OPTIMIZED HIERARCHICAL OCCLUSION CULLING FOR Z-BUFFER SYSTEMS

Draft, May 1999

Ned Greene

Abstract

We introduce a variation of hierarchical z-buffering that is highly optimized for conservative occlusion culling and show how it can be applied to dramatically reduce bandwidth requirements within a z-buffer pipeline. The algorithm is employed within a conservative culling stage of the pipeline that receives transformed geometry, uses optimized hierarchical z-buffering to cull occluded geometry, and passes visible geometry on to an ordinary z-buffer rendering stage. Optimizations to the tiling algorithm and z-pyramid employed by hierarchical z-buffering enable conservative culling to be performed a great deal more efficiently than with the standard algorithm. Other innovations relate to very efficient culling of occluded bounding boxes and a simple way to trade off image quality for rendering speed. The method will work effectively on virtually any polygonal model that can be traversed in approximately front-to-back order. On very deeply occluded scenes that we tested, the method reduced the depth complexity of primitives that need to be rendered to approximately two.

1 INTRODUCTION

Recent years have witnessed remarkable advances in the performance of inexpensive z-buffer accelerators, which in turn has fueled demand for high-performance rendering of increasingly complex scenes. Growing complexity raises the relative importance of *occlusion culling*, by which we mean culling of occluded geometry prior to rasterization. The purpose of occlusion culling is to achieve *output sensitivity*, where ideal output-sensitive performance depends on the *visible complexity* of the scene and is independent of overall scene complexity. If a high degree of output sensitivity is achieved, very complex scenes can be rendered in real-time, provided of course that their actual visible complexity is tractable.

Our objective is to devise highly output-sensitive occlusion culling for z-buffer systems that works effectively on general models, leverages standard z-buffer rendering hardware, and has a simple scene-management interface. We achieve this by modifying and optimizing the *hierarchical visibility algorithm* [GKM93], or HV for short. HV renders a scene that is organized in nested bounding boxes by traversing the boxes front to back, culling occluded boxes as they are encountered, and rendering the primitives in visible boxes with *hierarchical z-buffering*. This algorithm is highly output sensitive because it only needs to render the primitives in visible boxes and hierarchical z-buffering tiles individual primitives very efficiently. However, makers of z-buffer hardware have been slow to adopt the algorithm because a) implementing hierarchical z-buffering in hardware requires a major architectural revision, and b) scene management is complicated by communication delays incurred when box-visibility tests are performed by tiling hardware and by the need to maintain the scene model in nested boxes. Our objective is to overcome these shortcomings while retaining the efficiency and generality of the original algorithm.

In order to utilize standard z-buffering rendering hardware, we add a separate *culling stage* to the pipeline that culls occluded primitives and passes visible primitives on to standard rendering hardware. The culling stage employs a variation

of hierarchical z-buffering that is highly optimized for conservative culling. Optimized hierarchical z-buffering employs a modified z-pyramid that uses coverage masks in finest-level tiles to reduce storage requirements 10-fold. In combination with tiling optimizations, this modified z-pyramid reduces the traffic in z values required to perform conservative culling approximately 10-fold over standard hierarchical z-buffering.

To simplify scene management, we make the "tip" of the z-pyramid accessible to the host, which enables fast software culling of occluded bounding boxes. This eliminates communication delays associated with box-visibility tests, which are so efficient that there is no need to nest boxes in a spatial hierarchy.

As with the original HV algorithm, achieving high output sensitivity only requires that the scene be processed approximately front to back. Generally speaking, complying with this requirement is much easier than satisfying the scene-management requirements of other occlusion-culling methods. We demonstrate through simulation that our architecture achieves high output sensitivity for general models with respect to classic bandwidth bottlenecks in z-buffer pipelines, traffic in geometric data and z values. On very complex and deeply occluded models that we tested, the culling stage reduced the depth complexity of primitives that needed to be rendered to approximately two.

We also show that a simple change to the propagation procedure for hierarchical z-buffering enables error-bounded non-conservative culling with a "quality knob" mechanism that trades off image quality for rendering speed.

2 PREVIOUS WORK

Occlusion culling for z-buffer systems presents multiple challenges: output sensitivity, generality, simplicity, and effective use of hardware accelerators. Here we examine some existing occlusion-culling methods in the context of these objectives.

First we consider the *hierarchical visibility algorithm* [GKM93] on which the innovations explored in this article are based. Hierarchical visibility (HV) maintains the scene model in a spatial hierarchy (e.g., an octree) and a depth image in a z-pyramid. A scene is rendered with a recursive subdivision procedure beginning at the hierarchy's root node which determines the visibility of a node's bounding box with respect to the z-pyramid. Such a visibility query will be abbreviated *v-query*. If the box is visibl
e
Error!

For the limited class of models that can be effectively organized as "rooms with portals," occlusion culling is an elegantly solved problem [Air90,Tel92,Fun93]. Real-time rendering systems with software culling stages that feed z-buffer hardware have been demonstrated by Teller [Tel92] and Funkhouser and Sequin [Fun93]. Furthermore, geometry that will be coming into view can be anticipated with this method, so delays caused by paging of the scene model can be avoided. However, this approach does not cull general models effectively, which motivates the search for a more general method.

Another way to accelerate object-space culling is to incorporate the *v-query* operation into a z-buffer accelerator that reports whether a portal or bounding box is visible. Some Kubota [Tit93] and Hewlett-Packard [Sco98] workstations have this feature. Although hardware *v-query* effectively accelerates rendering in some cases, this method is much less effective than culling with a z-pyramid when portals or bounding boxes overlap deeply, as shown by simulation in section 6.

Another culling method that exploits z-buffer hardware is to create an occlusion image of foreground occluders which is then used for culling when the scene is rendered [Zha97,Zha98]. One clever feature is to control non-conservative culling with the opacity of the occlusion image. The main strength of this method is that it improves the performance of existing z-buffer accelerators. However, rendering in two passes adds complexity, effectiveness depends on being able to select efficient foreground occluders, and it is easier to implement non-conservative culling when the system maintains a z-pyramid, as described in section 4.3.

Model simplification (e.g. [Hop96]) and geometric compression (e.g. [Dee94]) are complimentary strategies for reducing geometry and memory traffic that can be used in combination with occlusion culling.

3 BANDWIDTH REQUIREMENTS FOR VISIBILITY COMPUTATIONS

Before introducing the proposed architecture we review block diagrams of visibility components within a conventional z-buffer architecture, *Arch_ZB* in figure 1a, and the analogous architecture for hierarchical visibility, *Arch_HV* in figure 1b, and summarize bandwidth generated by visibility computations within these architectures. With *Arch_ZB*, *geometry traffic* consists of all on-screen primitives (possibly excepting backfacing ones), which are transformed to image space and scan converted into a z-buffer. The *z-traffic* this generates (i.e., z-buffer memory traffic) is proportional to scene depth complexity (DC), falling between DC+1 and 2xDC, depending on the traversal order of primitives.

