level below it. Figure 1 illustrates this with a simple 3-level example. Each quadrant of level 0 has its maximum value represented in the corresponding level 1 entry, and the uppermost level is the maximum z for the entire z-buffer.

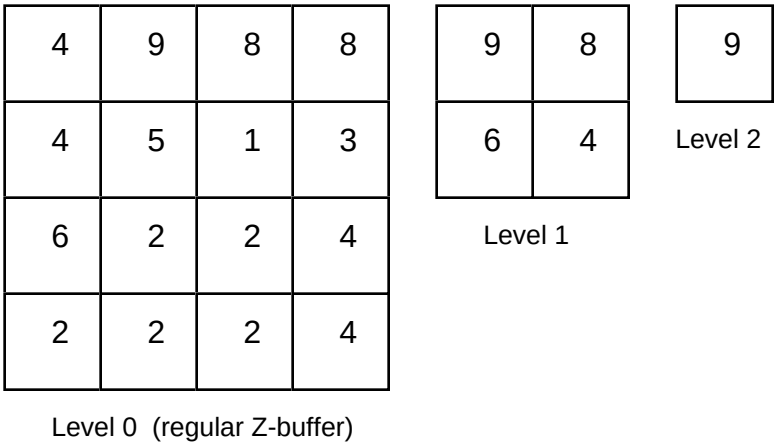

Fig. 1: 3 levels of a simple Z Pyramid

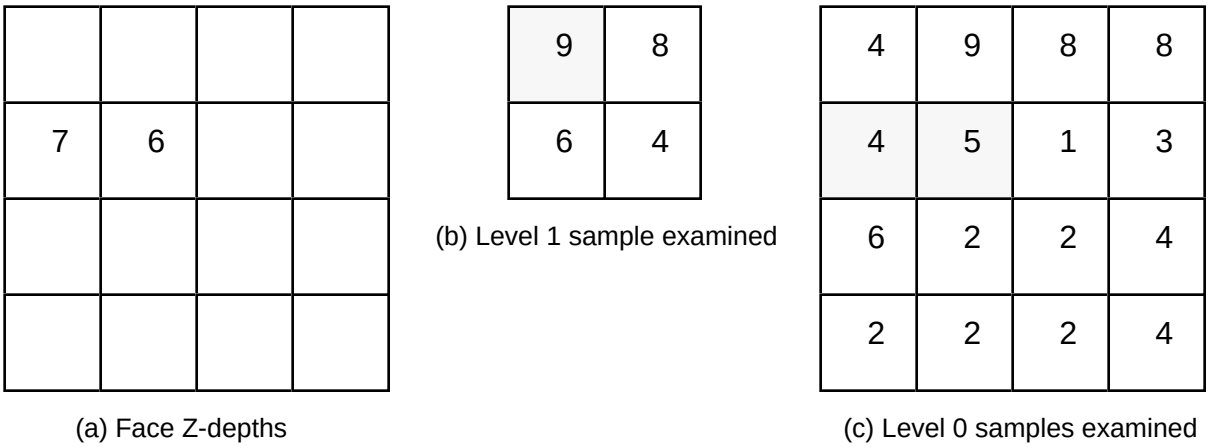

Fig 2: Testing a bounding box face against the Z Pyramid.
(a) represents the z-depths of the face
(b) and (c) show the corresponding samples
examined in the Z Pyramid of Fig. 1

When an object is being tested for visibility, the faces of its associated octree cube are rasterized using the z-pyramid by first determining the screen-space bounding box of each face, and then finding the single finest resolution pixel in the pyramid that still completely covers the bounding box area. The z-value of this sample is compared to the closest z-value of the face (which will be one of the vertices). If the closest face z is further than the sample z (the furthest z of the all visible pixels in the region), then the face must be obscured. The face may still be obscured even if this test fails to reject because the face may not necessarily overlap the particular maximum-depth pixel at full resolution. This can be determined by descending to higher resolutions in the z-pyramid and seeing if all samples the face intersects are closer; eventually either the face will be rejected or it will descend to the highest resolution and determine once and for all that the face is visible.

