Adaptive Hierarchical Visibility in a Tiled Architecture

Feng Xie [謝丰] and Michael Shantz
Intel Corporation

Abstract

This paper describes a method for occlusion culling in a tiled 3D graphics hardware architecture. Adaptive hierarchical visibility (AHV) is a simplified method for occlusion culling that is integrated into a tiled architecture for hardware rendering. AHV constructs a list of polygon bins for each tile where the bins are bucket sorted in order of increasing depth or Z. Polygon bins are rendered starting with the bin closest to the viewer. After some number of bins are rendered, a one layer, hierarchical Z-buffer (HZ) is constructed from the Z-buffer thus far accumulated for the rendered bins. Subsequent bins are rendered by first testing their polygons against the HZ to see if they are hidden. AHV is far simpler to implement in hardware and gives performance that matches or surpasses progressive hierarchical visibility (PHV) methods which update the HZ for each rendered pixel. Results show that AHV is superior on scenes with high depth complexity and small polygons. For tiles of widely ranging statistics, AHV competes surprisingly well with PHV. It offers dramatic performance improvement on low cost hardware for scenes of high depth complexity.

CR Categories and Subject Descriptors: I.3.1 [Computer Graphics]: Hardware Architecture; I.3.7 [Computer Graphics] Visible Line/Surface Algorithms

Key Words and Phrases: Visibility culling, hierarchical z buffer, occlusion culling

1 Introduction

In recent years the size and complexity of the graphical databases have been rapidly increasing for many interactive 3D graphics applications. These 3D graphics models typically have high depth complexity, i.e. a given pixel is rendered many times due to many overlapping polygons, and only the object closest to the viewer ends up being visible. Identifying and culling these occluded objects represents a huge performance improvement opportunity. Low cost graphics accelerators have not yet been able to incorporate effective occlusion culling. Previously proposed occlusion culling algorithms have been too complex for integration into low cost architectures. This paper presents a simple and effective occlusion culling method, namely AHV, for low cost graphics hardware.

Advanced architectures must address both the *computational* load of transformations, texturing, and lighting, as well as the increasingly important *bandwidth* load of accessing the geometry, texture, and visibility data. Graphics accelerators for PCs must address these computation and bandwidth issues while keeping the hardware cost very low. High output bandwidth implies high cost memory for the Z-buffering and anti-aliasing. This cost can be kept down by using a tiled architecture that renders the scene one tile or chunk at a time and reuses the fast expensive memory for each tile. AHV is specifically designed for a tiled architecture, adds little complexity to the pipeline, and it reduces both computation and bandwidth when rendering high depth complexity scenes.

1.1 Contribution

This paper presents an adaptive hierarchical visibility algorithm (AHV) integrated into a tiled hardware graphics pipeline architecture. For each tile it automatically and adaptively selects portions of the model as occluders using a bucket sort in Z. A hierarchical Z-buffer is constructed after accumulating critical coverage within the tile. Critical coverage is a simple heuristic that exploits frame to frame visibility coherence. This hierarchy is built and used for visibility testing only if a simple test indicates that it will be worthwhile. The method aims to be simple enough for low cost hardware implementation yet effective enough to give significant performance improvement in scenes of high depth complexity.

Compared to the traditional Z-buffer method, AHV can reduce both the computation and bandwidth costs. Compared to a deferred shading pipeline (where the texturing and shading are not computed if the Z-buffer test fails), AHV can reduce the computation cost with little additional bandwidth cost. To the best of our knowledge, AHV is the first algorithm to integrate occlusion maps into a tiled architecture.

1.2 Background

View frustum culling [4] and hidden surface removal [9,24] have long been used in 3D graphics. The Z-buffer [1,2] is surprisingly persistent as the basic hidden surface removal method for modern PC graphics accelerators despite its brute force approach and lack of support for occlusion culling.

Software algorithms for static scenes, that perform an object space preprocessing operation to construct a binary space partitioning (BSP) data structure [10], or a potentially visible set (PVS) for cell-to-cell visibility in an object space octree [24,18], have been used to perform visibility culling. Major performance improvements are achieved by identifying and eliminating those polygons that can not be seen from some regions of space. This saves the cost of scan converting, shading and texturing those polygons.

Another class of object space methods for dynamic data sets involves dynamic identification of convex occluding objects and identifies spaces where objects are occluded [5,17]. Alternatively, portals such as windows and doors are used to produce culling planes in a manner similar to view frustum culling [18]. These object space methods have difficulty in building up or identifying those collections of small objects that together may form an effective occluding assemblage.

Image space hierarchies have been used for anti-aliased texture mapping [25], hierarchical Z-buffers and z tests for visibility culling [12,13,14], and area coverage trees [15]. The combination of an object space octree for rendering front to back, and an image space z pyramid for visibility culling [14], delivers the best occlusion results, albeit at significant cost. Progressive hierarchical Z-Buffering updates the HZ with each rendered pixel and thus provides the most "exact" hierarchical Z-buffer method. This approach would require significant changes to a graphics hardware pipeline and it bears the high cost of a full screen hierarchical Z-buffer. We are not aware of any hardware implementation.

An alternative to object space sorting is a more ad-hoc method of occluder selection combined with an image space hierarchical occlusion map [26]. This algorithm is a hybrid solution that can be implemented in software. Unfortunately, it is not well suited to commercial graphics hardware.

Tiled architectures have been thoroughly analyzed for cost benefit tradeoffs. The statistics of polygon coverage and redundancy have been modeled [3,6,7,8,11,16,20,21,22]. This paper revisits hierarchical Z-buffering and evaluates its potential costs and benefits in a tiled architecture. The unique properties of a tiled architecture allow the integration of a simplified, adaptive version of hierarchical Z-buffering. The result reduces both computation and bandwidth costs significantly and can be implemented with a fairly small number of changes to the pipeline.

2 Adaptive Hierarchical Visibility

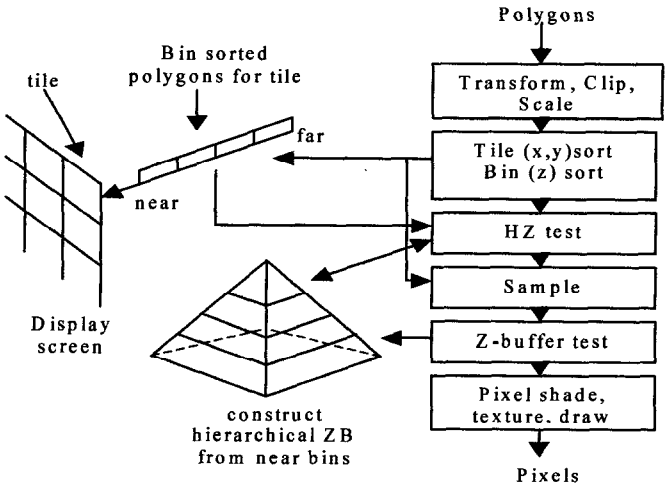


Figure 1. The pipeline on the right bucket sorts polygons into tiles and at the same time bucket sorts into z bins within each tile. For each tile some number of bins are drawn then the resulting Z-buffer is used to construct a hierarchical Z -buffer.