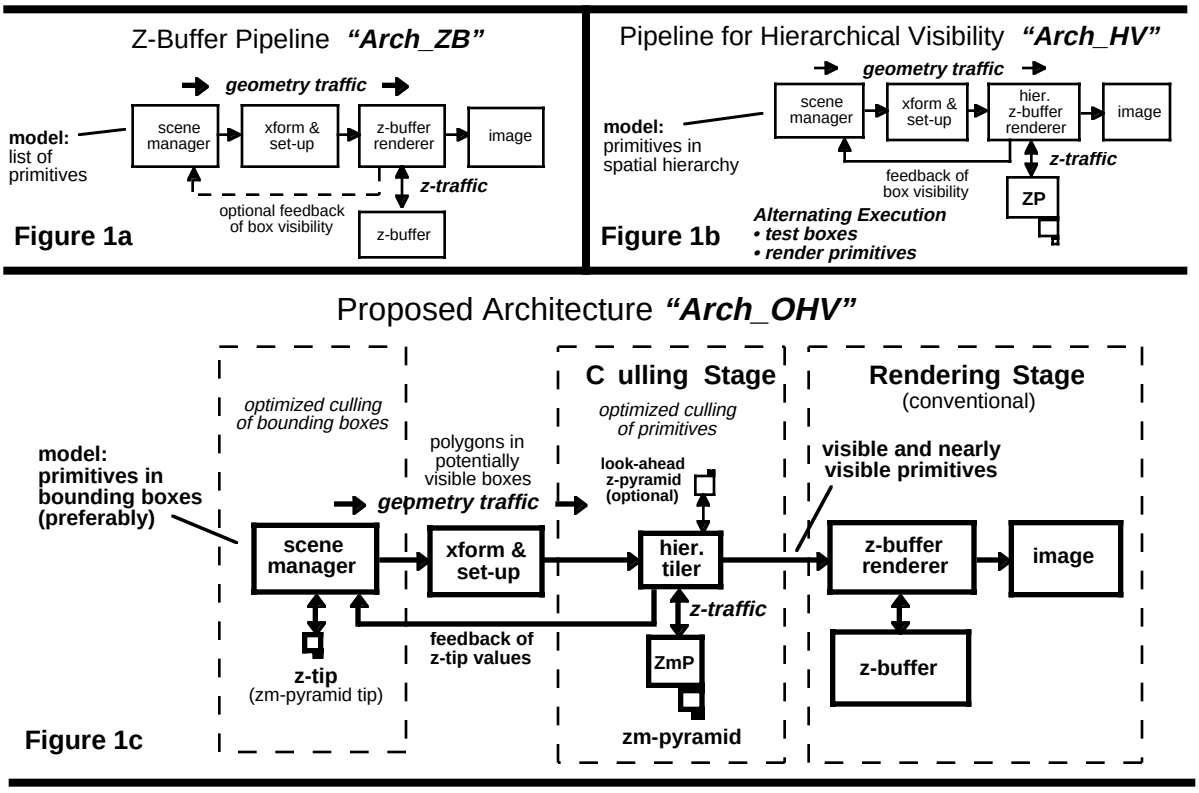
With *Arch_HV*, the scene manager maintains the scene model in a spatial hierarchy and runs the HV algorithm (summarized in the preceding section), with execution alternating between v-query tests on bounding boxes, results of which are reported on a feedback connection, and rendering of primitives in visible boxes. *Geometry traffic* consists of visible boxes and their children and the primitives in visible boxes. *Z-traffic* consists of the z-pyramid values accessed during hierarchical tiling of these boxes and primitives.

These properties make *Arch_HV* highly output sensitive, as confirmed by simulations in section 6 which show, for example, that for frames of a deeply occluded scene that we rendered, average *z-traffic* remained within a factor of 1.5 of the *ideal performance* for z-buffering of one read and one write per image sample. More generally, the simulations show that for both *geometry* and *z-traffic*, *Arch_HV's* bandwidth consumption is dramatically lower than *Arch_ZB's* when processing deeply occluded scenes.

Despite these big advantages in relative performance, there are practical drawbacks to implementing *Arch_HV* in hardware, foremost among them the cost of a major architectural revision which would scrap huge investments in the design of scan-conversion hardware. Secondly, if v-query tests on bounding boxes initiated by the scene manager are performed by hardware residing on a bus, communication delays will hamper performance. Finally, maintaining the scene model in a spatial hierarchy of nested bounding boxes complicates scene management and is problematic for some applications programs.

The architecture we propose addresses these concerns by employing conventional scan-conversion hardware in its rendering stage, by making values in the "tip" of the z-pyramid accessible to the scene manager so that box culling can be performed in software on the host, and by dispensing with nesting of bounding boxes. In addition, separating culling from rendering enables hierarchical z-buffering to be highly optimized for conservative culling

.



4 OPTIMIZED MULTI-STAGE OCCLUSION CULLING

Figure 1c shows the occlusion-culling architecture (*Arch_OHV*) that we propose to leverage conventional z-buffer hardware, simplify scene management, and perform highly output-sensitive rendering of general models. The philosophy of this architecture is to maximize the cost effectiveness of culling operations by culling in three stages, where computation and bandwidth costs rise progressively from stage to stage and the full cost of definitive visibility computations is paid only when necessary.

First, most occluded bounding boxes are efficiently culled in software by the scene manager using a concise pyramid of z values called the *z-tip*. Next, an optimized conservative *culling stage*, which receives only the primitives in potentially visible boxes, culls most occluded primitives using a compact variation of a z-pyramid called a *zm-pyramid*. Finally, the *rendering stage* only needs to render the visible and nearly visible primitives it receives from the culling stage.

In more detail, scenes are processed with a modified version of HV that assumes that scene primitives are organized in bounding boxes, which need not be nested. Preferably, the scene manager processes the bounding boxes in front-to-back order. Each box is tested for visibility against the z-tip. If this test fails to cull a box, the primitives it contains are sent through the pipeline, where they are transformed to image space before arriving at the culling stage.

Using an optimized variation of hierarchical z-buffering, the culling stage tiles primitives one-by-one into a zm-pyramid. Periodically, the coarsest levels of the zm-pyramid - the *z-tip* - are copied to the scene manager to enable box culling on the host. If tiling does not show that a primitive is occluded, the primitive is passed on to the rendering stage where it is rendered with conventional z-buffer hardware.

4.1 Optimized Hierarchical Z-Buffering for Conservative Culling

Here we describe optimizations to hierarchical z-buffering that enable the culling stage to perform conservative culling with far less work than the hierarchical polygon tiling algorithm presented in [Gre96a]. When modified for z-buffering by substituting a z-pyramid for the "coverage pyramid," the algorithm of [Gre96a] tiles polygons by Warnock-style subdivision [War69], performing polygon-cell overlap tests with "triage" coverage masks and performing hierarchical occlusion tests with the z-pyramid. Actually, this variation is not described in [Gre96a], but adapting the algorithm for z-buffering is straightforward, as just outlined. Although this algorithm culls occluded polygons very efficiently, it needs to subdivide to the image-sample level where a polygon is visible, performing one z read and one z write at each visible sample on the polygon. Moreover, definitive visibility computations require maintaining a full-precision depth value for each image sample.

With *conservative culling*, however, such definitive computations are not necessary, and we exploit this fact to greatly reduce computation, *z-traffic*, and storage requirements. Our method employs a novel data structure, a z-pyramid with coverage masks, which we call a *zm-pyramid* for short (ZmP), which reduces storage requirements and *z-traffic* 10-fold while accelerating tiling computations and retaining high culling efficiency.

As shown in figure 2, a zm-pyramid has the structure of an ordinary z-pyramid with NxN decimation, except that each finest-level NxN tile is stored as two z values and a coverage mask instead of an NxN array of z values. As shown in figure 3, z value *zfar_of_tile* indicates the farthest z for the whole tile, a *mask* indicates samples that have been covered by one or more polygons since *zfar_of_tile* was established, and *zfar_of_mask* is the farthest z value of these samples. The A-buffer [Car84] also uses coverage masks to expedite processing within image tiles, but the data structure we use is much more compact and specifically adapted for conservative culling.

