Fast and Simple Occlusion Culling using Hardware-Based Depth Queries

K. Hillesland B. Salomon A. Lastra D. Manocha
University of North Carolina at Chapel Hill
{khillesl,salomon,lastra,dm}@cs.unc.edu
http://www.cs.unc.edu/∼walk/FSOC

Abstract: *We present a conservative occlusion culling algorithm for large environments. As part of a preprocess, we decompose the scene using a spatial subdivision and render the primitives at runtime in a front-to-back order. Our algorithm uses hardware accelerated occlusion queries to test the visibility of more distant volumes of space in a progressive manner. The resulting algorithm is simple to implement and makes use of hardware features including the occlusion queries and vertex shaders for fast performance. We have implemented it on a PC with an NVidia GeForce3 card and are able to render a powerplant model composed of 12.5 million triangles at* $10 - 20$ *frames a second. We are able to achieve a speedup of from four to ten times in frame rate with no loss in image quality.*

1 Introduction

In spite of the rapid progress in the performance levels of graphics hardware, it is still not possible to render very large models at interactive rates. The models used in common applications including CAD, virtual environments, visualization and simulations are getting more complex. At the same time, the bandwidth to the graphics cards is not increasing as fast as computational power. Therefore, to achieve peak rates requires that rendering be done in retained mode. Since models must be stored on the memory of the graphics card, there is a hard limit on the size of models that can be rendered at full rates. Given that the rendering of very large models is bandwidth limited, our first priority is to ensure that we minimize the number of primitives sent to the graphics card.

In massive models many of the underlying primitives do not contribute to the final image. We can classify these primitives into three categories.

1. Those outside the view frustum.
2. Those that project to less than a pixel in screen space and are not rendered due to the sampled nature of the frame buffer.
3. Those fully occluded by other primitives (including backfacing primitives).

The goal of view frustum culling is to quickly reject primitives in category one. Level-of-detail (LOD) and image-based impostor techniques are commonly used to reduce the number of type two primitives, while occlusion culling aims to eliminate primitives of type three. View frustum culling is used routinely, and the use of automatically-generated LODs or impostors is becoming more common. However, no simple and general solutions are known for occlusion culling. Current occlusion culling algorithms fall into two main categories. Some are specific to certain types of models, such as architectural or urban environments and not applicable to general environments. The more general approaches either require very specialized hardware, extensive pre-processing of visibility, multiple passes using multiple graphics pipelines, or the presence of large, easily identifiable occluders in the scene.

Main Results: We present a novel occlusion culling method that is simple, conservative, general, and progressive in nature. It begins by precomputing a spatial subdivision of the model. Based on the subdivision, we render the primitives in approximate front-to-back order. As rendering progresses, we use hardware-based occlusion or depth queries to test the visibility of more distant volumes of space. We disable color and z buffer writes, scan convert the boundaries of the spatial cells, and query the hardware to see whether any pixels would have been rendered. If the primitives in a cell would be occluded, we avoid sending them to the graphics card. We present results from both uniform and hierarchical spatial subdivisions. We also use the user-programmable vertex engine for efficient traversal of the subdivisions.

The occlusion-query hardware scan converts the specified primitives to determine whether any frame-buffer pixels would be affected. These queries vary in functionality. The first ones widely available, such as the OpenGL culling extension from Hewlett Packard[1], performed one query at a time. Unfortunately, this test could result in a pipeline stall while waiting for results. More recent versions of culling tests, including a newer one from HP, avoid the stall by pipelining queries on multiple primitives. These tests also separate the procedure calls to render the query primitives from the call to obtain the results. Thus the pipeline can be kept full with either other queries or normal rendering. We are using the NVIDIA OpenGL extension GL_NV_occlusion_query[2], which exploits the occlusion-query hardware available on the GeForce3 and GeForce4

[1] http://oss.sgi.com/projects/ogl-sample/registry/HP/occlusion_test.txt
[2] http://oss.sgi.com/projects/ogl-sample/registry/NV/occlusion_query.txt

cards.

We tested the algorithm on a model containing 13 million triangles. We obtain, on average, a factor of four speedup over view-frustum culling alone. However, the performance difference on difficult frames is more dramatic, with a frame-rate speedup of over ten times. This, of course, includes the overhead associated with the culling algorithm and the depth queries. Note that performance will vary with depth complexity. If there is no occlusion, the queries will slow the system. We have not seen this on a complex model.

