Hierarchical Z-Buffer Visibility

Ned Greene* Michael Kass† Gavin Miller†

Abstract

An ideal visibility algorithm should a) quickly reject most of the hidden geometry in a model and b) exploit the spatial and perhaps temporal coherence of the images being generated. Ray casting with spatial subdivision does well on criterion (a), but poorly on criterion (b). Traditional Z-buffer scan conversion does well on criterion (b), but poorly on criterion (a). Here we present a hierarchical Z-buffer scan-conversion algorithm that does well on both criteria. The method uses two hierarchical data structures, an object-space octree and an image-space Z pyramid, to accelerate scan conversion. The two hierarchical data structures make it possible to reject hidden geometry very rapidly while rendering visible geometry with the speed of scan conversion. For animation, the algorithm is also able to exploit temporal coherence. The method is well suited to models with high depth complexity, achieving orders of magnitude acceleration in some cases compared to ordinary Z-buffer scan conversion.

CR Categories and Subject Descriptors: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism - Hidden line/surface removal; J.6 [Computer-Aided Engineering]: Computer-Aided I.3.1 [Computer Graphics]: Hardware Architecture - Graphics Processors

Additional Key Words and Phrases: Octree, Pyramid, Temporal Coherence, Spatial Coherence, Z Buffer.

1 Introduction

Extremely complex geometric databases offer interesting challenges for visibility algorithms. Consider, for example, an interactive walk-through of a detailed geometric database describing an entire city, complete with vegetation, buildings, furniture inside the buildings and the contents of the furniture. Traditional visibility algorithms running on currently available hardware cannot come close to rendering scenes of this complexity at interactive rates and it will be a long time before faster hardware alone will suffice. In order to get the most out of available hardware, we need faster algorithms that exploit properties of the visibility computation itself.

There are at least three types of coherence inherent in the visibility computation which can be exploited to accelerate a visibility algorithm. The first is object-space coherence: in many cases a single computation can resolve the visibility of a collection of objects which are near each other in space. The second is image-space coherence: in many cases a single computation can resolve the visibility of an object covering a collection of pixels. The third is temporal coherence: visibility information from one frame can often be used to accelerate visibility computation for the next frame. Here we present a visibility algorithm which exploits all three of these types of coherence and sometimes achieves orders of magnitude acceleration compared with traditional techniques.

The dominant algorithms in use today for visibility computations are Z-buffer scan conversion and ray-tracing. Since Z buffers do not handle partially transparent surfaces well, we will restrict the discussion to models consisting entirely of opaque surfaces. For these models, only rays from the eye to the first surface are relevant for visibility, so the choice is between Z buffering and ray-casting (ray-tracing with no secondary rays).

Traditional Z buffering makes reasonably good use of image-space coherence in the course of scan conversion. Implementations usually do a set-up computation for each polygon and then an incremental update for each pixel in the polygon. Since the incremental update is typically much less computation than the set-up, the savings from image-space coherence can be substantial. The problem with the traditional Z-buffer approach is that it makes no use at all of object-space or temporal coherence. Each polygon is rendered independently, and no information is saved from prior frames. For extremely complex environments like a model of a city, this is very inefficient. A traditional Z-buffer algorithm, for example, will have to take the time to render every polygon of every object in every drawer of every desk in a building even if the whole building cannot be seen, because the traditional algorithm can resolve visibility only at the pixel level.

Traditional ray-tracing or ray-casting methods, on the other hand, make use of object-space coherence by organizing the objects in some type of spatial subdivision. Rays from the eye are propagated through the spatial subdivision until they hit the first visible surface. Once a ray hits a visible surface, there is no need to consider any of the surfaces in the spatial subdivisions further down along the ray, so large portions of the geometry may never have to be considered during rendering. This is an important improvement on Z buffering, but it makes no use of temporal or image-space coherence. While ray-casting algorithms that exploit temporal coherence have been explored, it seems extremely difficult to exploit image-space coherence in traditional ray casting algorithms.