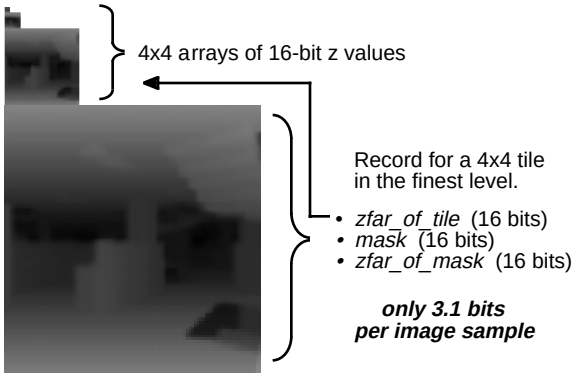


Figure 2

Structure of the "zm-pyramid" employed by the culling stage.

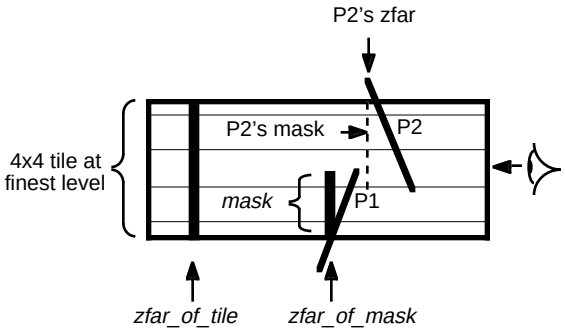


Figure 3

Schematic side view of a finest-level tile in the zm-pyramid.

In figure 3, suppose that a zfar value for the tile (*zfar_of_tile*) has been established before processing polygons P1 and P2. When P1 is encountered, the *mask* of visible samples that it covers is created and *zfar_of_mask* is set to the zfar value of these samples. Next, when P2 is processed, since it covers the tile collectively with the stored mask, a new zfar value is established for the tile, which is written to *zfar_of_tile* (this is the old value for *zfar_of_mask*). The tile's *mask* is set to P2's mask and *zfar_of_mask* is set to P2's zfar value.

In all, there are five cases that need to be considered when updating a tile record, which are schematically shown in figure 4 where dashed lines indicate

whether the zfar value of the visible samples covered by the polygon is nearer or farther than *zfar_of_mask* and whether the polygon's visible samples cover the tile in combination with the stored *mask*. The example of figure 3 corresponds to case C4.

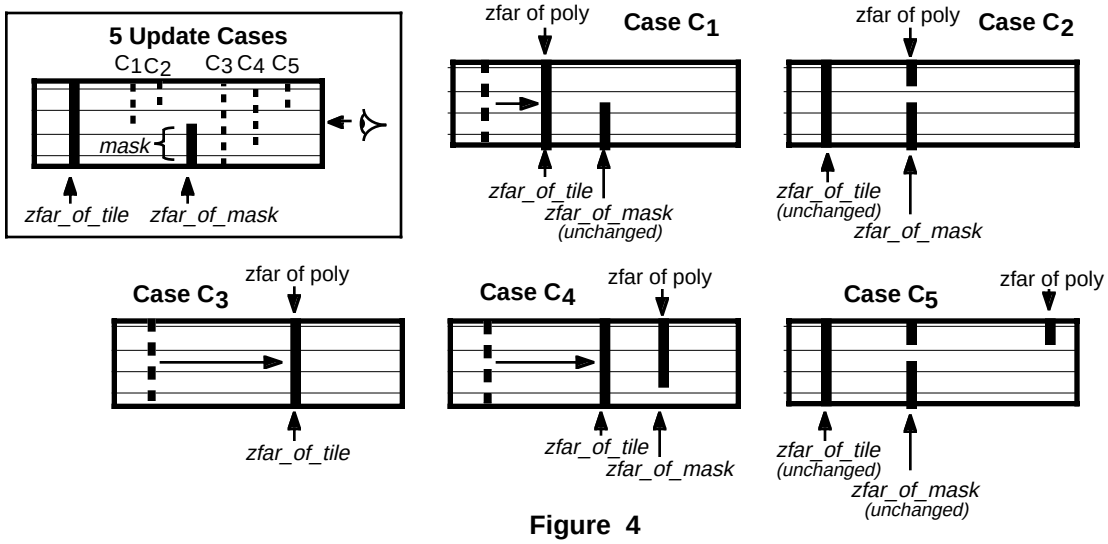


Figure 4

Schematic side views of finest-level tiles in the zm-pyramid showing how tile records are updated when a visible polygon is encountered.

To reduce storage requirements further, low-precision z values are employed throughout the zm-pyramid. As shown in figure 2, use of 16-bit values in a zm-pyramid with 4x4 decimation requires only approximately 3.1 bits per image sample.

In addition to conserving *z-traffic*, culling with a zm-pyramid saves a great deal of computation because sample-by-sample depth comparisons are not necessary within finest-level tiles. Rather, visibility operations are performed with coverage masks in combination with depth comparisons involving at most three depth values: *zfar_of_tile, zfar_of_mask,* and the zfar value of the polygon's visible samples.

Surprisingly, this simple data structure culls very effectively. Although *zfar_of_tile* does not advance as far at some tiles as it would in a standard z-pyramid, what matters is overall statistical performance, and as confirmed by simulations in section 6, even when the average screen size of polygons is small and many tiles are covered by three of more polygons, the zm-pyramid specified in figure 2 culled with an efficiency of approximately 90%, as efficiency is defined in section 6. Incidentally, other simulations reported in section 6 show that for the models we tested, efficient culling required maintaining sample-by-sample occlusion information. The alternative of culling with a "low-resolution z-pyramid," for example a z-pyramid missing it's finest level, culled a much lower fraction of occluded polygons.

4.2 Early Termination of Tiling

Another tiling optimization that can be exploited for conservative culling is "early termination" of tiling in situations where subdivision to the finest level cannot advance z values more than some threshold distance, call it *zdelta*. As illustrated in figure 5, if the *znear* value of a polygon that covers a tile is farther than the tile's *zfar* value offset by *zdelta*, the culling stage assumes that the polygon is visible and stops tiling in that region of the screen. Early termination reduces computation and z-traffic significantly while hardly impairing culling efficiency. The method saves work whether conservative culling is performed with a zm-pyramid or a standard z-pyramid.

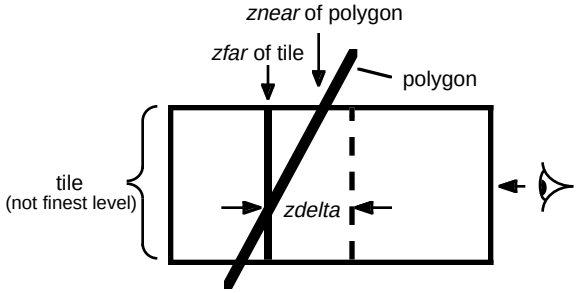


Figure 5

With "early termination" of tiling, tiling stops when continuing cannot advance z values more than a threshold distance, *zdelta.*

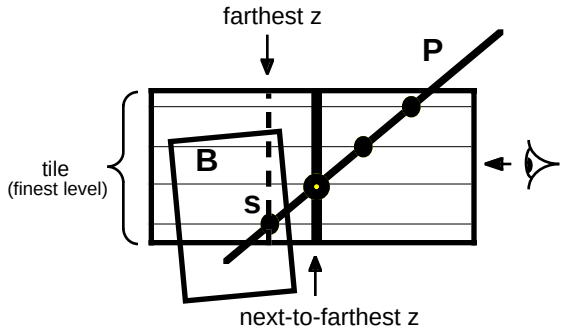


Figure 6

Propagating the next-to-the-farthest z value through the pyramid may result in culling objects that are visible at only one sample.