Figure 2 illustrates this process. An octree cube face covers the 2 pixels marked in (a) with the depths indicated. The closest vertex of the face is at depth 6. The search starts at the single highest resolution pixel that covers the entire bounding box of the face. This turns out to be the upper right pixel of level 1. The value there (9) is farther than the closest vertex (6), so the search continues one level deeper in the pyramid. At level 0, the 2 pixels covered by the face are both closer than the closest vertex, so this face is known to be invisible and the test can proceed to the next face of the current cube. If all faces are invisible, the cube's contents can be disregarded.

If the rejection test fails, the octree cube is assumed to be visible and all associated geometry must be rendered. In the course of doing this the z-pyramid must be updated at every resolution; the primitive is rasterized normally at full resolution and then the affected parts of lower resolution levels are updated as necessary.

3.3 Exploiting Temporal Coherence

The goal of obscuration culling is to maximize the chance of a successful rejection. If the object is invisible in the final frame, then the algorithm

should reject it if at all possible. Rendering order affects this; the first object drawn will not be obscured regardless of its final visibility because the z-buffer is empty, and if you happen to render objects in back to front order nothing will be obscured at the time it is drawn. Ideally you would like to do the opposite: draw objects roughly front to back so invisible objects will have the greatest possible chance of having their bounding boxes obscured.

Temporal coherence in the visibility relation offers a solution. When generating a sequence of frames in an interactive application, the view parameters do not change much from frame to frame and so most objects that were visible last frame will be visible this frame. If you keep a list of all objects whose bounding boxes were visible last frame and draw them first, then the z-buffer will be "almost complete" in that the only remaining obscuration tests that will fail are previously-invisible objects that have just come into view. This is what "hierarchical visibility" does; the objects on the "visible" list are rendered normally, then the rest are rendered if they pass the obscuration test. Every time the test fails to reject, the object is added to the visible list. At the end of each frame all objects on the visible list are checked against the final z-buffer and deleted if found invisible.

4. Parallelization Issues

The original hierarchical visibility algorithm is quite effective for extremely large models, but there are difficulties in parallelizing it for an architecture with multiple distributed (i.e. no shared memory) graphics pipelines.

4.1 Hierarchy Distribution

The first question to be resolved is how to distribute the spatial hierarchy over the various processors. If the hierarchy is flat (just a group of disjoint bounding volumes with no children) then there are two basic approaches (described in [7]). If a *structure* is defined as "a bounding volume and its associated geometry," then *distribute by structure* allocates entire structures to single processors, while *distribute by primitive* puts part of every structure on all processors by passing out the primitives of each in round-robin fashion. The first strategy has poor load-balancing characteristics; if one processor contains only structures that are invisible in that frame then it is essentially sitting idle for much of the frame-generation time. The second method solves this problem since every processor works on part of every object, but every processor also incurs the overhead associated with rendering each structure, which in this case includes the cost of the bounding volume obscuration test.

If this cost is significant then it will be wasteful and expensive to have all processors perform the test for all structures. Furthermore since the test results are independent of one another, some may be done too early and incorrectly fail to reject, resulting in more wasted rendering effort. So the best approach is probably a hybrid of both strategies: individual structures should be kept together on a single processor, but the environment should be subdivided finely enough so that nearby areas can be distributed equally among the processors to balance the load for arbitrary views, but not so finely that the cumulative per structure overhead starts to dominate any possible benefit. Once the structures have been distributed, each processor can recombine its own allotment into a tree hierarchy for greater culling efficiency.

4.2 Temporal Coherence Lists

Another type of load imbalance occurs because the structures are no longer being processed in order. In the sequential algorithm, all structures whose bounding volumes were visible last frame are rendered first to build up the "almost-complete" z-buffer. On a parallel machine, each processor has its own separate list of structures visible (and invisible) last frame. Assuming a MIMD architecture where each processor operates independently, the "visible" and "invisible" stages of rendering will not be synchronized among the processors because the length of the visible lists and the time needed to render them will vary. Consider the case where one processor holds only structures that were visible last frame, while another has only structures that were invisible. Ideally the visible list should be drawn first to increase the chance of an obscuration culling reject. Instead the two types of structures are rendered simultaneously. Redistributing the structures to rebalance the lists every frame is too expensive (unless the processors share memory), and forcing processors to wait until all visible lists are drawn wastes computational power, so other methods of increasing obscuration culling efficiency are needed.