Our occlusion culling approach has several advantages. These include:

- It requires no explicit occluder selection, which is a very difficult problem [ZMHH97, KS01].
- Unlike purely hardware-based methods like ATI's Hyper-Z or NVIDIA's Z-Cull, which are meant to reduce demands on fill, the method we present also reduces the bandwidth to the graphics card.
- Our approach involves very little preprocessing and makes no assumption related to model format, connectivity or any explicit information like big occluders.
- It performs conservative occlusion culling.
- It can be easily combined with view-frustum culling as well as LOD- based algorithms for interactive display of very large and complex environments.

Organization: The rest of the paper is organized in the following manner. Section 2 provides an overview of related work. The algorithms based on uniform and hierarchical spatial subdivision are presented in Section 3, and the performance results in Section 4. We close with conclusions and proposed future work in Section 5.

2 Related Work

In this section, we give a brief overview of previous work on occlusion culling and related techniques for faster display of large datasets.

2.1 Occlusion Culling

The problem of computing portions of the scene visible from a given viewpoint is one of the fundamental problems in computer graphics. It has been well studied for more than three decades and a recent survey of different algorithms is given in [COCS01]. In this section, we give a brief overview of occlusion culling algorithms.

Many culling algorithms have been designed for specialized environments, including architectural models based on cells and portals [ARB90, Tel92] and urban datasets composed of large occluders [CT97, HMC$^+$97, SDDS00, WWS00, WWS01]. However, they may not be able to obtain significant culling on large environments composed of a number of small occluders.

Algorithms for general environments can be broadly classified based on whether they are conservative or approximate, whether they use object space or image space hierarchies or whether they compute visibility from a point or a region. The conservative algorithms compute the *potentially visible set* (PVS) that includes all the visible primitives, plus a small number of potentially occluded primitives [CT97, GKM93, HMC$^+$97, KS01, ZMHH97]. On the other hand, the approximate algorithms include most of the visible objects but may also cull away some of the visible objects [BMH99, KS00, ZMHH97].

Object space algorithms make use of spatial partitioning or bounding volume hierarchies; however, it is hard to perform "occluder fusion" on scenes composed of small occluders with object space methods. Image space algorithms including the hierarchical Z-buffer (HZB) [GKM93, Gre01] or hierarchical occlusion maps (HOM) [ZMHH97] are generally more capable of capturing occluder fusion. The HZB approach presents a progressive scheme that involves updating the Z-pyramid after rasterizing each primitive. However, it needs special hardware to support that capability. Greene et al. [GKM93] has also presented a two-pass approach, where it renders the occluders, builds a HZB (e.g. in software) and uses it to cull the geometry. The HOM is a two-pass approach that makes use of texture-mapped rasterization hardware for occlusion culling. It is also able to perform approximate culling based on varying the opacity thresholds parameters used in occlusion maps [ZMHH97]. However, its effectiveness depends on being able to efficiently select all the foreground occluders.

It is widely believed that none of the current algorithms can compute the PVS at interactive rates for complex environments on current graphics systems [ESSS01]. Recently, three different approaches have been proposed to improve their performance.

Region-based visibility algorithms: These pre-compute visibility for a region of space to reduce the runtime overhead [DDTP00, SDDS00, WWS00]. Most of them work well for scenes with large or convex occluders. Nevertheless, there is a tradeoff between the quality of the PVS estimation for a region and the memory overhead. These algorithms may be extremely conservative or not able to obtain significant culling on scenes composed of small occluders.

Hardware visibility queries: A number of image-space visibility queries have been added by manufacturers to their graphics systems to accelerate visibility computations. These include the HP occlusion culling extensions, item buffer techniques, ATI's HyperZ extensions etc. [BMH99, KS01, Gre01, MBH$^+$02]. Their effectiveness varies based on the model and the underlying hardware. [KS01] has presented a two-pass approach that utilizes the GL_HP_occlusion_test and [Gre01] has proposed a modification to improve the performance of HZB. As compared to these approaches, we present a simple and effective progressive occlusion culling algorithm that makes use of the new features of graphics cards, including depth query tests and vertex programs.