4.3 A "Quality Knob" for Non-Conservative Culling

When a scene is too complex to render at the desired frame rate, even when occlusion culling is performed, one way to accelerate rendering is to perform *non-conservative* culling of geometry that is only "slightly visible." Here we describe a very simple way to trade off image quality for rendering speed in systems that maintain a conventional z-pyramid.

Zhang et al. perform non-conservative culling by rendering foreground occluders into a *hierarchical occlusion map* (HOM), which is then used for image-space culling [Zha97,Zha98]. Aggressiveness of culling can be specified with an opacity value.

Non-conservative culling can be performed much more efficiently when the system maintains a z-pyramid, because implementation only requires changing the procedure for propagating z values through the pyramid. When hierarchical tiling determines visible samples within a finest-level NxN tile, instead of propagating the tile's farthest z value through the z-pyramid [GKM93], we propagate the *Eth-to-the-farthest* z value, where E is an *error limit* which can be set to any value from zero to N^2-1. With this simple change, tiling with standard hierarchical z-buffering will cull primitives within regions of the screen where they are visible at E or fewer samples within any finest-level tile, and the output image will have E or fewer errors within any NxN tile of samples. By an error at an image sample we mean that its color differs from a standard z-buffer image.

For example, figure 6 shows a finest-level 4x4 tile which has been covered by a polygon P. Assuming that the error limit is one, the next-to-the-farthest z value will be propagated when P is processed, which will later result in culling bounding box B within the tile, even though B is visible at one image sample, labeled **s**.

This non-conservative culling method provides a simple "quality knob" that enables sacrificing image quality in exchange for faster rendering. If E is 0, a standard error-free z-buffer image is created, if E is 1 there will be at most one error in any NxN tile of samples in the output image, and so forth. In practice, the usefulness of this method depends on how much acceleration is gained and how much image quality is sacrificed. The tradeoff is good when the depth image of a scene contains numerous "pinholes," because E or fewer pinholes within a finest-level tile will be "plugged" automatically. The quality knob is a useful mechanism for a certain class of scenes, producing approximate images considerably faster than standard hierarchical z-buffering. It is also possible to apply this method to culling with a zm-pyramid (rather than a conventional z-pyramid) by propagating *zfar_of_mask* rather than *zfar_of_tile* when the bit count of the *mask* is less than or equal to E, although culling will be less aggressive for a given value of E.

Comparing this approach to non-conservative culling with Zhang's method [Zha97][Zha98], the chief advantage is that it is not necessary to select and render foreground occluders in a separate rendering pass. In addition, since a z-pyramid contains actual sample-by-sample z values for all primitives rendered thus far, it culls a higher fraction of occluded geometry than an HOM. Finally, Zhang's method does not provide as much local control over image quality because the opacity parameter controls *average* error within regions of the screen but not distribution of error, which can result in culling of visible objects which cover entire NxN tiles in the output image.

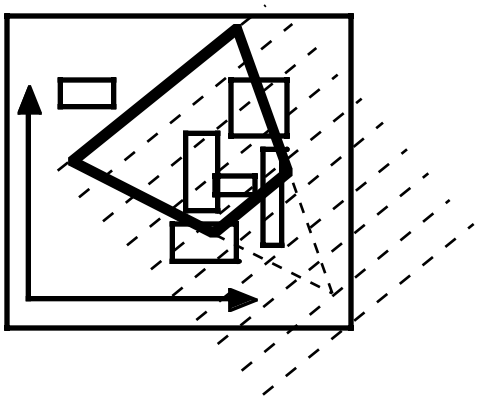


Figure7

Approximate front-to-back traversal can be facilitated by sorting bounding boxes into depth buckets.

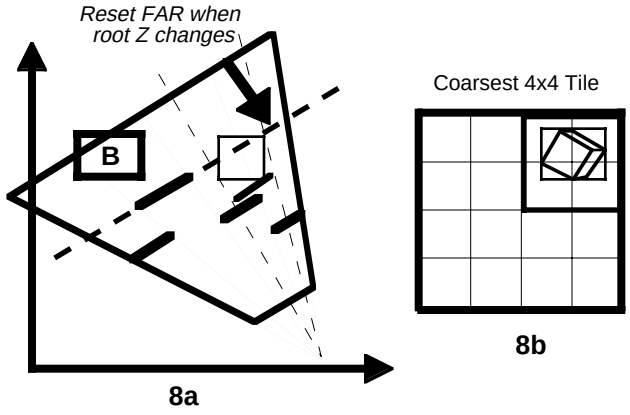


Figure 8

Culling with the tip of the z-pyramid.

4.4 Efficient Box-Culling in Software on the Host

Approximate front-to-back traversal of scene geometry is crucial to achieving the high output sensitivity of the HV algorithm. To accomplish this without maintaining nested boxes, the scene manager groups primitives into boxes and sorts the boxes into "depth buckets" based on the depth of their nearest vertex, as schematically illustrated for axis-aligned boxes in figure 7, and then traverses the buckets front to back. This method is a straightforward implementation detail that may have precursors in other systems.

To test a bounding box for occlusion by the z-tip, the scene manager computes its screen bounding box and determines a *znear* value by computing the depth of the

box's nearest corner. Next, the smallest MxN-cell region in pyramid that encloses the box is determined, and if the box's *znear* value is behind the z-tip value at each cell that the box overlaps, the box is culled. Similar box culling methods are described in [GKM93] and [Zha98]. The process is illustrated in figures 8a and 8b for a box within a 2x2-cell region in the coarsest 4x4 tile in the z-tip. In practice, we use 4x4 regions, so up to 16 depth comparisons are made per box.

The novel idea here is enabling the scene manager to harness the power of hierarchical culling with a z-pyramid without paying the cost of actually creating the pyramid, which requires tiling primitives. Rather, z-tip values are copied from the culling stage, which consumes very little bandwidth, particularly if values are copied only when they change.

As indicated in figure 8a, when the root value in the z-tip changes, this advances the far clipping plane, enabling culling of boxes behind the new FAR value (e.g. box B) with a simple cull-to-frustum test.

4.5 "Look Ahead" Computations

Now we consider an optional "look-ahead" feature of the conservative culling stage that reduces paging delays in scenes where the model is too complex to fit in main memory and the scene model is not compatible with established precomputed visibility methods [Air90,Tel92,Fun93]. If the culling stage includes a second "look-ahead zm-pyramid" (or z-pyramid) as shown in figure 1c, it is possible to determine what bounding boxes are likely to come into view by estimating where the view frustum is likely to be in the near future and tiling the corresponding "look-ahead frame." These look-ahead computations create only a zm-pyramid and not the corresponding image. Maintaining a separate look-ahead zm-pyramid enables these computations to be interleaved with rendering of ordinary frames. The method assumes that bounding boxes are stored in main memory but the primitives inside them may be in slower memory. With this interleaved approach, when a box in a look-ahead frame is visible but its primitives are not available, the culling stage resumes work on an the current frame while missing primitives are fetched. Look-ahead computations may be assigned low priority so they will be performed only when the processor and culling stage are not busy with more-pressing tasks. The effectiveness of the method will vary according to frame-to-frame coherence in the motion sequence and the number of look-ahead frames generated.

5 ARCHITECTURAL CONTEXT

Now we consider different architectural contexts for the proposed culling architecture, as a software culling stage (figure 9a), or as a hardware culling stage on a graphics card (figure 9b) or integrated with a processor having a z-buffer pipeline (figure 9c).