One useful technique is to render the visible list in order of each object's screen-space area. The goal is to fill the z-buffer as fast as possible, so the objects that cover the largest screen area should be drawn first. This area can be approximated quickly using the projected area of the bounding volume (or even more quickly using the screen-space bounding box of the projected vertices of the bounding volume). This computation is done for all structures every frame and the visible list sorted from largest to smallest area.

It would seem as if the structures that were invisible last frame should also be drawn in the order of bounding volume area, but a better heuristic is available. The efficiency of obscuration culling is the sum over all objects of the probability that an object invisible in the final frame is rejected by the cull multiplied by the work saved by culling the object. Under the assumption of temporal coherence, most objects invisible last frame will still be invisible this frame, so rendering them will not contribute to final z-buffer and will not increase the chance of another object being obscured. To increase efficiency we should instead concentrate on increasing the amount of work saved by the rejections that are successful. Structures that cost the most should have their obscuration test postponed as long as possible to allow other processors to fill up the z-buffer, maximizing the chance of a rejection.

If the time to render any primitive is assumed to be roughly constant, (valid on the Pixel-Planes machines but less so on an architecture like the SGI Reality-Engine [8] where rendering time is proportional to screen-size of the primitives) then the work required to render a structure can be measured strictly in terms of number of primitives. The structures on the invisible list can then be sorted from smallest to largest number of primitives, and rendered in that order.

Deletion from the visible list presents a final problem with parallelizing the original sequential algorithm. At the end of each frame, “hierarchical visibility” assumes you can test every structure on the visible list against the final z-buffer, and delete those whose bounding volume does not appear. If the obscuration test is expensive (if it requires some network communication), you would rather not do these extra tests every frame. Instead, the full recheck can be done after a fixed number of frames, which should be fairly small to ensure the visible list does not grow too large. The recheck could be done

earlier if a large number of structures were suddenly added, indicating a possibly drastic change in overall visibility. If the architecture allows you to quickly determine the maximum depth in the z-buffer, a rapid shift in this value can be another early indicator of significant visibility change (the viewer may have moved in front of a wall).

5. Obscuration Culling on Pixel Planes 5

I have implemented obscuration culling on Pixel-Planes 5, a high-performance graphics multicomputer developed at UNC [9]. The Pixel-Planes 5 system consists of a host workstation, several dozen Graphics Processors (GPs), 10-20 SIMD processor arrays called Renderers, and a frame buffer, all of which are mounted on a high bandwidth ring network. Each GP is an Intel i860, a general purpose floating point processor with its own memory. In traditional rendering mode, each GP transforms its segment of the display list and hands the result off to the Renderers. The Renderers rasterize primitives using an 128x128 array of 1-bit processors, each assigned to a different screen pixel. To rasterize, all processors outside the primitive boundary turn themselves off, then all processors at which the primitive is farther than the last stored pixel turn themselves off (z-buffering), and finally the remaining processors write the appropriate information for the primitive into their pixel memory. Each Renderer handles a different 128x128 region of the screen. When rasterization is complete, the screen-subimages are collected in the frame buffer and displayed.

Pixel-Planes 5 is an attractive machine to perform obscuration culling on because its hardware allows a z-query test to be implemented efficiently. To know if the primitives composing the bounding volume are visible, you need a way to tell if any of the pixels of a rasterized primitive appeared in the z-buffer. The heavily pipelined architecture of some popular machines like the SGI Reality Engine prevent this from being done quickly, but the Pixel-Planes Renderers, like most parallel SIMD machines, have a global OR of all processor enables that allows you to immediately tell if any of the processors are turned on by polling a status flag (called the "All Enables Off", or AEO bit) (see [10]). The z-query is done by simply checking the value of this bit after rasterizing a primitive.

The z-pyramid concept of hierarchical visibility is not useful on Pixel-Planes. The z-pyramid saves time if you are doing the comparisons sequentially, checking the nearest z of a primitive against progressively increasing z-buffer resolutions. On Pixel-Planes, you can do this comparison at every pixel simultaneously, effectively going to the highest resolution at no extra cost.