Separate visibility server: The use of an additional graphics system as a visibility server has been proposed by [WWS01]. It computes the PVS for a region at runtime in parallel with the main rendering pipeline and works well for urban environments. However, it uses the *occluder shrinking* algorithm [WWS00] to compute the region-based visibility, which works well only if the occluders are large and volumetric in nature. The method also makes assumptions about the user's motion. More recently, Baxter et al. [BSGM02] have used a two-pipeline based occlusion culling algorithm for interactive walkthrough of complex 3D environments. It uses a variation of two-pass HZB algorithm and combines it with hierarchies of levels-of-detail.

2.2 Interactive Display of Large Datasets

Other approaches to faster display rely on the use of image-based representations or the use of multiple acceleration techniques. Image-based impostors can be used to replace geometry distant from the viewpoint and thereby speed up the frame rate. Impostors can be combined with LODs and occlusion culling using a cell based decomposition of the model [ACW+99]. However, the use of impostors can lead to popping or dis-occlusion artifacts because of poor sampling.

A framework to integrate occlusion culling and LODs has been presented in [ASVNB00]. It tries to estimate the degree of visibility of each object in the PVS and uses it to select an appropriate LOD. However, no general and efficient algorithms are known for accurately estimating the degree of visibility in scenes composed of small occluders. Another integrated approach uses the prioritized-layered projection visibility approximation algorithm with view-dependent rendering [ESSS01]. The resulting rendering algorithm performs approximate visibility, as opposed to conservative, and the runtime overhead for large complex environments can be high.

The UC Berkeley Architecture Walkthrough system [FKST96] combined hierarchical algorithms with visibility computations [Tel92] and LODs for architectural models. The BRUSH system [SBM+94] used LODs with hierarchical representation for large mechanical and architectural models. The QSplat system [RL00] elegantly uses a single data structure that combines view frustum culling, backface culling and LOD selection with point rendering for progressive display of large meshes at interactive rates. Another fast approach to render large models is based on interactive ray-tracing. It also provides a solution to the visible surface computation or the occlusion problem. A fast algorithm for distributed ray-tracing of highly complex models has been described in [WSB01]. It can render the Powerplant model at $4-5$ frames a second at 640×480 pixel resolution on a cluster of seven dual processor PCs.

3 Algorithm

We begin by sorting the model geometry into bins based on a spatial subdivision. We can test the visibility of each subdivision cell to determine if its contents should be rendered. Each cell can only be tested against geometry previously drawn in the current frame. Therefore, we would like to test the spatial subdivision cells in a front-to-back ordering from the eye.

An *occlusion query* is accomplished by sending *query geometry* to the graphics card for transformation and rasterization. To complete the occlusion query, we make a function call that returns whether or not any fragment of the query geometry passed the depth test.

We first describe a simple implementation using a uniform grid. We then proceed to describe how to use a hierarchical spatial subdivision (nested grid). Section 3.4 describes approaches we take to reduce the amortized cost of making the necessary occlusion queries.

The choice of spatial subdivision type determines the simplicity or complexity of traversal. We use a uniform or nested grid. Other choices for spatial subdivision could have been made, such as a BSP tree, or a simple octree. Choosing the best subdivision scheme is non-trivial, and model dependent, as is evidenced by experience in the raytacing literature. Our choice of a uniform grid and nested grid is based on amortizing the setup cost in our iterative traversal scheme.

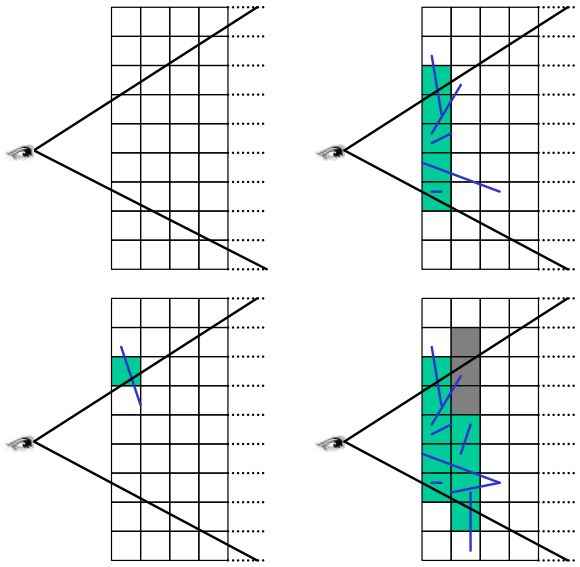

Figure 1: *Algorithm Overview. The first cell is tested and deemed visible (green), so all intersecting geometry is rendered. All the cells in the first "slab" are deemed visible. Some of the cells in the second "slab" are disovered to be not visible (gray).*