5.1 Software Culling on a Host Processor

First we consider the system of figure 9a where hierarchical culling is performed in software on a host processor. The processor performs scene management and culling, and sends potentially visible polygons to a conventional z-buffer accelerator. In this case, the processor can access the culling stage's zm-pyramid (ZmP) directly, so there is no need to maintain the z-tip separately. This is also true when the culling stage and z-buffer renderer are integrated with the processor, as diagrammed in figure 9c.

Within the software culling stage, tiling can be performed very efficiently with hierarchical polygon tiling where polygon-cell overlap tests are performed with coverage masks as described in [Gre96a]. This use of masks dovetails neatly with the use of masks in the zm-pyramid. The 10-fold reduction in z-traffic resulting from

using a zm-pyramid (compared with a traditional z-pyramid) is a big advantage that would be expected to improve performance substantially.

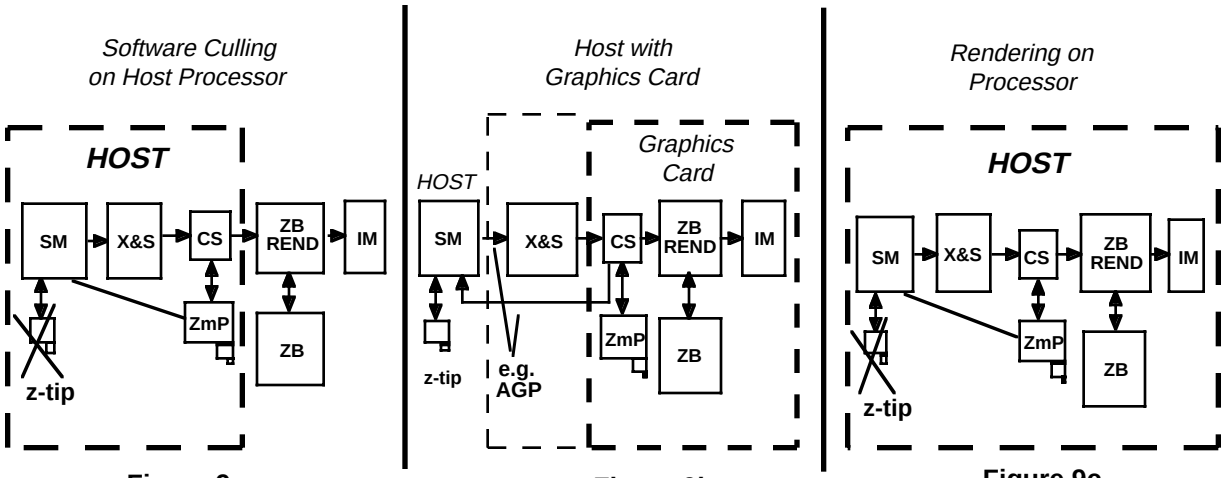


Figure 9a Figure 9b Figure 9c

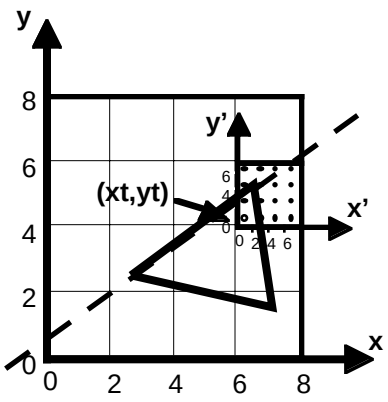
5.2 Hardware Implementations of the Culling Stage

Next we consider systems where the culling stage is implemented in hardware, for example on a graphics card as shown in figure 9b or as part of a hardware z-buffer pipeline integrated with a processor as shown in figure 9c. The basic subdivision algorithm for hierarchical polygon tiling is described in [Gre96a], which can be modified for hierarchical z-buffering as already described. 4x4 tiles are a good choice for tile size because they are small enough for high utilization and large enough to amortize the overhead of memory access. It is straightforward to modify the recursive tiling algorithm for tile-based processing and memory access by maintaining a stack of tile records.

Next, we describe a hierarchical method for accelerating computation of the linear equations defining a polygon in the context of tiling into a conventional z-pyramid. This approach can also be adapted to tiling into a zm-pyramid.

With custom hardware, hierarchical polygon tiling can be performed very efficiently by hierarchically evaluating a polygon's edge and depth equations. Within NxN tiles at the finest-level of the z-pyramid it is necessary to evaluate the polygon's linear equations on an NxN grid of samples points. For example, a triangle is defined by three edge equations of the form $Ax + By + C$ and a depth equation of the form $z = Ax + By + C$ (A, B, and C in the two equations are distinct, of course). Although these equations could be evaluated at all 16 grid points in parallel, this requires 128 multipliers which would consume considerable chip real estate.

To circumvent the need for general-purpose multiplication when these equations are evaluated, we use the hierarchical method illustrated in figure 10. In set-up computations, coefficients of the edge and depth equations of a polygon are computed relative to the coordinate frame (scaled as shown) of the smallest enclosing z-pyramid tile, which is where tiling commences. When the tile is recursively subdivided, the edge and depth equations are transformed to the child tile's coordinate frame using the formulas in figure 10. In these formulas, the expression $Axt + Byt + C$ (where (xt,yt) is the origin of the child tile) has already been computed at the parent tile, and since N is a power of two, evaluation requires only shifting. A similar algorithm for accelerating evaluation of linear equations on a pixel grid is presented in [Fuc85].



Edge Equation
parent tile: Ax + By + C = 0
child tile: Ax' + By' + C' = 0
where C' = N(Axt + Byt + C)

Depth Equation
parent tile: z = Ax + By + C
child tile: z = A'x' + B'y' + C'
where A' = A/N
B' = B/N
C' = Axt + Byt + C

Figure 10

Hierarchical evaluation of edge and depth equations. Coefficients are computed in the coordinate frame where tiling begins. When child tiles are processed, edge and depth equations are transformed to the child coordinate frame using the indicated formulas.

The advantage of this method is that edge and depth equations need to be evaluated only at small integers (e.g., between 0 and 8 in 4x4 tiles), rather than the at full range of sample coordinates of the image. Hence, terms Ax and By can be evaluated with custom logic that is a great deal more compact than the multipliers that would otherwise be required. This method is compatible with jitter if jittering is restricted to samples on an "oversize" integer grid within a pixel, say a 32x32 grid. Note that at coarser levels of the pyramid it is necessary to evaluate each edge and depth equation at only one corner of each cell. Depth equations are evaluated at the corner where the polygon's plane is nearest, which corresponds to the quadrant of the screen projection of a backfacing normal to the polygon. Similarly, the "normal" to an edge indicates which corner of a cell should be substituted into the edge's equation.

This method of transforming equations from parent to child tile can also be applied to Gouraud interpolation and interpolation of texture coordinates. Actually, it can be applied to any polynomial equation in N dimensions.

6 SIMULATION RESULTS

We implemented a high-level software simulator programmed in C in order to compare bandwidth requirements for three z-buffer architectures: z-buffering (*Arch_ZB*), hierarchical visibility (*Arch_HV*), and the proposed architecture having an optimized culling stage (*Arch_OHV*). Our primary test model was a publicly available model of a skyscraper consisting of cubicles alternating with open staircases, arranged in a lattice of cubic modules. Each cubic module consists of 9,336 polygons organized in an octree as described in [GKM93]. Octree boxes contain an average of approximately 25 polygons, stored in random order. Polygon tessellation is an artifact of radiosity computations and is unrelated to the octree subdivision, except at the coarsest levels. The modular nature of the model is a convenience that facilitates controlling model complexity, and it does not affect simulation results significantly.