Here we present a visibility algorithm which combines the strengths of both ray-casting and Z buffering. To exploit object-

*Apple Computer, U.C. Santa Cruz
†Apple Computer

space coherence, we use an octree spatial subdivision of the type commonly used to accelerate ray tracing. To exploit image-space coherence, we augment traditional Z-buffer scan conversion with an image-space Z pyramid that allows us to reject hidden geometry very quickly. Finally, to exploit temporal coherence, we use the geometry that was visible in the previous frame to construct a starting point for the algorithm. The result is an algorithm which is orders of magnitude faster than traditional ray-casting or Z buffering for some models we have tried. The algorithm is not difficult to implement and works for arbitrary polygonal databases.

In section II, we survey the most relevant prior work on accelerating ray casting and scan conversion. In section III, we develop the data structures used to exploit object-space, image-space and temporal coherence. In section IV, we describe the implementation and show results for some complex models containing hundreds of millions of polygons.

2 Prior Work

There have been many attempts to accelerate traditional ray-tracing and Z buffering techniques. Each of these attempts exploits some aspect of the coherence inherent in the visibility computation itself. None of them, however, simultaneously exploits object-space, image-space and temporal coherence.

The ray-tracing literature abounds with references to object-space coherence. A variety of spatial subdivisions have been used to exploit this coherence and they seem to work quite well (e.g. [1, 2, 3, 4, 5]). Temporal coherence is much less commonly exploited in practice, but various techniques exist for special cases. If all the objects are convex and remain stationary while the camera moves, then there are constraints on the way visibility can change[6] which a ray tracer might exploit. On the other hand, if the camera is stationary, then rays which are unaffected by the motion of objects can be detected and used from the previous frame[7]. When interactivity is not an issue and sufficient memory is available, it can be feasible to render an entire animation sequence at once using spacetime bounding boxes[8, 9]. While these techniques make good use of object-space coherence and sometimes exploit temporal coherence effectively, they unfortunately make little or no use of image-space coherence since each pixel is traced independently from its neighbors. There are heuristic methods which construct estimates of the results of ray-tracing a pixel from the results at nearby pixels (e.g. [10]), but there seems to be no guaranteed algorithm which makes good use of image-space coherence in ray tracing.

With Z-buffer methods (and scan conversion methods in general) the problems are very different. Ordinary Z-buffer rendering is usually implemented with an initial set-up computation for each primitive followed by a scan-conversion phase in which the affected pixels are incrementally updated. This already makes very good use of image-space coherence, so the remaining challenge with Z-buffer methods is to exploit object-space and temporal coherence effectively.

A simple method of using object-space coherence in Z-buffer rendering is to use a spatial subdivision to cull the model to the viewing frustum [11]. While this can provide substantial acceleration, it exploits only a small portion of the object-space coherence in models with high depth complexity. In architectural models, for example, a great deal of geometry hidden behind walls may lie within the viewing frustum.

In order to make use of more of the object-space coherence in architectural models, Airey et. al. [12, 13] and subsequently Teller and Sequin[15] proposed dividing models up into a set of disjoint cells and precomputing the potentially visible set (PVS) of polygons from each cell. In order to render an image from any viewpoint within a cell, only the polygons in the PVS need be considered. These PVS schemes are the closest in spirit to the visibility algorithm presented here since they attempt to make good use of both object-space and image-space coherence. Nonetheless, they suffer from some important limitations. Before they can be used at all, they require an expensive precomputation step to determine the PVS and a great deal of memory to store it. Teller and Sequin, for example, report over 6 hours of precomputation time on a 50 MIP machine to calculate 58Mb of PVS data needed for a model of 250,000 polygons[15]. Perhaps more importantly, the way these methods make use of cells may limit their appropriateness to architectural models. In order to achieve maximum acceleration, the cells must be 3D regions of space which are almost entirely enclosed by occluding surfaces, so that most cells are hidden from most other cells. For architectural models, this often works well since the cells can be rooms, but for outdoor scenes and more general settings, it is unclear whether or not PVS methods are effective. In addition, the currently implemented algorithms make very special use of axially-aligned polygons such as flat walls in rectilinear architectural models. While the methods can in principle be extended to use general 3D polygons for occlusion, the necessary algorithms have much worse computational complexity[15]. Finally, although the implementations prefetch PVS data for nearby cells to avoid long latencies due to paging, they cannot be said to exploit temporal coherence in the visibility computation very effectively.