We may also have used a bounding box hierarchy. However, bounding box hierarchies raise further complications in terms of traversal order and intelligent construction. Furthermore, the original object definitions are often quite ineffective for the purpose of visibility testing.

3.1 Uniform Grid Decomposition

Model triangles are first sorted into a uniform grid. A Triangle that intersects more than one grid element, or cell, is assigned to each cell that it intersects. We return to the issue of shared triangles at the end of this section.

At render time, the grid is traversed in a front-to-back order with respect to the eye-point. Each cell is tested for visibility. If the cell is found to be visible, all triangles that intersect the cell are rendered.

An occlusion query for a uniform grid cell is as follows:

1. Turn off z and color writes
2. Render the cell (a cube) as query geometry
3. Obtain the result as to whether any part of the query geometry passed the z-test.

The result of the occlusion query is in terms of how many fragments passed the z-test. If the result is zero, the cube would not be visible. Since the bounding cube is not visible, none of its contents are visible. This occlusion query mechanism is provided by the NVIDIA OpenGL extension GL_NV_occlusion_query.

If a triangle intersects more than one grid element, a frame counter is checked to see if the triangle has already been rendered in the current frame. This is to avoid rendering it multiple times. We found that this is faster than simply re-rendering any triangles shared between two cells.

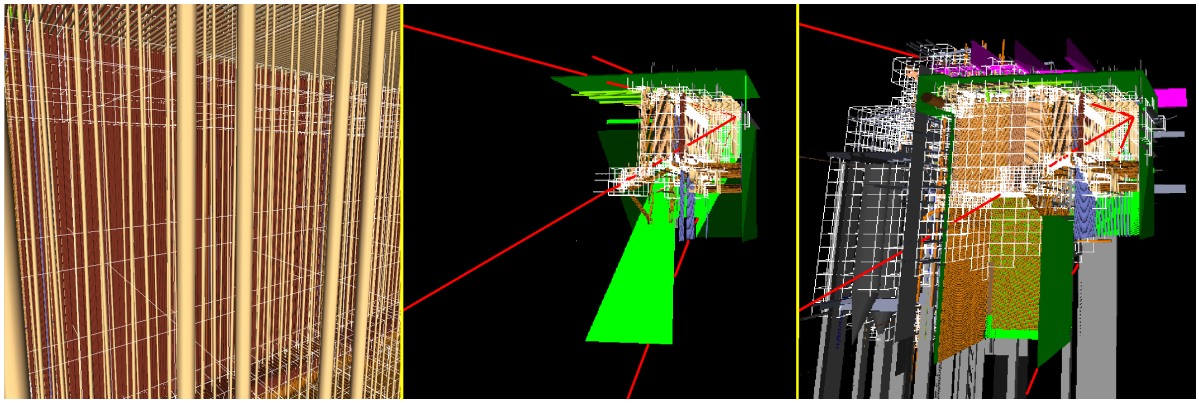

Figure 2: *This view shows a screen shot from the running system with the grid cells rendered in wireframe. A third person view from the side shows the view frustum with the visible cells rendered again in wireframe. The geometry extends outside the cells because all triangles that intersect the visible cells are rendered (triangles are not clipped to cells.) The third image shows the same third person's view using only view frustum culling.*

3.2 Nested Grid Decomposition

The effectiveness of a uniform grid is highly dependent on triangle distribution. Cells that contain many primitives tend to also contain many occluded primitives that are not culled. In order to alleviate this problem, we extend the algorithm presented above to include a hierarchy of grids. Cells found to have an associated set of triangles above some threshold are subdivided further, recursing until a maximum depth is reached, or no cell (leaf) has more than the threshold number of triangles associated with it.