2.1 Overview

Figure 1 shows a generalized graphics rendering pipeline and the associated AHV data structures. The display screen is partitioned into tiles. Each tile is 128 by 128 pixels, although the size may be different based on memory costs and other factors. The algorithm proceeds as follows. For a given point in time, a display frame is generated by sorting the polygons of the scene into the x,y tiles that they overlap. The polygons for a tile are also z sorted into bins using a bucket sort. The tiles are then rendered sequentially. To render a tile, polygons are rendered from the nearest polygon bins using an ordinary Z-buffer (ZB). This continues until a coverage parameter reaches a threshold value. Then an HZ is constructed from the Z-buffer. Polygons from subsequent bins undergo a visibility test against the HZ before they are rendered. They are not scan-converted or drawn if this test shows that they are completely hidden.

The sections below will describe the algorithm in more detail.

2.2 Tile and Z Bin Sorting

In addition to dividing the screen into x,y tiles as in traditional chunking, each screen tile is further bucket sorted into bins using a z (depth) value. Each tile now has its own list of z sorted bins of polygons. For big polygons, exact triangle binning is performed using a computation of the intersection of the triangle with the tile. Exact binning for large triangles can reduce the computation and bandwidth overhead of overlap regardless of whether or not AHV is implemented.

The estimated z value (EZ) used to sort a polygon is defined as the depth in Z-buffer coordinates of the nearest point of the polygon that exists within the x,y extent of the tile. If the polygon is entirely contained by the tile, the value is simply the smallest z value of all the vertices. If the polygon is small, this value is still used even if the triangle straddles multiple tiles. For the exact binning of big polygons, the z value is the minimum z of the intersection points of the triangle with the tile edges plus any triangle vertices lying within the tile. The x,y bounding box (Ebox) of the area of overlap between the triangle and the tile is computed as follows. If the triangle is small and straddles one or more tiles, the intersection of the triangle's bounding box with the tile is used. For large triangles the bounding box of the exact intersection between polygon and tile is used.

The EZ and Ebox of the triangle will eventually be used for testing against the HZ for occlusion. The binning operation is implemented by the following code that also accumulates depth distribution statistics and the total estimated overlap area of triangles with the tile. The total estimated overlap area is used later to determine whether and where to construct the HZ. The depth distribution statistics are used in the next frame to setup Z buckets:

```
for each triangle, tri, in the scene  {
    for each tile, tile, the triangle overlaps  {
        tri.EZ          = EstimateZ( tri, tile);
        tri.Ebox        = EstimateBox( tri, tile);
        locate bin b in tile such that
            b.MinZ <tri. EZ< b.MaxZ;
        b.AddTriangle( tri);
        tile.MeanZ      += tri.EZ;
        tile.StdZ       += tri.EZ * tri.EZ;
        tile.TotalArea  += tri.Ebox.Area;
    }
}
```

2.3 Review of Hierarchical Z-Buffer

After some number of bins of triangles for the current tile are rendered using a standard Z-buffer, the HZ is computed. Let $h_0(i,j)$ be the Z-buffer array for a tile of size 128x128. The hierarchical Z-buffer at level k is computed from

$$h_k(i,j) = \underset{a=0,b=0}{\overset{a=D-1,b=D-1}{Max}} \left(h_{k-1}(Di+a, Dj+b) \right)$$

where D is the degree of the HZ. Each value at level k is the maximum of the corresponding D by D region of level *k-1* as shown in figure 2. The root of this hierarchy is a single pixel containing the maximum z or furthest point of the scene in the tile.

A function HZVisQuery(Sbox) is needed that can test a triangle against the HZ to see if it is completely hidden. The Sbox of a triangle is the x,y screen coordinate bounding box of the portion of the triangle inside the tile (Ebox) together with its minimum z value EZ.

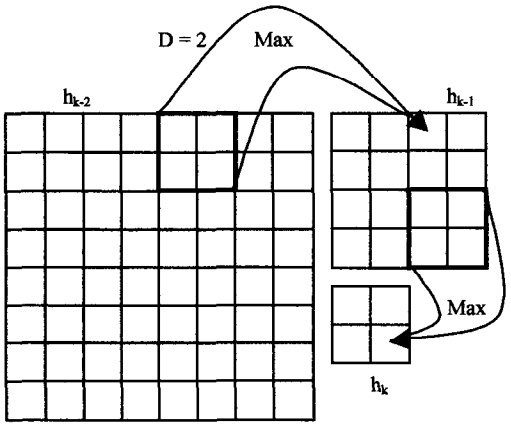


Figure 2. Hierarchical Z-buffer construction. Each level k has values that are the maximum of the values in a D by D region of the previous level.

HZVisQuery first computes L, the level in HZ to test the Sbox for visibility. Secondly, HZVisQuery tests the box against level L. For a simple architecture it is desirable to only compute one HZ level and to test every polygon's visibility using only this level. The level L to construct and use is determined by the desired cost of visibility testing using the following

$L = log_D(sqrt(1 / Cppt))$

Where *Cppt* is the desired cost per pixel of HZ testing expressed as a fraction of the total cost of rendering a pixel in a standard Z-buffer architecture. For example, if we wish to spend 1/16 as much time per pixel on HZ testing as on rendering then *Cppt* is 1/16. A polygon covers n pixels at the lowest level, n/D^2 at the next and so on. A visibility test at level L need only test the number of HZ elements that the polygon covers at that level. Testing a polygon at level L fixes the HZ per pixel test cost at a constant. If the pyramid is of degree $D=2$, choosing $Cppt=1/16$, gives $L=2$; and choosing $Cppt=1/256$ gives $L=4$. After the selection of L, the triangle is tested against level L of HZ as follows:

```
for each pixel in level L of the HZ covered by Sbox {
  if ( PixelZ > MinZ )
      return true;
}
return false;
```

The technique of selecting a level L for testing is well suited to hardware pipelining because it constrains the triangle occlusion test cost to a prescribed per-pixel fraction of pixel Z-buffering cost.

2.4 When to Build the HZ

Progressive hierarchical visibility (PHV) updates the HZ with each pixel rendered. This has a significant impact on the rendering pipeline. Much of the visibility culling benefit is obtained with a simple method that builds the HZ once, at some selected point, and then uses it for subsequent visibility testing. AHV constructs the HZ from the Z-buffer once, based on an adaptive, pixel coverage value that exploits frame to frame coherence.