The algorithm presented here shares a great deal with the work of Meagher[16] who used object-space octrees with image-space quadtrees for rendering purposes. Meagher tried to display the octree itself rather than using it to cull a polygonal database, so his method is directly applicable to volume, rather than surface models. Nonetheless his algorithm is one of the few to make use of both object-space and image-space coherence. The algorithm does not exploit temporal coherence.

3 Hierarchical Visibility

The hierarchical Z-buffer visibility algorithm uses an octree spatial subdivision to exploit object-space coherence, a Z pyramid to exploit image-space coherence, and a list of previously visible octree nodes to exploit temporal coherence. While the full value of the algorithm is achieved by using all three of these together, the object-space octree and the image-space Z pyramid can also be used separately. Whether used separately or together, these data structures make it possible to compute the same result as ordinary Z buffering at less computational expense.

3.1 Object-space octree

Octrees have been used previously to accelerate ray tracing[5] and rendering of volume data sets[16] with great effectiveness. With some important modification, many of the principles of these previous efforts can be applied to Z-buffer scan conversion. The result is an algorithm which can accelerate Z buffering by orders of magnitude for models with sufficient depth complexity.

In order to be precise about the octree algorithm, let us begin with some simple definitions. We will say that a polygon is hidden with respect to a Z buffer if no pixel of the polygon is closer to the observer than the Z value already in the Z buffer. Similarly, we will say that a cube is hidden with respect to a Z buffer if all of its faces are hidden polygons. Finally, we will call a node of the octree hidden if its associated cube is hidden. Note that these definitions depend on the sampling of the Z buffer. A polygon which is hidden at one Z-buffer resolution may not be hidden at another.

With these definitions, we can state the basic observation that makes it possible to combine Z buffering with an octree spatial subdivision: If a cube is hidden with respect to a Z buffer, then all polygons fully contained in the cube are also hidden. What this means is the following: if we scan convert the faces of an octree cube and find that each pixel of the cube is behind the current surface in the Z buffer, we can safely ignore all the geometry contained in that cube.

From this observation, the basic algorithm is easy to construct. We begin by placing the geometry into an octree, associating each primitive with the smallest enclosing octree cube. Then we start at the root node of the octree and render it using the following recursive steps: First, we check to see if the octree cube intersects the viewing frustum. If not, we are done. If the cube does intersect the viewing frustum, we scan convert the faces of the cube to determine whether or not the whole cube is hidden. If the cube is hidden, we are done. Otherwise, we scan convert any geometry associated with the cube and then recursively render its children in front-to-back order.

We can construct the octree with a simple recursive procedure. Beginning with a root cube large enough to enclose the entire model and the complete list of geometric primitives, we recursively perform the following steps: If the number of primitives is sufficiently small, we associate all of the primitives with the cube and exit. Otherwise, we associate with the cube any primitive which intersects at least one of three axis-aligned planes that bisect the cube. We then subdivide the octree cube and call the procedure recursively with each of the eight child cubes and the portion of the geometry that fits entirely in that cube.

The basic rendering algorithm has some very interesting properties. First of all, it only renders geometry contained in octree nodes which are not hidden. Some of the rendered polygons may be hidden, but all of them are “nearly visible” in the following sense: there is some place we could move the polygon where it would be visible which is no further away than the length of the diagonal of its containing octree cube. This is a big improvement over merely culling to the viewing frustum. In addition, the algorithm does not waste time on irrelevant portions of the octree since it only visits octree nodes whose parents are not hidden. Finally, the algorithm never visits an octree node more than once during rendering. This stands in marked contrast to ray-tracing through an octree where the root node is visited by every pixel and other nodes may be visited tens of thousands of times. As a result of these properties, the basic algorithm culls hidden geometry very efficiently.