The nested grid is traversed in the same front-to-back manner as the uniform grid, testing for visibility of each cell. In this case, however, if a cell is determined to be visible, and contains a subgrid, we recurse to traverse its contained grid.

3.3 Traversal

Efficient traversal in a front-to-back manner is important. We need to quickly determine the next cell. Choosing cells such that geometry in later cells does not occlude geometry in earlier cells is important to the success of a progressive approach. Our traversal is a variant of the axis aligned slabs used in volume rendering. We use slabs that are equivalent to rasterized planes approximately orthogonal to the view vector.

3.4 Efficient Querying

The performance of the overall algorithm is determined by the number of depth queries that we can perform in the given time frame. The more occlusion queries we can perform, the more model geometry we can potentially cull. We have therefore made an effort to reduce the cost of occlusion queries. This section highlights our approach to minimize both the time to render the query geometry, and the pipeline stalls caused by waiting on query results.

The result of an occlusion query on a particular set of query geometry is not available until the geometry has finished rasterization. This creates a potential for pipeline stalls. We therefore try to keep the pipeline busy by submitting a number of query geometry sets at once. This is, in fact, an explicit design intention of the GL_NV_occlusion_query extension.

The algorithm to keep the pipeline full is as follows:

For each slab, where a slab is a collection of cells as described in Section 3.3:

1. Get the next n cells within the slab, where n is the maximum number of occlusion queries that may be in the pipeline at one time
2. For i = 1 to n
   - Render C_i query geometry (z and color writes off)
3. For i = 1 to n
   - Get result of query for C_i query geometry
   - If C_i is visible:
     - Render the model geometry associated with cell C_i

Between the time a query is submitted, and the time we need the results, a number of other queries and model geometry has been submitted. This will reduce pipeline stalls.

We want all visible geometry intersecting slab i to be rendered before beginning the visibility determination of slab $i+1$. Otherwise, we stand to loose some amount of culling due to occlusion of parts of slab $i+1$ by geometry in slab i.

The regularity of our occlusion representation allows us to exploit a programmable vertex shader to more efficiently render the cubes of the subdivision. We reduce the necessary host to graphics data transfer size, and provide for more efficient transformation of the subdivision cube vertices. For each subdivision grid, we transfer world space origin of the grid, and its scale. For each cube, we send the cube indices for a canonical cube, and the indices defining which grid element the cube will represent. The vertex program computes the positions of all eight vertices of the cube.

3.5 Levels-of-detail

The algorithm we have so far described helps to reduce the number of primitives sent to the graphics card that are occluded or fall outside the view frustum. In some circumstances, the remaining triangles may be too many to render

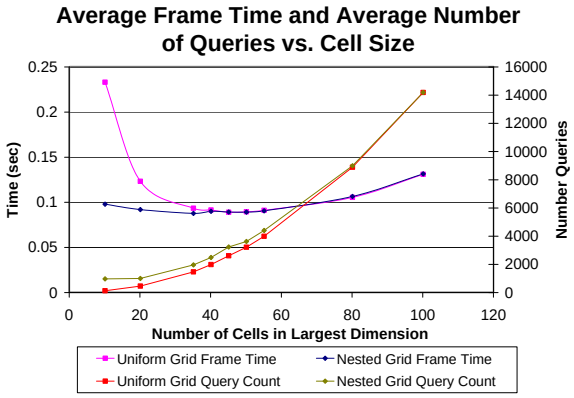

Figure 3: *Avg Frame Rate and Avg Queries vs Cell-Size: The average framerate and query count of the two implementations are graphed here as a function of the grid resolution. The uniform grid implementation is more sensitive to smaller cell sizes while the nested grid implementation can compensate through subdivision. This can be seen by the query plots. However, the frame time mimina are nearly identical and the plots converge as cell size decreases.*

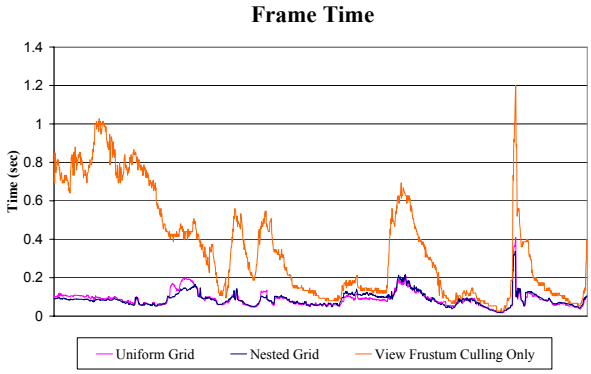

Figure 4: *Frame Time: This figure compares frame time of our best configuration for the nested grid implementation, uniform grid implementation, and view frustum culling only implementation. It is clear that both systems with occlusion culling vastly outperform view frustum culling only. Frame times for nested grid and uniform grid implementations are comparable.*