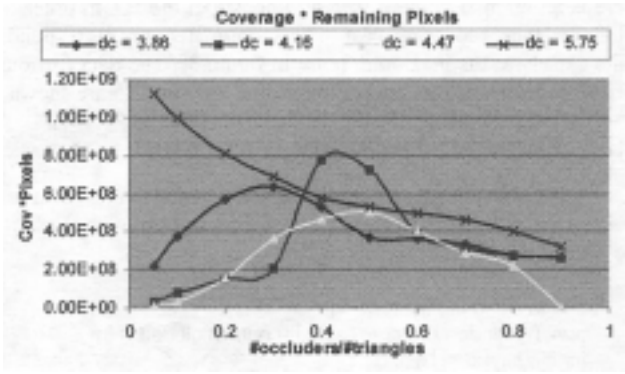


Figure 3. HZ pixel coverage weighted by the number of remaining pixels versus the percentage of polygons used as occluders. Four tiles with different depth complexities (dc) are plotted.

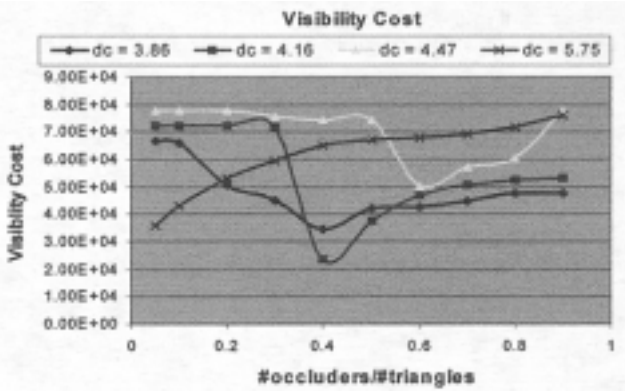


Figure 4. As more occluders are rendered before constructing the HZ, the tile's total per-pixel cost is plotted for the same four tiles as figure 3. Visibility cost represents the total visibility resolution cost of the tile when HZ is constructed and used after the given fraction of triangles are rendered.

When objects are sorted in depth order, the rate of occlusion (the fraction of triangles being occluded) is roughly proportional to the accumulated coverage (the fraction of the tile's pixels that have been hit). When accumulated coverage is one, updates to the HZ increase total cost, without improving occlusion culling.

Coverage is easily computed during rendering by counting the number of fresh z writes (a write into a z that was cleared). The HZ is constructed when the fraction of polygons rendered (FOPR) for the current frame time reaches the ideal fraction computed for last frame. More precisely, during the last frame we

compute and save the FOPR, F_{max}, for which the following expression was a maximum

(Pixels_remaining)(Coverage – HZ_Test_Cost)*

where pixels remaining is the sum of all polygon areas still to process. If the coverage is quite high and many pixels remain, then the occlusion savings will likely be high. If there is good temporal coherence, then the ideal fraction computed during the last frame will also be a good estimation for the ideal place to construct the HZ this frame. So when the current frame FOPR reaches F_{max}, the HZ is constructed. This will be fully explained in section 3.

Figures 3 & 4 give further motivation to this choice of coverage metric. Figure 3 shows how the tile coverage times the remaining pixels reaches a clear maximum as an increasing fraction of the polygons are rendered. Figure 4 data indicates that there is an optimal point at which to construct the HZ in order to minimize total rendering cost. This optimal point corresponds quite closely to the maximum point in figure 3. The data for four tiles with depth complexities ranging from 3.86 to 5.75 are shown.

2.5 Adaptive Visibility Algorithm

Once the triangles for a tile are z sorted into bins as described above, the polygons are rendered starting with those in the nearest bin. The complete algorithm is given here

```
for each tile in the screen    {
  numTriRendered = numTested = numOccluded = 0;
  for tile bins b = b0 to bn {
    for each triangle tri in b {
        Vis = true;
        if ( HZExist) {
          numTested++;
          Vis = HZVisQuery( tri.Sbox);
          if ( !Vis)
            numOccluded++;
        }
        if (Vis) {
          render triangle tri;
          UpdateIdealHZPlace();
          numTriRendered++;
        }
    }
    if ( !HZExist) {
      if ( ReachedIdealHZPlace()) {
          if (tile. IsHZWorthWhile( ))
            ConstructHZ( );
      }
    }
  }
  setup buckets in tile using tile.MeanZ and tile.STDZ;
  if ( numTested > 0)
        HZOcclusionRate = numOccluded / numTested;
}
```

UpdateIdealHZPlace() is used to locate the best place to construct the HZ. Using the optimal coverage metric from the last frame obviates a deadly stall of the pipeline by exploiting frame to frame temporal coherence. **ReachedIdealHZPlace()** is used to check if we have reached the ideal place to construct HZ as determined during the previous frame. The **IsHZWorthWhile()** function returns the value of the following inequality:

(HZBuildCost + numTriangleTests * HZTestCost) < (numTriangleTests*HZOcclusionRate*ScanConvertCost)

using the following definitions.

numTriangleTests: Number of the triangles left to be rendered after the HZ has been constructed. This is the number of triangles in the tile minus the number of triangles (occluders) already rendered when HZ construction started, and minus the number of triangles rendered while HZ was being constructed.

ScanConvertCost can be set to the **averageTriangleArea** with a normalized Z-buffering cost of 1 per pixel.

HZTestCost is proportional to the **averageTriangleArea**, that is, **HZTestCost = Cppt * averageTriangleArea.**

The inequality requires that the cost of building the HZ plus the expected cost of testing polygons against it, is less than the cost of rendering the expected number of occluded polygons. So the HZ is only built if it is likely to be useful and this is determined by a simple computation.

3 Cost Benefit Analysis

A cost model is needed in order to compare the cost of AHV, PHV and Z-buffering in a more analytic manner. The total pixel rendering cost of a deferred shading pipeline is considered. The total pixel rendering cost *Ctp* for all three algorithms is the sum of visibility (resolution) cost *Cvis* plus the cost of rendering visible pixels *Crender* or:

$$Ctp = Cvis + Crender$$

With the same scene and same depth sorting strategy, AHV, PHV and Z-buffering yields the same number of visible pixels because AHV and PHV do not reduce or increase the actual number of visible pixels, they are only used to reduce the number of Z-buffer tests. So the difference between the pixel cost of three algorithms lies entirely in visibility cost.

For a deferred shading ordinary Z-buffer algorithm (ZB) the cost *Cvis* is the same as *Czb*. If the per pixel Z-buffer cost is normalized to *1*, the *Czb* is just the total number of pixels *Tp* in all the triangles within the tile or

$$Czb = Tp$$

3.1 Cost Model of PHV

In PHV, whenever a pixel's z value is modified, the change to the Z-buffer is propagated to higher levels of the HZ in an update operation. For our cost comparisons, only one level of the HZ is updated so that the HZ being used is the same for PHV as for AHV. As pixel p is being rendered, function $V(p)$ is used to denote the probability that the p^{th} pixel is visible. For pixel one $V(1)=1$ and for the pixel p_0 when the last visible pixel has been drawn $V(p_0)=0$. If the pixels are rendered in perfect near to far order, then $V(p)$ will be a monotonically decreasing function. We refer to the area under the $V(p)$ curve integrated between 0 and p_0 as V_{sum}, the sum of all visible pixels.