This model was employed because it is a good example of a general model that challenges visibility algorithms. Unlike typical architectural models having highly

constrained room-to-room visibility or game environments that are designed to simplify visibility computations, this model's open geometry makes it possible to see deep into the scene, creating complex occlusion relationships and making it a poor candidate for methods which precompute room-to-room visibility.

The simulator generated 1024x1024-pixel images of polygonal scenes with 4x4 jittered samples within each pixel. Z-pyramids and zm-pyramids were organized in 4x4 tiles. When a polygon was hierarchically tiled, the first culling operation tested the visibility of its bounding box in the smallest enclosing 2x2-cell region of the pyramid.

For each of the three architectures the simulator measured workload at three classic bottlenecks in the graphics pipeline: a) *geometry_traffic*: the number of polygons and bounding boxes (if any) sent through the pipeline, b) per-sample *z-traffic*: the average number of reads and writes of z values per image sample, and c) *rendered_depth:* the depth complexity of polygons processed by the rendering stage. To assess the impact of depth complexity on performance we generated the six versions of the model shown in figure 11, which have depth complexities of 3.0, 6.6, 9.8, 16.0, 27.7, and 53.2 when viewed with the camera parameters of figure 11.

Figure 11

The six scenes used for the bandwidth simulations of Graph 1. Depth complexity (DC) varies from 3.0 to 53.2.

In Graph 1, *Bandwidth Graphs*, the horizontal axes plot log of average depth complexity of the scene, with the six vertical lines representing the six scenes of figure 11. The vertical axes in these graphs are also log-scale, indicating *geometry traffic*, *z-traffic*, and *rendered_depth*. For *Arch_ZB* and *Arch_HV*, *z-traffic* refers to the z-traffic generated within the rendering stage, so the terms *z-traffic* and *rendered_depth* are closely related. For *Arch_OHV*, *z-traffic* refers to z-traffic within the culling stage only, and to simplify the graphs, z-buffer traffic generated within the rendering stage is not represented. Chart 1, *Description of Bandwidth Graphs*, describes what each of the curves represents and should be referred to in interpreting the following results.

6.1 Relative Performance of Z-Buffering and Hierarchical Visibility

First we observe the dramatic bandwidth reductions of *Arch_HV* compared with *Arch_ZB*. With *geometry traffic*, *Arch_ZB* needs to send all front-facing primitives inside the view frustum, but *Arch_HV* only needs to send visible bounding boxes and their children and the primitives in visible boxes (only front-facing primitives for this model). The deeper the scene, the greater the relative advantage, with *geometry traffic* reduced by an order of magnitude at scene depth 16.

Likewise, *Arch_HV* generates far less *z-traffic* and *rendered_depth*, again roughly an order of magnitude less at scene depth 16. With *Arch_HV*, *z-traffic* stays within a factor of 1.5 of the *ideal bandwidth* for z-buffering of one read and one write per image sample (bold dashed line in the *z-traffic* graph), and likewise, *rendered_depth* stays within a factor of 1.3 of the *ideal performance* corresponding

to the depth complexity of polygons which are visible in the output image (bold dashed line in the *rendered depth* graph).

6.2 Performance of Optimized Hierarchical Visibility

Now that the performance of the standard HV algorithm has been established, we use this as a baseline to measure the relative performance of *Arch_OHV* in which (a) the scene manager culls bounding boxes using the z-tip, (b) a conservative culling stage culls most remaining occluded geometry using optimized hierarchical z-buffering, and (c) a conventional z-buffer rendering stage establishes definitive sample-by-sample visibility for the remaining visible and nearly visible primitives. In the simulation of *Arch_OHV*, bounding boxes where traversed front to back and every box that intersected the view frustum was tested for visibility against the z-tip (finest level: 64x64) using the test illustrated in figure 8. The culling stage employed the zm-pyramid specified in figure 2, and to reduce tiling computations we used "early termination" with a *zdelta* value of 5% of the distance between the near and far clipping planes.

The efficiency of culling with *Arch_OHV* is apparent in the *z-traffic* panel of Graph 1. For our test scenes, which range from simple to complex, this log-scale graph shows that optimized hierarchical z-buffering performed within *Arch_OHV* (curve OHV) generates only about 10% of the *z-traffic* generated by standard hierarchical z-buffering (curve HV), and between .5% and 3% of the z-traffic generated by traditional z-buffering. These low figures show that conservative culling can be performed with a great deal less work than definitive sample-by-sample visibility computations, and that our optimized culling methods accomplish this objective.

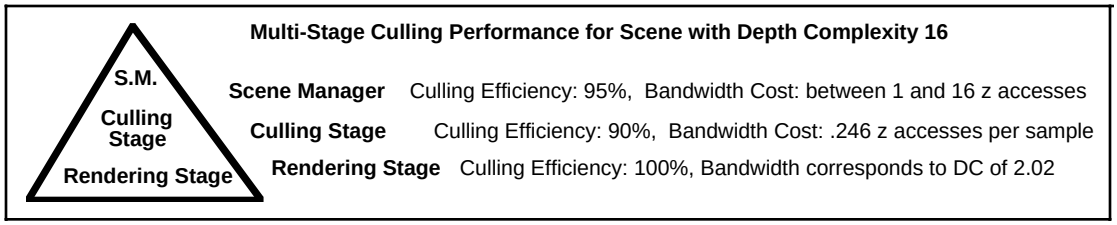


Figure 12

Figure 12 summarizes culling performance for the scene with depth 16. The scene manager culled bounding boxes using the z-tip (having 4x4, 16x16, and 64x64 levels), performing between 1 and 16 depth comparisons per box. This culled 95% of occluded boxes. Values in the z-tip changed 5,160 times in the course of rendering the scene, so the total amount of information which needs to be copied to the host was only about 10K bytes (each z value is 16 bits). These figures indicate that software culling of bounding boxes on the host is fast and effective.

The culling stage culled occluded primitives with an efficiency of 90%, by which we mean that if the culling stage performed visibility tests definitively, the depth of polygons sent to rendering stage would be 90% of the corresponding figure for this conservative culling stage. This efficiency is achieved with only .246 z accesses per image sample using the zm-pyramid. This low bandwidth figure indicates the high efficiency of optimized hierarchical z-buffering.

The rendering stage performs definitive z-buffering on polygons whose collective depth averages 2.02, which is within a factor of 1.3 of the depth of polygons which are actually visible in the output image. Summing up, at a bandwidth cost of approximately .25 accesses per image sample, *Arch_OHV* reduces the effective depth of this scene 8-fold, from 16 to 2.02.

6.3 Relative Performance of Z-Buffering and *Arch_OHV*

Now we summarize the relative performance of *Arch_ZB* and *Arch_OHV*. For our test scenes, Graph 2 compares *geometry traffic*, *z-traffic*, and *rendered depth* for the two architectures, where the OHV curves have been normalized as a fraction of the ZB curves, which are represented by the top line. For our test models, 2x reduction in *rendered_depth* occurs at scene depth 3.5, 4x occurs at depth 7.7, 8x occurs at depth 17, and 16x occurs at depth 33. The corresponding depths for reduction in *geometry_traffic* are 5.4, 10.2, 14, and 19. Assuming that one of these quantities is the actual bottleneck in the system, these figures offer a rough estimate of actual performance increases that could be expected. *Z-traffic* within the culling stage is so low that it is unlikely to be a bottleneck.