A weakness of the basic algorithm is that it associates some small geometric primitives with very large cubes if the primitives happen to intersect the planes which separate the cube's children. A small triangle which crosses the center of the root cube, for example, will have to be rendered anytime the entire model is not hidden. To avoid this behavior, there are two basic choices. One alternative is to clip the problematic small polygons so they fit in much smaller octree cells. This has the disadvantage of increasing the number of primitives in the database. The other alternative is to place some primitives in multiple octree cells. This is the one we have chosen to implement. To do this, we modify the recursive construction of the octree as follows. If we find that a primitive intersects a cube's dividing planes, but is small compared to the cube, then we no longer associate the primitive with the whole cube. Instead we associate it with all of the cube's children that the primitive intersects. Since some primitives are associated with more than one octree node, we can encounter them more than once during rendering. The first time we render them, we mark them as rendered, so we can avoid rendering them more than once in a given frame.

3.2 Image-space Z pyramid

The object-space octree allows us to cull large portions of the model at the cost of scan-converting the faces of the octree cubes. Since the cubes may occupy a large number of pixels in the image, this scan conversion can be very expensive. To reduce the cost of determining cube visibility, we use an image-space Z pyramid. In many cases, the Z pyramid makes it possible to conclude very quickly a large polygon is hidden, making it unnecessary to examine the polygon pixel by pixel.

The basic idea of the Z pyramid is to use the original Z buffer as the finest level in the pyramid and then combine four Z values at each level into one Z value at the next coarser level by choosing the farthest Z from the observer. Every entry in the pyramid therefore represents the farthest Z for a square area of the Z buffer. At the coarsest level of the pyramid there is a single Z value which is the farthest Z from the observer in the whole image.

Maintaining the Z pyramid is an easy matter. Every time we modify the Z buffer, we propagate the new Z value through to coarser levels of the pyramid. As soon as we reach a level where the entry in the pyramid is already as far away as the new Z value, we can stop.

In order to use the Z pyramid to test the visibility of a polygon, we find the finest-level sample of the pyramid whose corresponding image region covers the screen-space bounding box of the polygon. If the nearest Z value of the polygon is farther away than this sample in the Z pyramid, we know immediately that the polygon is hidden. We use this basic test to determine the visibility of octree cubes by testing their polygonal faces, and also to test the visibility of model polygons.

While the basic Z-pyramid test can reject a substantial number of polygons, it suffers from a similar difficulty to the basic octree method. Because of the structure of the pyramid regions, a small polygon covering the center of the image will be compared to the Z value at the coarsest level of the pyramid. While the test is still accurate in this case, it is not particularly powerful.

A definitive visibility test can be constructed by applying the basic test recursively through the pyramid. When the basic test fails to show that a polygon is hidden, we go to the next finer level in the pyramid where the previous pyramid region is divided into four quadrants. Here we attempt to prove that the polygon is hidden in each of the quadrants it intersects. For each of these quadrants, we compare the closest Z value of the polygon in the quadrant to the value in the Z pyramid. If the Z-pyramid value is closer, we know the polygon is hidden in the quadrant. If we fail to prove that the primitive is hidden in one of the quadrants, we go to the next finer level of the pyramid for that quadrant and try again. Ultimately, we either prove that the entire polygon is hidden, or we recurse down to the finest level of the pyramid and find a visible pixel. If we find all visible pixels this way, we are performing scan conversion hierarchically.

A potential difficulty with the definitive visibility test is that it can be expensive to compute the closest Z value of the polygon in a quadrant. An alternative is to compare the value in the pyramid to the closest Z value of the entire polygon at each step of the recursion. With this modification, the test is faster and easier to implement, but no longer completely definitive. Ultimately, it will either prove that the entire polygon is hidden, or recurse down to the finest level of the pyramid and find a pixel it cannot prove is hidden. Our current implementation uses this technique. When the test fails to prove that a polygon is hidden, our implementation reverts to ordinary scan conversion to establish the visibility definitively.