The visibility cost model of PHV can be represented as:

$$Cph = \int_0^{Tp} (V(p) * [1 + Cppu(p)] + Cppt(p))dp$$

where *Cppt* is the HZ query cost for pixel *p*, and *Cppu* is the HZ update cost for pixel *p*. For a single level HZ of degree *D*, every *D*D* pixels causes one HZ test , so the *Cppt* can be normalized to $1/D^2$. On the other hand, *Cppu* varies depending on the attributes of the triangle that *p* belongs to. In ordinary Z-buffer, the number of Z values accessed while scan converting a triangle is the area of the triangle. In PHV the number of z values accessed while scan converting a triangle *T* is equal to the number of *DxD* tiles that *T*

overlaps, multiplied by D^2, because every one of these *DxD* tiles can cause an update entry in the HZ. The number of *DxD* tiles, *NT*, that a triangle overlaps can be determined by the formula:

$$NT = \left\lfloor \frac{D^2 + A_t}{D^2} \right\rfloor + \left\lfloor \frac{3\sqrt{2A_t}}{D\sqrt{2}} \right\rfloor$$

Where A_t is the area of the triangle. The formula means that the number of *DxD* tiles a triangle covers is the number of *DxD* tiles inside the triangle plus the number of *DxD* tiles intersected by the triangle edges. It is a good estimation of the overlap factor for fat triangles. For skinny triangles, *NT* can be worse. The per pixel HZ update cost then would be:

$$C_{ppu} = \frac{(NT * D^2)}{A_t} - 1$$

It is simple to show that this value ranges from D^2 to 0 as triangle size varies from 1 to infinity. Since triangle size is trending smaller, the HZ update cost can be significant in PHV.

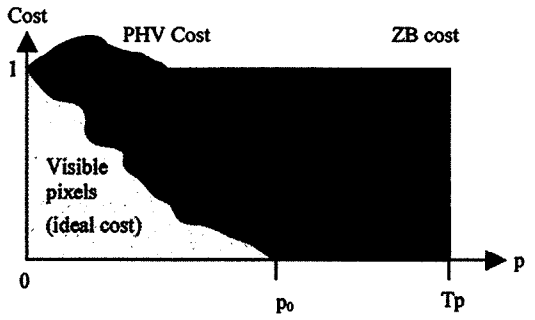


Figure 5. Model based expected PHV cost as a function of sequentially rendered pixels, is compared to expected ordinary Z-buffer method cost. The probability of pixel visibility is also plotted. Total PHV cost is the gray area under the PHV cost curve. p_0 identifies the pixel after which all pixels are hidden.

Figure 5 shows how the PHV cost is accrued as pixels are rendered, compared to the ZB cost and to the accumulation of visible pixels. It gives a good intuitive comparison of the cost of PHV vs. ZB as *Tp* changes. For $Tp<p_0$, which happens when depth complexity is low, PHV can be more expensive than ZB since the cost of updating the HZ can not be offset by the savings of not rendering hidden pixels. As $Tp-p_0$ (or depth complexity) increases, the benefit of PHV over ZB increases.

3.2 Cost Model of AHV

The visibility determination cost model of AHV, using the above, idealized analysis, is given by

$$Cah = p_{HZ} + (Tp - p_{HZ})(Cppt + V(p_{HZ})) + HZBuildCost$$

where p_{HZ} is the number of pixels Z-buffer tested before the HZ build, $V(p_{HZ})$ is the HZ test pass rate (ie. the fraction of pixels visible after construction of the HZ), and *HZBuildCost* is the cost of building the HZ.

Cah, the visibility cost for AHV is the sum of Z-buffer test cost of all the objects up until the HZ is built, plus the cost of HZ testing the remaining objects and cost of Z-buffer testing all pixels passing HZ tests, and plus the cost of building the HZ.

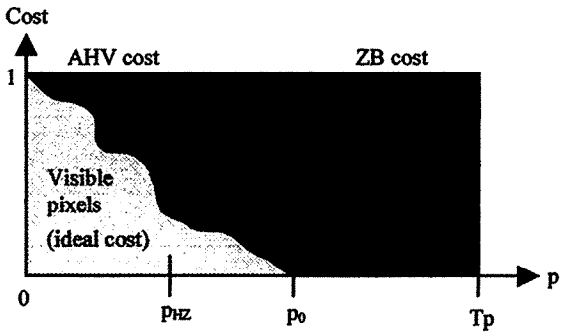


Figure 6. Model based expected AHV cost as a function of sequentially rendered pixels, is compared to expected ordinary Z-buffer method cost. The probability of pixel visibility is also plotted. The total AHV cost is the gray area under the AHV cost curve plus a fixed construction cost. The dark area represents the saving of AHV over ZB. AHV is only constructed if size of the dark area is greater than the construction cost. p_0 identifies the pixel after which all pixels are hidden. p_{HZ} identifies the pixel after which HZ is constructed.

Figure 6 shows how the AHV cost accrues as pixels are rendered compared to traditional ZB cost. Like figure 5, figure 6 gives a good intuitive comparison of the cost of AHV vs. ZB. Unlike PHV, AHV is never worse than ZB. For scenes with very low depth complexity, AHV does not construct or use an occlusion map, so the cost is the same as ZB. For scenes with medium to high depth complexity, it is obvious from figure 6 that the cost savings of AHV over ZB depends on selecting the right place to construct HZ.

The optimal HZ construction point happens when *Cah* is at a minimum or when the savings of *Cah* over *Czb* is greatest since *Czb* is fixed at *Tp*. We use *Csav* to denote the saving of *Cah* over *Czb:*

$$\begin{aligned} Csav &= Czb - Cah \\ &= Tp - p_{HZ} - (Tp - p_{HZ})(Cppt + V(p_{HZ})) - HZBuildCost \\ &= (Tp - p_{HZ})(1 - V(p_{HZ}) - Cppt) - HZBuildCost \end{aligned}$$

Since *HZBuildCost* is constant, maximizing *Csav* is equivalent to maximizing $(Tp - p_{HZ})(1 - V(p_{HZ}) - Cppt))$. In this formula, *Tp*, p_{HZ} and *Cppt* are well defined and measurable. $V(p_{HZ})$ is hard to measure locally because it is defined as the probability that pixel p_{HZ} is visible if pixels $0...p_{HZ}-1$ are rendered. To measure *V(p)* accurately for some *p*, we need to summarize the probability over all pixels after *p* and compute the expected value. It is impossible to do this within a frame. An alternative approach would be to estimate *V(p)* by computing it for the next N pixels. Unfortunately, the next N pixels can give an incorrect estimate of the distribution of pixels to follow and lead to a local minimum that gives poor global performance. Instead, the following expression involving the coverage function *Cov* is maximized:

$$(Tp - p_{HZ})(Cov(p) - Cppt) \qquad (1)$$

where *Cov(p)* is defined as the ratio of the number of non empty pixels in the tile after pixels 0..*p* are rendered over the total number of non empty pixels in the tile after all pixels are rendered. *Cov(p)* is a good approximation to *1-V(p)*, since it is clearly the case that the bigger the coverage at *p*, the bigger the probability that pixels after *p* will be occluded. In fact, as triangles approach the size of a single pixel, *V(p) = 1-Cov(p).*

As all the terms in (1) are easily measured while rendering the triangles in frame n, we can find the fraction of polygons rendered (FOPR) for which (1) is maximum in frame n. This ideal FOPR or F_{max} for frame n is saved and when frame $n+1$ is processed, HZ is built when the FOPR reaches $K*F_{max}$, and it is deemed to be worthwhile.