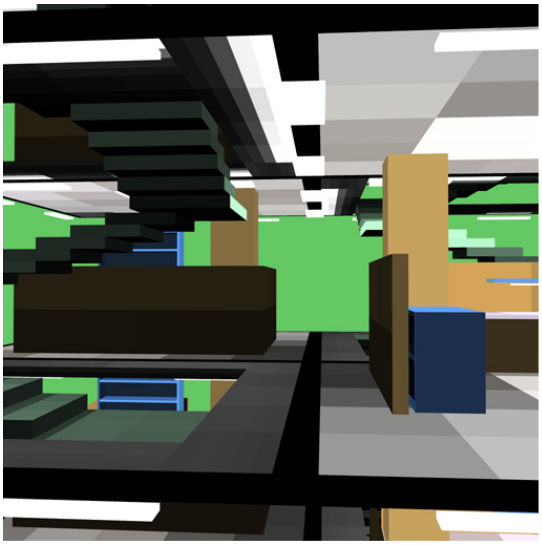


Figure 13

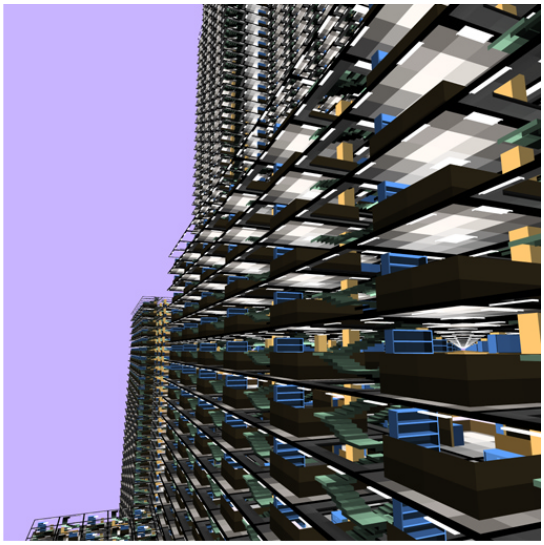


Figure 14

Optimized hierarchical visibility (*Arch_OHV*) generates less traffic in z values rendering the scene on the right (depth complexity 41.5) than ordinary z-buffering generates rendering the scene on the left (depth complexity 3.0). The culling stage within *Arch_OHV* generates only .58 z accesses per image sample and it reduces the depth complexity of primitives that need to be rendered to 2.28.

To verify that our basic conclusions are also valid for scenes without large foreground occluders we also ran simulations on the scene of figure 14 and plotted *Arch_HV* performance in blue circles and *Arch_OHV* performance in red circles in Graph 1, placing these circles on dashed lines at depth 41.5, the average depth complexity of this scene. Note that *geometry traffic* and *rendered depth* for *Arch_OHV* are nearly as low as for *Arch_HV*, and that *z-traffic* for *Arch_OHV* remains a great deal lower than for *Arch_HV*.

It is revealing to observe that traditional z-buffering generates more *z-traffic* rendering figure 13 (same as left-hand panel of figure 11) than *Arch_OHV* generates in both the culling and rendering stages when rendering figure 14, even though the latter scene has 13.8 times the depth complexity. In terms of bandwidth generated by conservative visibility operations in the culling stage, the *z-traffic* generated by rendering figure 14 with optimized HV is only about 1/7 of that generated by z-buffering figure 13.

Next, simulation of hierarchical z-buffering without box culling (curve HZ) showed that z-traffic increases only slightly compared with the HV curve, remaining within a factor of 1.65 of the *ideal bandwidth* of one read and one write per image sample. As with *Arch_ZB*, we used the octree to cull to the view frustum and traverse the scene in approximately front-to-back order, but of course hierarchical z-

buffering does not require organizing primitives in bounding boxes. The problem with hierarchical z-buffering without culling of bounding boxes is that all on-screen front-facing polygons must be sent through the pipeline, so *geometry_traffic* is likely to be a bottleneck.

To assess the importance of traversal order, we also simulated a variation of hierarchical visibility in which bounding boxes were traversed in random order instead of front to back. The resulting curves, labeled RAND, show that even with random traversal, *geometry_traffic* and *rendered_depth* are reduced substantially compared with *Arch_ZB*.

We repeated the simulations presented in Graph 1 for point-sampled images and determined that the *geometry traffic* and *rendered depth* curves were nearly identical, but that the *z-traffic* curves for OHV were higher by approximately a factor of two (but still many times lower than the corresponding HV curves). We also simulated *z-traffic* assuming tile-based access to z-buffers and z-pyramids, which raised overall *z-traffic* but did not significantly change the relative performance of the three architectures.

6.4 Ineffectiveness of Alternative Culling Methods

To show the limitations of box culling by traditional scan conversion, we substituted traditional z-buffer scan conversion for hierarchical z-buffering within *Arch_HV*. In this case, *geometry traffic* and *rendered depth* are the same as for HV but *z-traffic* is a great deal higher, as indicated by curve ZBbc ("Z-Buffer box culling"). Despite the front-to-back traversal of octree bounding boxes, exploiting box nesting, stopping the scan conversion at the first visible sample, and adjusting construction of the octree to maximize performance, *z-traffic* was a great deal higher than when culling with a z-pyramid, because bounding boxes overlap deeply on the screen and hierarchical culling is much more efficient. In short, for general models, box culling can be performed much more efficiently with hierarchical z-buffering than with traditional z-buffer scan conversion.

To determine if it is necessary for the culling stage's z-pyramid to maintain occlusion information at the full-resolution of the output image, we implemented a low-resolution tiling algorithm that maintained just one z value at each pixel in the oversampled image and overwrote a pixel's z value only if a polygon covered the entire pixel. This tiling algorithm was substituted into *Arch_HV* and the scenes were rendered with front-to-back traversal of the octree. We found that the "low-resolution" z-pyramid described above does not cull effectively, resulting in much higher *geometry traffic* and *rendered depth*. This experiment indicates that efficient culling of general models does require maintaining occlusion information on individual image samples.

7 CONCLUSION

We have analyzed two classic bottlenecks in z-buffer pipelines, traffic in geometric data and depth values, and shown that processing scenes with the hierarchical visibility algorithm can dramatically reduce bandwidth requirements compared to alternative methods. However, this approach has not been widely adopted by makers of hardware z-buffer pipelines because it requires a major architectural revision and scene management is complicated by communication delays associated with box culling and the need to maintain a spatial hierarchy.

This article introduces architectural innovations and optimizations that overcome these problems. Using these methods, it is possible to attain nearly the performance of the standard HV algorithm without modifying rendering hardware, without incurring delays when boxes are culled, and without maintaining a spatial hierarchy. Scene management is greatly simplified because all that is required to

achieve good performance is approximate front-to-back traversal of the scene. For virtually any scene traversed in this way, culling will reduce the depth complexity of primitives that need to be rendered to approximately two. Moreover, culling is performed extremely efficiency using optimizations to hierarchical z-buffering that reduce the traffic in z values generated by conservative culling by 10-fold compared with the standard algorithm. Taken together, these properties indicate that the proposed architecture could extend real-time rendering to a much broader class of scenes.

References

[Air90] J. Airey, "Increasing Update Rates in the Building Walkthrough System with Automatic Model-Space Subdivision and Potentially Visible Set Calculations," PhD Thesis, Technical Report TR90-027, Computer Science Dept., U.N.C. Chapel Hill, 1990.

[Car84] L. Carpenter, "The A-Buffer, an Antialiased Hidden Surface Method," *Proceedings of SIGGRAPH '84*, July 1984, 103-108.

[Cla76] J. H. Clark, "Hierarchical Geometric Models for Visible Surface Algorithms," *Communications of the ACM* 19(10), Oct. 1976, 547-554.

[Dee94] M. Deering, S. Schlapp, and M. Lavelle, "FBRAM: A new Form of Memory Optimized for 3D Graphics," *Proceedings of SIGGRAPH '94*, July 1994, 167-174.