3.3 Temporal coherence list

Frequently, when we render an image of a complex model using the object-space octree, only a small fraction of the octree cubes are visible. If we render the next frame in an animation, most of the cubes visible in the previous frame will probably still be visible. Some of the cubes visible in the last frame will become hidden and some cubes hidden in the last frame will become visible, but frame-to-frame coherence in most animations ensures that there will be relatively few changes in cube visibility for most frames (except scene changes and camera cuts). We exploit this fact in a very simple way with the hierarchical visibility algorithm. We maintain a list of the visible cubes from the previous frame, the *temporal coherence list*, and simply render all of the geometry on the list, marking the listed cubes as rendered, before commencing the usual algorithm. We then take the resulting Z buffer and use it to form the initial Z pyramid. If there is sufficient frame-to-frame coherence, most of the visible geometry will already be rendered, so the Z-pyramid test will be much more effective than when we start from scratch. The Z-pyramid test will be able to prove with less recursion that octree cubes and model polygons are hidden. As we will see in section IV, this can accelerate the rendering process substantially. After rendering the new frame, we update the temporal coherence list by checking each of the cubes on the list for visibility using the Z-pyramid test. This prevents the temporal coherence list from growing too large over time.

One way of thinking about the temporal coherence strategy is that we begin by guessing the final solution. If our guess is very close to the actual solution, the hierarchical visibility algorithm can use the Z pyramid to verify the portions of the guess which are correct much faster than it can construct them from scratch. Only the portions of the image that it cannot verify as being correct require further processing.

4 Implementation and Results

Our initial implementation of the hierarchical visibility algorithm is based on general purpose, portable C code and software scan conversion. This implementation uses the object-space octree, the image-space Z pyramid and the temporal coherence list. Even for relatively simple models the pure software algorithm is faster than traditional software Z buffering, and for complex models the acceleration can be very large.

In order to test the algorithm, we constructed an office module consisting of 15K polygons and then replicated the module in a three dimensional grid. Each module includes a stairway with a large open stairwell making it possible to see parts of the neighboring floors. None of the office walls extends to the ceiling, so from a high enough point in any of the cubicles, it is possible to see parts of most of the other cubicles on the same floor.

For simple models with low depth complexity, the hierarchical visibility method can be expected to take somewhat longer than traditional scan conversion due to the overhead of performing visibility tests on octree cubes and the cost of maintaining a Z pyramid. To measure the algorithm's overhead on simple models, we rendered a single office module consisting of 15K polygons at a viewpoint from which a high proportion of the model was visible. Rendering time for a 512 by 512 image was 1.52 seconds with the hierarchical visibility method and 1.30 seconds with traditional scan conversion, indicating a performance penalty of 17%. When we rendered three instances of the model (45K polygons), the running time was 3.05 seconds for both methods indicating that this level of complexity was the breakeven point for this particular model. Hierarchical visibility rendered nine instances of the same model (105K polygons) in 5.17 seconds, while traditional scan conversion took 7.16 seconds.

The chief value of the hierarchical visibility algorithm is, of course, for scenes of much higher complexity. To illustrate the point, we constructed a 33 by 33 by 33 replication of the office module which consists of 538 million polygons. The model is shown rendered in figure 1. 59.7 million polygons lie in the viewing frustum from this viewpoint, about one tenth of the entire model. Using the hierarchical visibility method, the Z-pyramid test was invoked on 1746 octree cubes and culled about 27% of the polygons in the viewing frustum. The bounding boxes of 687 cubes were scan converted which culled nearly 73% of the model polygons in the viewing frustum, leaving only 83.0K polygons of which 41.2K were front facing (.000076 of the total model) to be scan converted in software. On an SGI Crimson Elan, the entire process took 6.45 seconds. Rendering this model using traditional Z buffering on the Crimson Elan hardware took approximately one hour and fifteen minutes. Rendering it in software on the Crimson would probably take days.