3.3 AHV vs. PHV

Figure 5 and 6 together gives us some insight into how AHV compares with PHV. As Tp-p_0 goes to infinity (high depth complexity), p_{HZ} approaches p_0 and the ratio Cph/Cah approaches one. As Tp approaches 0 (low depth complexity), AHV will be better than PHV because the gain of removing a small number of hidden pixels can not justify the update cost of PHV.

For tiles with medium depth complexity, the performance of AHV vs. PHV depends on the distribution of triangles. The weakness of PHV lies in the update overhead, while the weakness of AHV lies in waiting for the "best" place to construct and use the occlusion map.

There are two extreme cases of triangle distribution where there is a clear winner between AHV and PHV.

PHV beats AHV when a few large triangles come first and fill up an area of the tile while leaving other areas uncovered (so AHV will not build its HZ). If many small triangles hidden behind these come along, followed by large triangles to fill other parts of the tile, PHV will win, because the initial large occluders incur little update overhead to accumulate into an effective occlusion map. AHV has an advantage if the visibility of the pixels drops slowly at first then sharply. This can happen when many small triangles come first and gradually merge to build coverage. PHV suffers from testing triangles that are not hidden and from incrementally updating an ineffective HZ.

4 Optimization

4.1 Early Termination

One simple optimization is to always maintain the maximum Z-buffer value written so far in a tile. When the tile becomes fully covered, any subsequent Z-bucket whose minimum z value exceeds the maximum Z-buffer value can be ignored because all of its triangles will be occluded.

4.2 Chunk Rejection

The occlusion rate or occlusion effectiveness of both AHV and PHV drops as the average triangle size of the scene increases. This is due to the fact that AHV and PHV are both rejecting at the triangle level. The larger the triangle, the wider the depth range across the triangle, and the less likely the bounding box test will pass. A simple optimization can be done to increase occlusion rate for both AHV and PHV for scan converters that generate pixels in 4x4 chunks [19]. For such scan converters, each 4x4 chunk yields one HZ test (when HZ has degree 4). The pixel processing within a chunk is aborted when the chunk fails the HZ test. This optimization significantly increases occlusion rates for both AHV and PHV in scenes with large triangles. It does not change the cost relationship between AHV and PHV because the increase of query granularity is orthogonal to the HZ construction difference between AHV and PHV.

5 Experimentation and Results

We have implemented PHV and AHV in a functional simulator and instrumented the measurement of the following quantities. **Potential Depth Complexity** was measured as the total number of Z-buffer tests. **HZ Test Cost** was measured as the total number of HZ tests. And finally, **Visibility Cost** was measured as the cost of determining all visible pixels inside all potentially visible triangles.

For the Z-buffer method, the visibility cost is the same as the potential depth complexity. For AHV, visibility cost is the sum of potential depth complexity, HZ test cost, and HZ construction cost. For PHV, visibility cost is the sum of potential depth complexity, HZ test cost, and the Z-buffer reads for updating HZ.

All the simulation results we present are gathered using the same configuration. Tile size is 128x128. The degree D of HZ is 4, so the HZ test cost per pixel is 1/16. Both PHV and AHV use only a single level of HZ. The number of z buckets for each tile is 10. Here is the area and depth distribution of the four test scenes we used:

	Avg Depth	Min Depth	Max Depth
Canyon	3.1	0.8	7.6
Island	2.2	1.0	4.3
Babylon 1	3.6	1.1	7.6
Babylon 2	3.4	1.8	6

Figure 7. Per pixel depth complexity for scene tiles

	Avg Area	Min Area	Max Area
Canyon	31	14	954
Island	1389	43	5418
Babylon 1	1163	76	3323
Babylon 2	535	24	2112

Figure 8. Average triangle area for scene tiles.

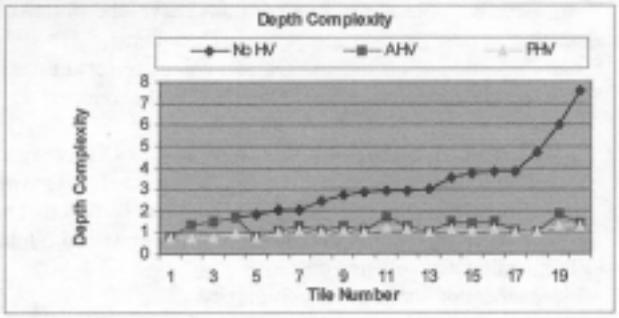
The following tables summarize the performance of PHV and AHV for the four test scenes:

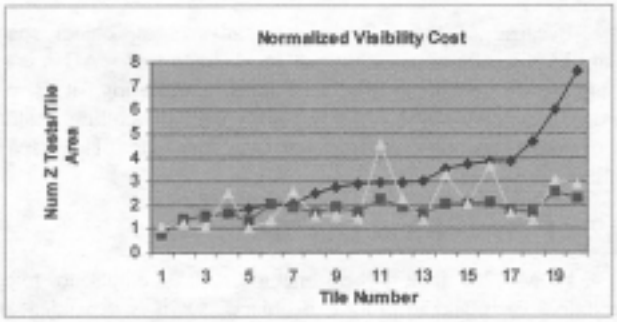
	Avg VisCost	Min VisCost	Max VisCost
Canyon	2.11	1.03	4.56
Island	1.47	1.0	2.96
Babylon 1	1.44	.76	2.56
Babylon 2	2.06	.81	3.3

Figure 9. PHV method: Per pixel visibility cost for scene tiles

	Avg VisCost	Min VisCost	Max VisCost
Canyon	1.81	.78	2.56
Island	1.48	1.0	2.36
Babylon 1	1.75	1.0	2.56
Babylon 2	2.08	1.20	2.75