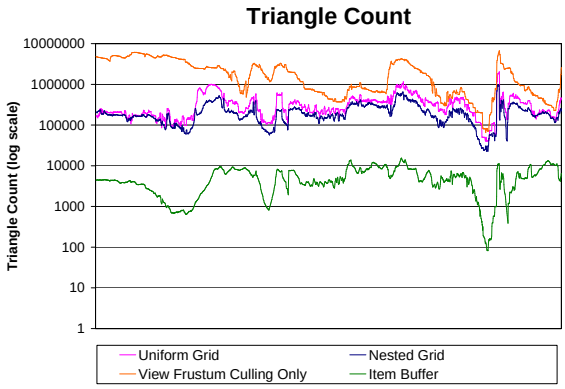

Figure 5: *Triangle Count: This graph shows the number of triangles rendered per frame versus the actual number visible as determined by an item buffer. Triangle counts are given for the nested grid, uniform grid, and view frustum only implementations. The item buffer rendering used the same screen resolution (*800*x*800*) as our other tests.*

at interactive rates. The use of levels- of-detail (LODs) can be used to alleviate this problem.

Our system does not preclude the use of LOD techniques. Triangles could still be sorted into cells as already described. Occlusion culling of primitives would still be determined on the basis of the spatial subdivision. However, the primitives would be stored such that they are identified with the original object and which representation of the object they belong to. At run time, when the contents of a cell are to be rendered, an LOD selection is made, and only those triangles within the cell that belong to that LOD are rendered. This makes it possible to maintain the integrity of the original LODs, while still allowing for occlusion culling with the spatial subdivision.

4 Implementation and Performance

In this section, we describe our implementation and highlight its performance on a complex model. In particular, we tested its performance on a model of a coal-fired power plant with more than 13 million triangles. Much of the upper portion of the model consists of a complex network of piping. Most occlusion in this section arises not from individual pipes, but from an aggregation of the occlusion provided by the pipes. We found that this portion of the model provided one of the most challenge scenario for our occlusion system aside from outside views of the whole model.

Our results are generated from a path through the model that begins on an upper floor, along an exposed walkway (as shown in the video). The path enters an enclosure containing thousands of pipes through a small window, and wanders through this area.

The test runs were performed on a dual processor Pentium 4 machine with a NVIDIA GeForce 4 card and 2 GB of RAM. Note that our application is single threaded. Moreover, the configurations used for optimal performance do not require more than 1GB of RAM for the power plant model.

4.1 Timing Results

There are a number of user specified parameters associated with the performance of our method. For a uniform grid, we can vary the resolution of the grid, and the threshold on the number of triangles in a cell that warrant an occlusion test. If the time to render the model geometry associated with a particular cell is less than the time to perform an occlusion query, then we could simply render the cell contents without the occlusion test. In practice, it is hard to predict the precise values of these times. A cell may intersect a few very large triangles which when rendered may have higher fill-rate requirements, as compared to the cell itself.

Timing comparisons indicated negligible performance differences among threshold values ranging from 1 to 50. For the results presented here, we used a threshold value of 1.

Figure 4 shows the average frame time of our test path with varying grid resolution. We have found that for the