5.1 Overview

My preliminary implementation takes a "distribute by structure" approach and assumes a strictly flat object hierarchy. The structures are passed out round-robin to the GPs and a 3D axial bounding box is computed for each. The structures are all initialized as "invisible last frame," and rendering proceeds.

First, for every structure on its visible list, each GP computes the area of the axial 2D bounding box of the projected 3D bounding box vertices and sorts the list in order of decreasing area. If the projected bounding box intersects the view frustum (i.e. it's in front of the viewer and would appear somewhere on screen), the structure's primitives are transformed and sent to the Renderers

as normal.

Next, the structures on the invisible list are traversed in order of increasing number of primitives (the sorting is done as a pre-process). Each structure inside the viewing frustum is tested for obscuration. The triangles composing the bounding box are transformed and sent to the Renderers to be rasterized (without overwriting any of the actual pixels), and each Renderer sends back a status word indicating the resulting AEO value. If the bounding box is invisible in all screen regions it covers, then its contents are completely obscured and can be skipped. Otherwise the structure is transformed, rasterized, and moved to the visible list for the next frame.

Every 30 frames, all structures on the visible list are checked against the final frame z-buffer and those that do not appear are moved to the invisible list. Note that view-frustum culled structures do not automatically move to the invisible list. In an interactive application, visible structures usually leave the view frustum due to rotational motion, not positional changes that would affect obscuration. This way if the viewer turns and then looks back at a structure it will be correctly predicted as visible.

One implementation detail that deserves special mention is the renderer allocation method. Usually there are fewer Renderers than screen regions, so there must be a way of deciding which renderers work on which regions. The standard Pixel-Planes 5 rendering control system uses dynamic renderer allocation: every frame, the regions that received the most primitives last frame are assigned renderers first, and when a renderer finishes it is assigned the next available region. This method is dynamic in the sense that the renderer-to-region assignment is not known before the frame begins. This causes problems for obscuration culling because the GPs need to be able to query all the screen regions a bounding box covers to see if its visible, but only the regions currently assigned to renderers would be accessible under the dynamic method. Since all screen regions need to be available simultaneously, the only solution is to have the renderers work on several regions at once. The assignment is done at startup so the GPs know which renderer is handling which region; this is called static renderer allocation. Static allocation has less overhead than dynamic, but drastically reduces the amount of memory available to store pixel shading parameters (since each renderer must now store multiple regions) and may perform poorly if all the primitives are concentrated in the regions of one renderer. It also forces you to abandon systems built on the standard rendering control software base (like the PPHIGS and vlib graphics libraries commonly used by Pixel-Planes applications).

5.2 Preliminary Results

The obscuration culling system was tested first with a simple model that had a plain box and a series of complex objects consisting of several thousand polygons. To achieve the most dramatic effect it was run on a 1-GP system. With all objects in sight the scene was rendered at about 2-3 frames/second, but when the viewer moved so that the box obscured his view of the other objects, most of the geometry was culled and the frame rate immediately jumped to near the monitor refresh rate (the maximum useful rate). A more realistic test involved a 300,000 triangle model of a submarine control deck that contained many large, occluding polygons and one very dense object: a heavily tesselated torpedo (some 100K triangles) hidden behind some walls. When the torpedo was in the view frustum but invisible, obscuration culling

would generally improve the frame rate by 20-30%. More extensive testing is required to assess the impact of the various optimizations on overall performance.

5.3 Possible Optimizations

There are several hardware quirks specific to Pixel-Planes 5 that could be exploited to improve efficiency. One is the capability of a renderer to quickly determine a global maximum z depth over all pixel-processors. If a GP finds that the closest vertex of the bounding box is farther than the maximum z in all regions it covers, it can reject the structure without sending it to the renderers. This is somewhat akin to a z-pyramid with 2 levels, a single pixel and full resolution. It is not absolutely clear at what point in the frame should the GPs ask the renderers for their maximum z values, but a good time to do this might be after the visible list has been rendered, when the z-buffer should be "almost-complete" anyway.

Another useful optimization would be to overlap obscuration culling tests for different structures. The test currently requires a GP to wait for AEO status replies from all regions the bounding box covers. To avoid the potentially significant latency involved, the GP could move on to test or transform other structures and come back later to process the replies.