[Fuc85] H. Fuchs, J. Goldfeather, J. Hulquist, S. Spach, J. Austin, F. Brooks, Jr., J. Eyles, and J. Poulton, "Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancements in Pixel-Planes," *Proceedings of SIGGRAPH '85*, July 1985, 111-120.

[Fun93] T. Funkhouser and C. Sequin, "Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments," *Proceedings of SIGGRAPH '93*, Aug. 1993, 247-254.

[GKM93] N. Greene, M. Kass, and G. Miller, "Hierarchical Z-Buffer Visibility," *Proceedings of SIGGRAPH '93*, July 1993, 231-238.

[Gre96a] N. Greene, "Hierarchical Polygon Tiling with Coverage Masks," *Proceedings of SIGGRAPH '96*, Aug 1996.

[Gre96b] N. Greene, "Naked Empire," ACM Video Review Issue 115: The Siggraph '96 Electronic Theatre," Aug 1996, 65-74.

[Hop96] H. Hoppe, "Progressive Meshes," *Proceedings of SIGGRAPH '96*, Aug 1996, 99-108.

[Mea82] D. Meagher, "The Octree Encoding Method for Efficient Solid Modeling," PhD Thesis, Electrical Engineering Dept., Rensselaer Polytechnic Institute, Troy, New York, Aug. 1982.

[Sco98] N. Scott, D. Olsen, E. Gannett, "An Overview of the VISUALIZE fx Graphics Accelerator Hardware," *The Hewlett-Packard Journal*, Vol. 49, No. 2, May 1998, 28-34.

[Tel92] S. Teller, "Visibility Computations in Densely Occluded Polyhedral Environments," PhD Thesis, Univ. of California at Berkeley, Report No. UCB/CSD 92/708, Oct. 1992.

[Tit93] "Denali Technical Overview," Kubota Pacific Computer Inc., Jan 1993.

[War69] J. Warnock, "A Hidden Surface Algorithm for Computer Generated Halftone Pictures," PhD Thesis, TR 4-15, Computer Science Dept., Univ. of Utah, June 1969

[Zha97] H. Zhang, D. Manocha, T. Hudson, and K. Hoff, "Visibility Culling Using Hierarchical Occlusion Maps," *Proceedings of SIGGRAPH '97*, Aug. 1997, 77-88.

[Zha98] H. Zhang, "Effective Occlusion Culling for the Interactive Display of Arbitrary Models," PhD Thesis, Computer Science Dept., U.N.C. Chapel Hill, 1998.

GRAPH 1 BANDWIDTH GRAPHS

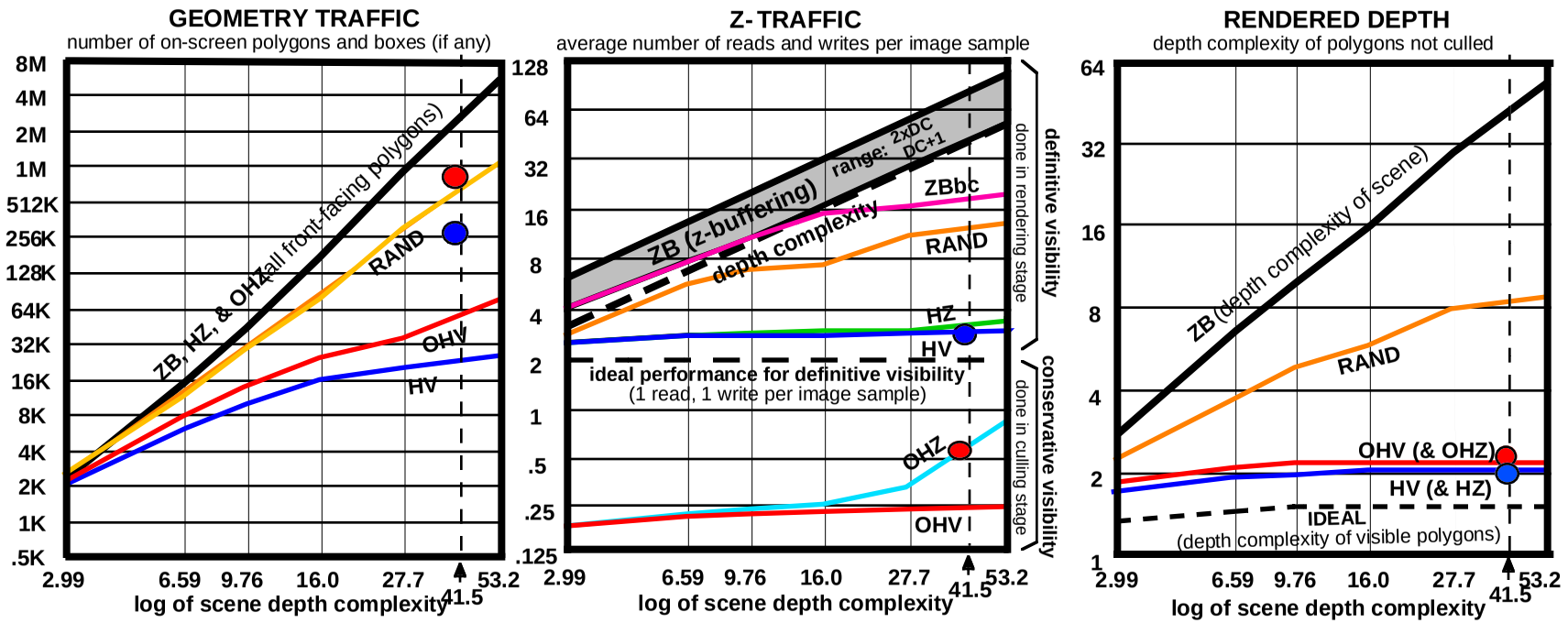


CHART 1 DESCRIPTION OF BANDWIDTH GRAPHS

LABEL color	DESCRIPTION	*geometry traffic* (**GT**), *z-traffic* (**ZT**), *rendered depth* (**RD**)	
DC	**depth complexity of scene**		
ZB	**naive z-buffering** model: polys organized in boxes to enable culling to view frustum	GT ZT RD	# of front-facing polys in boxes that intersect frustum (gray region) ZT generated by z-buffering GT DC
HV	**hierarchical visibility algorithm** model: polys organized in octree, octree traversed front to back	GT ZT RD	# vis. octree boxes & children + # polys in vis. boxes ZT generated by hier. tiling of GT DC of polygons not culled by hier. tiling
HZ	**hierarchical z-buffering** model: same as for **ZB**	GT ZT RD	same as **ZB** ZT generated by hier. tiling of GT DC of polygons not culled by hier. tiling
OHV	**optimized hierarchical visibility** model: polys organized in boxes (not nested)	GT ZT RD	# of polys in vis. boxes ZT generated by optimized conservative hier. tiling DC of polygons not culled by culling stage
OHZ	**optimized hierarchical z-buffering** model: same as **ZB**	GT ZT RD	same as **ZB** ZT generated by hier. tiling of GT DCof polygons not culled by culling stage
RAND	**hier. visibility - random traversal** model: polys organized in boxes	GT ZT RD	# of boxes plus # of polys in vis. boxes ZT generated by hier. tiling of GT DC of polygons not culled by hier. tiling
ZBbc	**z-buffering with *box culling*** (hier. vis. without a ZP) model: same as **HV**	GT ZT RD	same as **HV** (not labelled in GT graph) ZT ~ z-buffering of boxes and polys in vis. boxes same as **HV** (not labelled in RD graph)

GRAPH 2

Relative Performance
Z-Buffering vs. Optimized Hierarchical Visibility