Figure 10. AHV method: Per pixel visibility cost for scene tiles





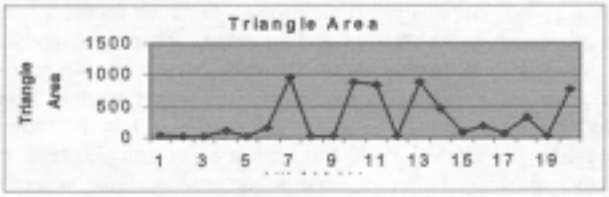
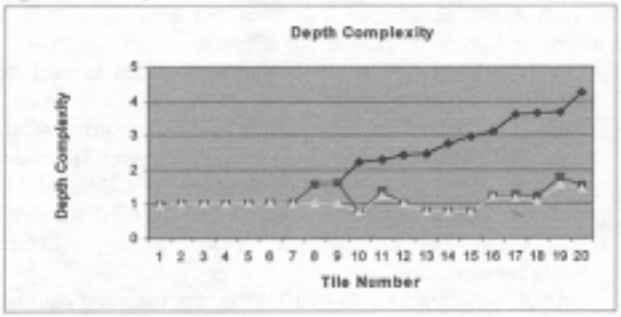
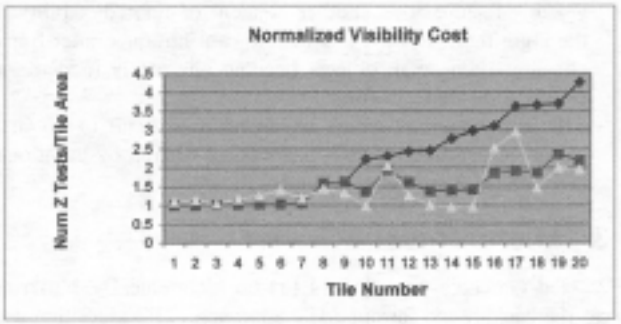


Figure 11. Canyon Scene.





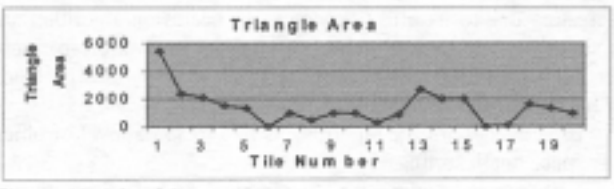
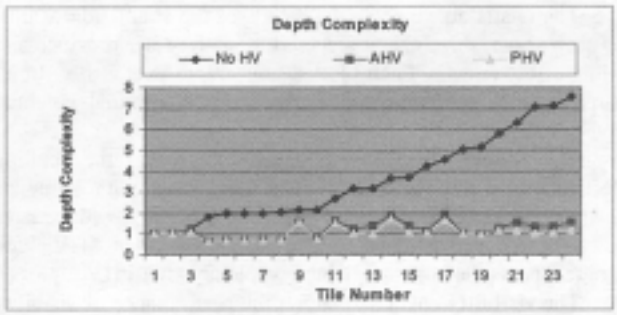
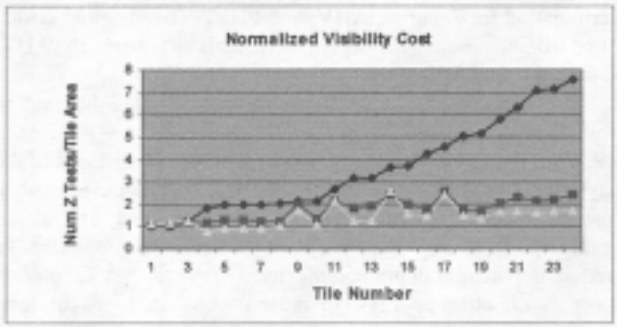


Figure 12. Island Scene.





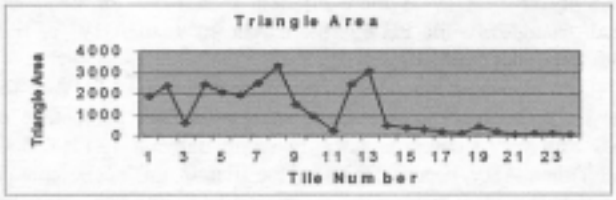
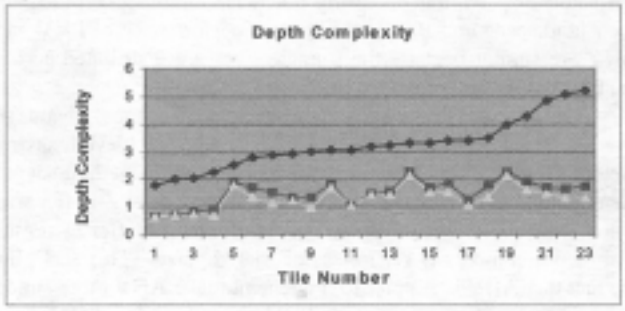
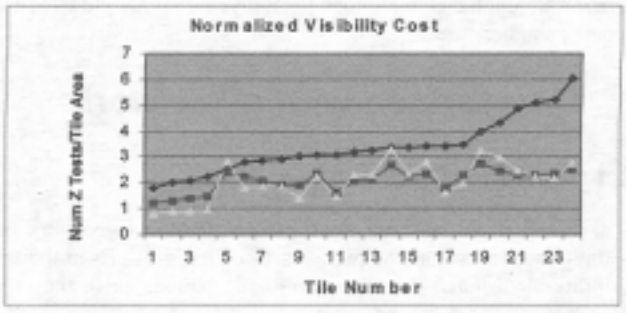


Figure 13. Babylon 1.





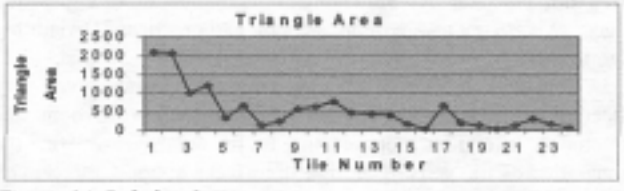


Figure 14. Babylon 2.

The charts above show the details of the simulation results. For each scene, three charts are used to analyze performance and culling effectiveness. Each plotted point represents a tile. In all charts the tiles are numbered in the order of original potential depth complexity.

The **depth complexity** chart compares the culling effectiveness of AHV and PHV. The data plotted with diamonds shows the potential depth complexity of Z-buffering, with squares shows the reduced potential depth complexity after AHV, with triangles shows the potential depth complexity after PHV.

The **visibility cost** chart shows the performance of visibility determination for Z-buffer, AHV, and PHV. The visibility cost is plotted using diamonds for the Z-buffer method, squares for AHV, and triangles for PHV.

The **triangle area** chart shows the triangle area distribution across the tiles.

The simulation results agree well with the predictions of the analytic cost models. The tiles in the test scenes cover a wide range of triangle distributions in terms of area and depth complexity. In most of the tiles, both methods are quite effective at reducing potential depth complexity. AHV, though simpler, is almost as effective as PHV in most cases. In terms of total visibility cost, AHV is more efficient than PHV for tiles with small triangles, while PHV is more efficient than AHV for tiles with large triangles.

At the scene level, AHV performs better than PHV in the canyon scene because this scene contains a high percentage of tiles with very small triangles. In scene Babylon 1, PHV does better than AHV because most of the tiles in this scene contain large triangles which quickly accumulate into an effective occlusion map without incurring much HZ update overhead. In the island scene and Babylon 2, overall performance of PHV and AHV are similar because the triangle sizes are distributed across the tiles more evenly than the previous two cases.