One feature that distinguishes Pixel-Planes 5 from other graphics platforms is the renderers' quadratic expression evaluator (see [9]), which allows them to directly render quadratic surfaces like spheres in parallel. In fact spheres are slightly faster than the seemingly simpler polygons because they are described by a single quadratic expression instead of 3 or more linear ones. Spheres could be used as another type of bounding volume that would fit some geometries tighter than the standard axial box.

6. Conclusion and Future Applications

Obscuration culling is an effective method for accelerating visibility calculations in many situations. I have implemented a prototype system on Pixel-Planes 5 and preliminary results are encouraging. In general obscuration culling can be expected to work well for models that are heavily occluded and subdivided in a way that captures the dense concentrations of primitives. The speedups will always be limited by the depth complexity of any particular view--if you can see everything, there is not much any visibility method could do.

A few applications of obscuration culling are worth investigating. One is the use of obscuration tests in a recursive portal culling scheme like that of Jones. Portals are similar to bounding boxes in that they bound the area over which a cell could appear, and if the portal is invisible from the current viewpoint then everything in the cell must also be invisible. Incorporating obscuration testing in a portal scheme would benefit situations where the interior of a cell is somewhat occluded so that portals are often invisible from typical viewpoints.

Another potential application is interactive radiosity. Radiosity is a global illumination algorithm that uses finite element methods to determine a view-independent lighting solution for a polyhedral environment (see [11] for more details). To do this, every polygonal patch is assigned an initial energy representing its inherent luminosity (the ones with non-zero values are

emitters). On each algorithmic iteration, it *shoots* its remaining energy at all the other patches it can see. The amount received by each patch depends on the inter-patch *form factor*, a geometric quantity representing the fraction of the total energy leaving the shooter that arrives at this particular receiver. Occlusion affects this energy transfer, so visibility methods come into play--one common way to compute the form factors involves rasterizing the scene from the point of view of every patch. Whenever the geometry of the model changes the energy transfer relationships change and the form factors must be completely recomputed. If the geometry change is incremental as might be expected in an interactive environment, then the patch visibility interrelationships will exhibit a good deal of temporal coherence from frame to frame, and obscuration culling is a natural choice to exploit it.

References

[1] Foley, et. al. "Computer Graphics: Principles and Practice," Addison-Wesley, 1990.

[2] C.B. Jones. "A New Approach to the 'Hidden Line' Problem," The Computer Journal, vol 14 no 3 (August 1971)

[3] J. Airey. "Increasing Update Rates in the Building Walkthrough System with Automatic Model-Space Subdivision and Potentially Visible Set Calculations," Ph.D. Thesis, CS Dept, UNC-CH, 1990.

[4] S. Teller. "Visibility Computations in Densely Occluded Polyhedral Environments," Ph.D. Thesis, CS Dept, UC Berkeley, 1992.

[5] H. Fuchs, Z. Kedem, B. Naylor. "On Visible Surface Generation by A Priori Tree Structures," Proceedings of ACM Siggraph 1980

[6] M. Kass and N. Greene. "Hierarchical Z-Buffer Visibility," Proceedings of ACM Siggraph 1993

[7] D. Ellsworth, H. Good, B. Tebbs. "Distributing Display Lists on a Multicomputer," Proceedings of ACM Siggraph 1990 Symposium on Interactive Graphics

[8] K. Akeley. "RealityEngine Graphics," Proceedings of ACM Siggraph 1993

[9] H. Fuchs, et. al. "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories," Proceedings of ACM Siggraph 1989

[10] J. Eyles, "Pixel-Planes 5 System Documentation," Ch. III.4 (Renderer Description), UNC CS Dept

[11] M. Cohen and J. Wallace. "Radiosity and Realistic Image Synthesis," Academic Press, 1993.

[12] T. Funkhouser and C. Sequin. "Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Environments," Proceedings of ACM Siggraph 1993

[13] P. Heckbert and M. Garland. "Multiresolution Modeling for Fast Rendering," Proceeding of Graphics Interface '94 (May 1994)

[14] M. Kaplan. The use of spatial coherence in ray tracing. In *Techniques for Computer Graphics, etc.*, D. Rogers and R.A. Earnshaw, Springer-Verlag, New York, 1987