The center left panel of figure 1 shows the depth complexity processed by the algorithm for the image in the upper left. The depth complexity displayed in this image is the number of times each pixel was accessed in a box visibility test or in Z-buffer polygon scan conversion. Note the bright regions corresponding to portions of the image where it is possible to see far into the model; these are regions where the algorithm has to do the most work. In this image, the average depth complexity due to box scans is 7.23, and due to polygon scan-conversion is 2.48 for a total of 9.71. The maximum depth complexity is 124. Dividing the number of times the Z pyramid is accessed by the number of pixels on the screen lets us assign a value of .43 for the “depth complexity” of the Z-pyramid tests. Thus, the total average depth complexity of Z-pyramid tests, box scans and polygon scans is 10.14. Note that this is not the depth complexity of the model itself, but only the depth complexity of the hierarchical visibility computation. Computing the true depth complexity of the scene would require scan converting the entire model of 538 million polygons in software, which we have not done. In the lower left of figure 1, we show the viewing frustum and the octree subdivision. The two long strings of finely divided boxes correspond to the two brightest regions in the depth complexity image. Note that the algorithm is able to prove that large octree nodes in the distance are hidden. In the lower right, we show the Z pyramid for the scene. Even at fairly coarse resolutions, the Z pyramid contains a recognizeable representation of the major occluders in the scene.

The office environment of figure 1 was chosen in part because it is a particularly difficult model for PVS methods. From every office cubicle in this environment, there are points from which almost every other cubicle on the same floor is visible. As a result, if the cubicles were used as cells in a PVS method, the potentially visible set for each cell would have to include nearly all the cells on its floor and many on other floors. Since each floor contains about 4 million polygons, the PVS methods would probably have to render many more polygons than the hierarchical method. In addition, the precomputation time for published PVS methods would be prohibitive for a model of this complexity. This model has 2000 times as many polygons as the model described by Teller and Sequin[15] which required 6 hours of pre-processing.

Admittedly, the replication of a single cell in the model means that it may not be a representative example, but it will be some time before people use models of this complexity without a great deal of instancing. The hierarchical visibility program we used for this example makes use of the replication in only two ways. First, the algorithm does not need to store half a billion polygons in main memory. Second, the algorithm only needs to consider a single cell in constructing the octree. These same simplifications would

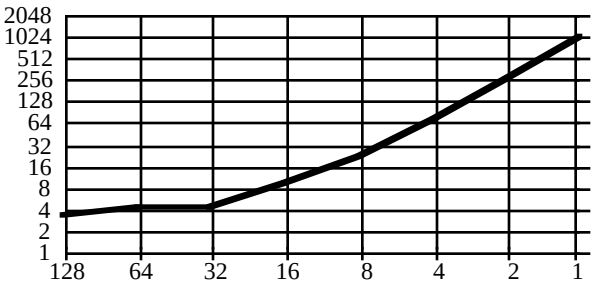

Fig. 3: Total time in seconds to render all windows as a function of the number of pixels on the side of each window.

apply to any complex model using a great deal of instancing.

Figure 2 shows the hierarchical visibility method applied to an outdoor scene consisting of a terrain mesh with vegetation replicated on a two-dimensional grid. The model used for the lower left image consists of 53 million polygons, but only about 25K polygons are visible from this point of view. Most of the model is hidden by the hill or is outside the viewing frustum. The corresponding depth complexity image for hierarchical visibility computations is shown at the top left. The algorithm works hardest near the horizon where cube visibility is most difficult to establish. This frame took 7 seconds to render with software scan conversion on an SGI Crimson. In the lower right, we show a model consisting of 5 million polygons. Even though the model is simpler than the model in the lower left, the image is more complicated and took longer to render because a much larger fraction of the model is visible from this point of view. This image took 40 seconds to render with software scan conversion on an SGI Crimson. The average depth complexity for the scene is 7.27, but it reaches a peak of 85 in the bright areas of the depth complexity image in the upper right. These outdoor scenes have very different characteristics from the building interiors shown in figure 1 and are poorly suited to PVS methods because (a) very few of the polygons are axis-aligned and (b) the cell-to-cell visibility is not nearly as limited as in an architectural interior. Nonetheless, the hierarchical visibility algorithm continues to work effectively.

4.1 Parallelizability and Image-space coherence

We have made our hierarchical visibility implementation capable of dividing the image into a grid of smaller windows, rendering them individually and compositing them into a final image. The performance of the algorithm as the window size is varied tells us about the parallel performance of the algorithm and the extent to which it makes use of image-space coherence. If, like most ray tracers, the algorithm made no use of image-space coherence, we could render each pixel separately at no extra cost. Then it would be fully parallelizable. At the other extreme, if the algorithm made the best possible use of image-space coherence, it would render a sizeable region of pixels with only a small amount more computation than required to render a single pixel. Then it would be difficult to parallelize. Note that if we shrink the window size down to a single pixel, the hierarchical visibility algorithm becomes a ray caster using an octree subdivision.