Overall, AHV shows a significant performance advantage over Z-buffer. For the tiles measured, AHV achieved saving factors ranging from 1 to 7. Even for tiles with medium depth complexity, AHV"s performance improvement over Z-buffer was more than 50%. The advantage of AHV over Z-buffer increases with depth complexity regardless of triangle size. This fact plus the fact that AHV is simpler to implement make AHV more suited to hardware implementation than PHV given that triangle sizes are trending smaller.

6 Comparison with other work

6.1 Object Space Algorithms

Much work has been done in computational geometry on visibility determination and hidden surface removal. Even though visibility algorithms with nice asymptotic bounds exist, they are often not practical for scenes with thousands of triangles and edges. We are not aware of any general and practical 3D visibility algorithm.

The work by Teller [23] on cells and portals gives very effective methods in the domain of 3D scenes with room like structures. It requires preprocessing on the database to define the boundaries of the cells. The algorithm lacks support for general 3D scenes, like outdoor scenes with no clear subdivision of space into cells and portals, or for scenes with many moving objects where the amount of preprocessing that can be done is limited.

In general, object space algorithms have the following fundamental advantages over image space algorithms:

1. Early rejection of occludees. Objects can be rejected prior to perspective transformation. Objects can be rejected as a group.
2. Independent of viewpoint and direction. Object space algorithms and data structures are constructed to answer visibility queries for any viewpoint and view direction. This makes these algorithms potentially useful for vision, global illumination and collision detection.
3. Independent of image space complexity.

Because of these fundamental advantages, object space visibility algorithms will continue to be important. AHV is an image space technique like traditional Z-buffering; it is not designed to replace efficient object space visibility culling. Rather it performs visibility culling that augments the object space culling.

6.2 Hierarchical Occlusion Map

HOM [26] is a hybrid method. It attempts to reject occludees early and in groups of objects, yet it is dependent on image space complexity and viewpoint. For each frame, a small set of occluders are selected and rendered. Then a hierarchical occlusion map is constructed to efficiently answer overlap queries for projected bounding boxes. HOM does not contain depth information, so when the projected bounding box is entirely covered by the HOM, a depth test against either a single plane or a depth estimation buffer is done to determine if the object is hidden. This is a software method that can be effective for scenes with high depth complexity, but does not lend itself well to hardware implementation due to the following issues:

1. The effectiveness of HOM is intrinsically tied to occluder selection, yet the algorithm does not provide a systematic method to select good occluders. This means that the algorithm's effectiveness is hard to predict given a 3D model.
2. Because HOM has no depth information, an additional data structure called depth estimation buffer is used for depth comparison. This additional complexity is not well suited for hardware pipelining.
3. One of the key attributes of HOM is "approximate culling", yet this feature adds another element of unpredictability to the algorithm. In fact, this feature can introduce inter-frame aliasing when small objects become alternately hidden and visible from frame to frame.
4. Efficient querying against an HOM is difficult to do in a hardware pipeline. This is a drawback shared by traditional hierarchical Z-buffering as well.

6.3 Hierarchical Z Buffer

Ned Greene's work [13,14,15] on hierarchical Z-buffering forms the background for our AHV approach. This algorithm was interesting due to its effectiveness as an occlusion algorithm and its simplicity. It showed great potential for hardware pipelining. Yet full screen hierarchical Z-buffer faces some serious problems for hardware implementation:

1. Effectiveness of the algorithm is tied to some level of object space depth sorting.
2. Full screen progressive hierarchical Z-buffering impacts hardware complexity.

3. Query against hardware HZ is difficult to pipeline.

A tiled architecture offers a unique opportunity to revisit this method because some of the above drawbacks go away or become less of a problem:

1. While binning a triangle into (x,y) tiles, we can bin the triangle into z buckets almost for free either in the CPU or the graphics pipeline. Bucketing in z gives us a good depth sort and a good method for occluder selection.
2. Depth complexity is not evenly distributed over the whole screen. In a tiled architecture, the HZ can be adaptively invoked when needed.
3. AHV in a tiled architecture requires little change to the pipeline and incurs little cost.

7 Conclusion

In this paper we have presented an adaptive hierarchical visibility algorithm in a tiled architecture that can significantly reduce the computation cost for scan converting triangles. The algorithm has the following unique properties:

Adaptation: Tiled architectures sort in image space to facilitate cache coherence in the frame-buffer and Z-buffers. This increases geometry bandwidth while reducing frame-buffer and Z-buffer bandwidth. AHV takes advantage of this sorting stage even further to sort the triangles in z using a bucket sort that does not increase geometry bandwidth. For each tile it gathers statistics regarding the distribution of depth and triangle area. These statistics are used later to determine whether it is worthwhile to construct the occlusion map, and if so, to decide when it should be built to minimize total rendering cost.

Temporal Coherence: AHV takes advantage of frame to frame coherence in the following ways:

1. For each tile, the depth distribution of frame n is used to ensure good distribution of triangles in the z buckets in frame n+1.
2. For each tile, statistics from frame n are used to decide when to build the HZ for frame n+1. The maximum of coverage times remaining pixels is found in frame n. The corresponding percentage of triangles rendered is saved and used to decide when to build the HZ in frame n+1. Since temporal coherence increases as frame rate increases, it is likely to be important in the future.

7.1 Strength

AHV integrated into a tiled architecture is simpler for hardware implementation than other hierarchical visibility methods. It performs comparably to a comprehensive hierarchical visibility method such as PHV, and better in small polygon cases. Performance gains from such culling can approach a factor of 16, the difference between a visibility test and scan conversion.

AHV reduces both the computation and bandwidth costs over a traditional Z-buffer method. Compared to a deferred shading pipeline, AHV can reduce the computational cost with little additional bandwidth cost.

AHV has advantages over full screen progressive hierarchical Z-buffering methods in that 1) it adaptively selects a good visibility solution for each tile based on statistics gathered during tile processing, and 2) it is easier to implement in a graphics hardware pipeline.

AHV improves on full screen mode, hierarchical occlusion map methods by offering 1) cheaper and more effective occluder selection through 3D binning, 2) fast pixel accurate occlusion instead of approximate occlusion, and 3) simpler and easier integration in a hardware pipeline.

7.2 Weakness

AHV works well as an instrument to accelerate hidden surface removal in a tiled architecture, but it is not a general visibility determination mechanism. The effectiveness of occlusion culling depends on how early in the pipeline objects are rejected and at what granularity they are rejected.

The impact of AHV is constrained by rejecting triangles post transformation. Since scenes with high pixel depth complexity often have high geometry complexity, it is highly possible for the computation and bandwidth costs of transforming and lighting of such scenes to exceed the geometry processing capabilities of the system. In which case AHV's reduction of the rendering cost alone can not change the total throughput.

The effectiveness of AHV depends on choosing a good place to construct the HZ. The optimal point is when (OcclusionProbability(p) * PixelsRemaining) is a maximum. Since there is no way of determining "OcclusionProbability" efficiently for each triangle, the "Coverage" value is used to approximate it. This approximation has the desirable property that as triangle size approaches subpixel, Coverage = Occusion Probability. However, there are still triangle distributions where this approximation breaks down. This is similar to caching where the ideal page to replace is the one that is least likely to be used in the future. Because we can not predict the future, we use an approximation like the page that is least recently used. This works well most of the time but not all of the time.

8 Future work

The weakness of the algorithm should be addressed in the future. First, we would like to do a comprehensive study of scenes to see how well the use of coverage as an approximation of occlusion rate works. Second, we would like to study some alternative binning strategies where triangles are sorted by a weighted sum of area and depth to see if that would enable earlier construction of HZ and better occlusion rate. Third but certainly not the last, we would like to investigate the integration of AHV into a scene manager where the HZ can be used to do occlusion queries for groups of objects.

Acknowledgments

Thanks to Scott Nelson, Jim Hurley, and Jonathan Sweedler for helpful discussions of graphics pipeline issues, and to Nola Donato and Jeff Ma for help with the polygonal test databases. We thank Leonidas Guibas and Milton Chen for insight into tiling and hierarchical graphical data structures. We are grateful to Bob Liang and Richard Wirt for supporting this work.

References

[1] E. Catmull, *A subdivision algorithm for computer display of curved surfaces*. PhD thesis, University of Utah, 1974.

[2] J. C. Chauvin (Sogitec). An advanced Z-buffer technology. *IMAGE VII*, pages 76-85, 1994.

[3] M. Chen, G. Stoll, H. Igehy, K. Proudfoot and P. Hanrahan, Simple Models of the Impact of Overlap in Bucket Rendering, *1998 Siggraph/Eurographics Workshop on Graphics Hardware*, pp. 105-122..

[4] J. H. Clark, Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547-554, 1976.

[5] S. Coorg and S. Teller. A spatially and temporally coherent object space visibility algorithm. Technical Report TM 546, Laboratory for Computer Science, Massachusetts Institute of Technology, 1996.

[6] M. Cox, *Algorithms for Parallel Rendering*, Ph.D. thesis, Princeton University, May 1995.

[7] M. Cox, and N. Bhandari, Architectural Implications of Hardware-Accelerated Bucket Rendering on the PC, 1997 *Siggraph/Eurographics Workshop on Graphics Hardware*, pp. 25-34.

[8] D. Ellsworth, *Polygon Rendering for Interactive Visualizations on Multicomputers*, Ph.D. thesis, University of North Carolina at Chapel Hill, December 1996.

[9] J. Foley, A. Van Dam, J. Hughes, and S. Feiner, *Computer Graphics: Principles and Practice*. Addison Wesley, Reading, Mass., 1990.

[10] H. Fuchs, Z. Kedem, and B. Naylor. On visible surface generation by a priori tree structures. *Computer Graphics (Proc. Siggraph)*, Vol. 14, No. 3, 1980.

[11] H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Teggs, L. Israel, Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories, *Computer Graphics (Proc. Siggraph)*, Vol. 23, No. 3, July 1989, pp 79-88.

[12] C. Georges, Obscuration culling on parallel graphics architectures, Technical Report TR95-017, Department of Computer Science, University of North Carolina, Chapel Hill, 1995.

[13] N. Greene, and M. Kass, Error-bounded antialiased rendering of complex environments. *Computer Graphics (Proc. Siggraph)*, Vol. 28, No. 2, 1994.

[14] N. Greene, M. Kass, and G. Miller. Hierarchical Z-buffer visibility, *Computer Graphics (Proc. Siggraph)* Vol. 27, 1993.

[15] N. Greene, Hierarchical polygon tiling with coverage masks, *Computer Graphics (Proc. Siggraph)* 1996.

[16] S. Herrod, *Using Complete Machine Simulation to Understand Computer System Behavior*, Ph.D. thesis, Stanford University, February 1998.

[17] T. Hudson, D. Manocha, J. Cohen, M. Lin, K. Hoff and H. Zhang. Accelerated occlusion culling using shadow frusta. Technical Report TR96-052, Department of Computer Science, University of North Carolina, 1996.

[18] D. Luebke and C. Georges. Portals and mirrors: Simple, fast evaluation of potentially visible sets. In ACM Interactive 3D Graphics Conference, Monterey, CA, 1995.

[19] J. McCormack, R. McNamara, *et al.*, Neon: a single-chip workstation graphics accelerator, 1998 *Siggraph/Eurographics Workshop on Graphics Hardware*, pp. 123-132.

[20] S. Molnar, *Image-Composition Architectures for Real-Time Image Generation*, Ph.D. thesis, University of North Carolina at Chapel Hill, October 1991.

[21] S. Molnar, M. Cox, D. Ellsworth, H. Fuchs, A Sorting Classification of Parallel Rendering, *IEEE Computer Graphics and Applications*, Vol. 14, No. 4 July 1994.

[22] S. Molnar, J. Eyles, and J. Poulton, PixelFlow: High-Speed Rendering Using Image Composition, *Computer Graphics (Proc. Siggraph)*, Vol. 26, No. 2, July 1992.

[23] S. Teller and C.H. Sequin. Visibility preprocessing for interactive walkthroughs. Computer Graphics (Proc. Siggraph), 1991.

[24] J. Warnock, A hidden-surface algorithm for computer generated halftone pictures, Technical Report TR 4-15, NTIS AD-753 671, Department of Computer Science, University of Utah, 1969.

[25] L. Williams, Pyramidal parametrics, Computer Graphics, (Proc. Siggraph), 1983.

[26] H. Zhang, D. Manocha, T. Hudson, K. E. Hoff III, Visibility Culling using Hierarchical Occlusion Maps, Siggraph '97.

Figure 15. Canyon Scene, courtesy of 3D WinBench.

Figure 17. Babylon 1, courtesy of Virtual Alchemy Studios.

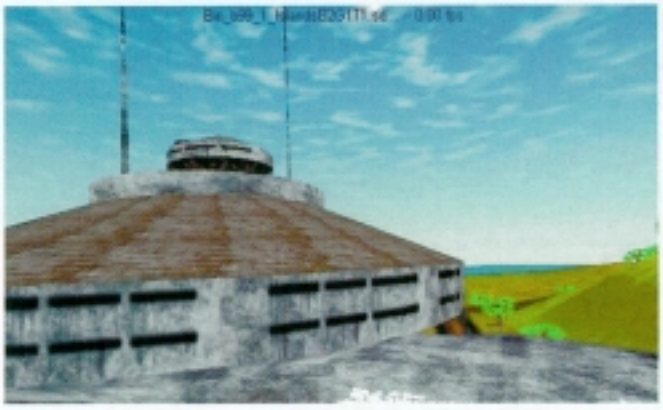


Figure 16. Island Scene, courtesy of 3D WinBench.

Figure 18. Babylon 2, courtesy of Virtual Alchemy Studio.