Figure 3 graphs the rendering time for a frame from a walkthrough of the model shown in figure 1 as a function of the window size. For window sizes from 32 by 32 on up, the curve is relatively flat, indicating that the algorithm should parallelize fairly well. For window sizes below 32 by 32, however, the slope of the curve indicates that the time to render a window is almost independent of the window size. The algorithm can, for example, render a 32 by 32 region for only slightly more than four times the computational expense of ray-casting a single pixel with this algorithm. Comparing the single pixel window time to the time for the whole image, we find that image-space coherence is responsible for a factor of almost 300 in running time for this example.

4.2 Use of graphics hardware

In addition to the pure software implementation, we have attempted to modify the algorithm to make the best possible use of available commercial hardware graphics accelerators. This raises some difficult challenges because the hierarchical visibility algorithm makes slightly different demands of scan-conversion hardware than traditional Z buffering. In particular, the use of octree object-space coherence depends on being able to determine quickly whether any pixel of a polygon would be visible if it were scan converted. Unfortunately, the commercial hardware graphics pipelines we have examined are either unable to answer this query at all, or take milliseconds to answer it. One would certainly expect some delay in getting information back from a graphics pipeline, but hardware designed with this type of query in mind should be able to return a result in microseconds rather than milliseconds.

We have implemented the object-space octree on a Kubota Pacific Titan 3000 workstation with Denali GB graphics hardware. The Denali supports an unusual graphics library call which determines whether or not any pixels in a set of polygons are visible given the current Z buffer. We use this “Z query” feature to determine the visibility of octree cubes. The cost of a Z query depends on the screen size of the cube, and it can take up to several milliseconds to determine whether or not a cube is visible. Our implementation makes no use of the Z pyramid because the cost of getting the required data to and from the Z buffer would exceed any possible savings. On a walk-through of a version of the office model with 1.9 million polygons, the Titan took an average of .54 seconds per frame to render 512 by 512 images. Because of the cost of doing the Z query, we only tested visibility of octree cubes containing at least eight hundred polygons. Even so, 36.5% of the running time was taken up by Z queries. If Z query were faster, we could use it effectively on octree cubes containing many fewer polygons and achieve substantial further acceleration. The Titan implementation has not been fully optimized for the Denali hardware and makes no use of temporal coherence, so these performance figures should be considered only suggestive of the machine's capabilities.

The other implementation we have that makes use of graphics hardware runs on SGI workstations. On these workstations, there is no way to inquire whether or not a polygon is visible without rendering it, so we use a hybrid hardware/software strategy. We do the first frame of a sequence entirely with software. On the second frame, we render everything on the temporal coherence list with the hardware pipeline. Then we read the image and the Z buffer from the hardware, form a Z pyramid and continue on in software. With this implementation, on the models we have tried, temporal coherence typically reduces the running time by a factor of between 1.5 and 2.

In the course of a walk-through of our office model, we rendered the frame in the upper left of figure 1 without temporal coherence, and then the next frame shown in the upper right of figure 1 using temporal coherence. The new polygons rendered in software are shown in magenta for illustration. For the most part, these are polygons that came into view as a result of panning the camera. The center right shows the depth complexity of the hierarchical computation for this frame. The image is much darker in most regions because the algorithm has much less work to do given the previous frame as a starting point. This temporal coherence frame took 3.96 seconds to render on a Crimson Elan, as compared with 6.45 seconds to render the same frame without temporal coherence.

Current graphics accelerators are not designed to support the rapid feedback from the pipeline needed to realize the full potential of octree culling in the hierarchical visibility algorithm. Hardware designed to take full advantage of the algorithm, however, could make it possible to interact very effectively with extremely complex environments as long as only a manageable number of the polygons are visible from any point of view. The octree subdivision, the Z pyramid and the temporal coherence strategy are all suitable for hardware implementation.

5 Conclusion

As more and more complex models become commonplace in computer graphics, it becomes increasingly important to exploit the available coherence in the visibility computation. Here we present an algorithm which combines the ability to profit from image-space coherence of Z-buffer scan conversion with the ability of ray tracing to avoid considering hidden geometry. It appears to be the first practical algorithm which materially profits from object-space, image-space and temporal coherence simultaneously. The algorithm has been tested and shown to work effectively on indoor and outdoor scenes with up to half a billion polygons.

The hierarchical visibility algorithm can make use of existing graphics accelerators without modification. Small changes in the design of graphics accelerators, however, would make a large difference in the performance of the algorithm. We hope that the appeal of this algorithm will induce hardware designers to alter future graphics hardware to facilitate hierarchical visibility computations.

Acknowledgements

We thank Frank Crow and the Advanced Technology Group at Apple Computer for supporting this research. We also thank Mike Toelle, Avi Bleiweiss, Helga Thorvaldsdottir and Mike Keller of Kubota Pacific Corporation for helping us test our algorithm on a Titan workstation.

References

[1] S. M. Rubin and T. Whitted. A 3-dimensional representation for fast rendering of complex scenes. *Computer Graphics*, 14(3):110-116, July 1980.

[2] A. Glassner. Space subdivision for fast ray tracing. *IEEE CG&A*, 4(10):15-22, Oct. 1984.

[3] D. Jevans and B. Wyvill. Adaptive voxel subdivision for ray tracing. *Proc. Graphics Interface '89*, 164-172, June 1989.

[4] T. Kay and J. Kajiya. Ray tracing complex surfaces. *Computer Graphics*, 20(4):269-278, Aug. 1986.

[5] 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.

[6] H. Hubschman and S. W. Zucker. Frame to frame coherence and the hidden surface computation: constraints for a convex world. *ACM TOG*, 1(2):129-162, April 1982.

[7] D. Jevans. Object space temporal coherence for ray tracing. *Proc. Graphics Interface '92*, Vancouver, B.C., 176-183, May 11-15, 1992.

[8] A. Glassner. Spacetime ray tracing for animation. *IEEE CG&A*, 8(3):60-70, March 1988.

[9] J. Chapman, T. W. Calvert, and J. Dill. Spatio-temporal coherence in ray tracing. *Proceedings of Graphics Interface '90*, 196-204, 1990.

[10] S. Badt, Jr. Two algorithms for taking advantage of temporal coherence in ray tracing *The Visual Computer*, 4:123-132, 1988.

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

[12] J. Airey. Increasing update rates in the building walkthrough system with automatic model-space subdivision. Technical Report TR90-027, The University of North Carolina at Chapel Hill, Department of Computer Science, 1990.

[13] J. Airey, J. Rohlf, and F. Brooks. 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):41-50, 1990.

[14] S. Teller and C. Sequin. Visibility preprocessing for interactive walkthroughs. *Computer Graphics*, 25(4):61-69, 1991.

[15] S. Teller and C. Sequin. Visibility computations in polyhedral three-dimensional environments. U.C. Berkeley Report No. UCB/CSD 92/680, April 1992.

[16] D. Meagher. Efficient synthetic image generation of arbitrary 3-D objects. *Proc. IEEE Conf. on Pattern Recognition and Image Processing*, 473-478, June 1982.

Figure Captions

Figure 1: A 538 million polygon office environment rendered with hierarchical visibility. Upper left: Rendered image. Center left: Depth complexity of the hierarchical visibility computation. Lower Left: Viewing frustum and octree cubes examined while rendering the image in the upper left. Lower right: Z pyramid used to cull hidden geometry. Upper right: Image rendered with temporal coherence. Polygons not rendered in the previous frame are shown in magenta. Center right: Depth complexity of the hierarchical visibility computation for the frame rendered using temporal coherence.

Figure 2: Lower left: Image of a 53 million polygon model (mostly hidden) rendered using hierarchical visibility. Upper left: Corresponding depth complexity for the hierarchical visibility computation. Lower right: Image of a 5 million polygon model. Upper right: Corresponding depth complexity for the hierarchical visibility computation.