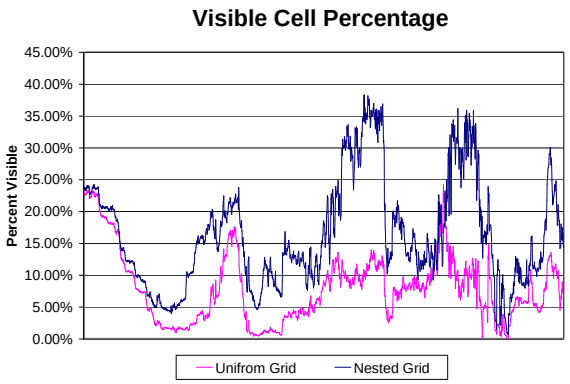

Figure 6: *This graph shows what percentage of the cells were determined to be visible. Given that the top level resolution of the nested grid system is comparable to the resolution of the uniform grid, we see that it is much more difficult to identify geometry to be culled beyond the top level of the hierarchy.*

power plant model, a resolution between 35 and 50 in the maximum dimension gives the best results. The nested grid implementation is much less sensitive to the grid resolution, as a deeper tree makes up for a coarse top level subdivision. It is clear from Figure 4 that the overall performance of our algorithm is much better than using only view frustum culling. The average frame time for the view frustum culling was 0.36 seconds while for our uniform and nested grid approaches were 0.087 and 0.088 seconds, respectively. Moreover, large spikes in the frame time obtained with using only view frustum culling are reduced by the occlusion culling algorithm.

For the nested grid scheme, we used two additional parameters: branching factor, and splitting threshold. If the triangle count of a grid cell is greater than the threshold value, the cell is subdivided according to the branching factor. We have found that a branching factor of 4 in each dimension, or a total of 64 cells, and a threshold of $10,000$ produced the best results.

4.2 Efficiency in Occlusion Culling

We compare the number of triangles in the potentially visible set computed per frame by our method against the exact visible set determined by an item buffer in Figure 5. In the item buffer test each triangle is rendered using a different color. By reading back the color buffer, we were able to determine the number of primitives visible in each frame, up to the screen space resolution. Ideally, we would like our algorithm to exactly compute this visible set, which is also governed by the discrete sampling nature of the frame buffer. By setting our splitting threshold to 150 (an impractical value with regards to performance), we were able to approach within a factor of 10 of the number of triangles determined visible by the item buffer method. View frustum culling alone produces triangle counts that on average are 13 times higher than the fastest configuration for nested grid approximation and 7 times higher than that for the uniform grid.

A measure of the overhead incurred to attain these results is the change in triangle throughput, not counting the primitives used only for occlusion. We have found the throughput

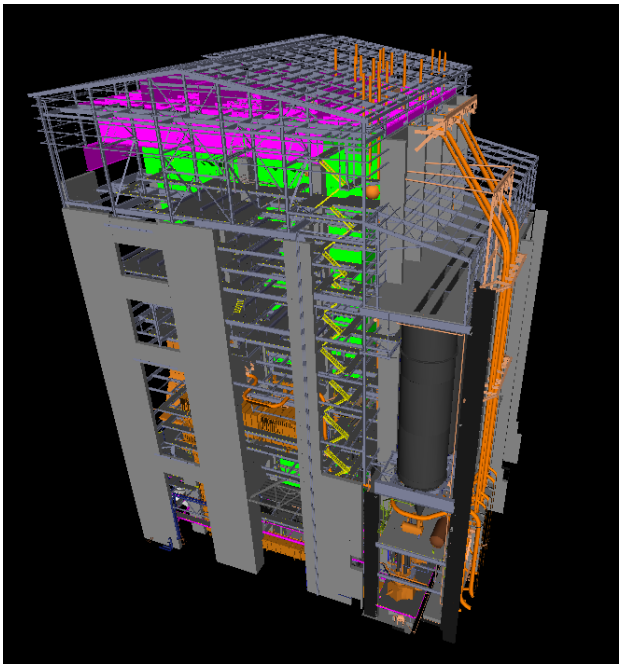

Figure 7: *Powerplant Model: This image shows the outside view. It consists of more than 13 million triangles.*

for the system with view frustum culling only to be 5.23 million triangles per second (MTPS). This is primarily limited by AGP bandwidth, as we are not able to feed the graphics card fast enough. The system presented here obtains 3.79MTPS and 2.17MTPS for the uniform grid and nested grid, respectively. We used standard OpenGL vertex arrays to render model geometry.

Our algorithm renders the grid boundaries to perform occlusion queries. The additional geometry rendered for occlusion querying accounts for part of the decrease in throughput. The remaining throughput reduction is attributed to the stalls that occur when waiting for model geometry rendering to finish, before rendering additional query geometry. This drop in throughput is the cost of performing the occlusion queries using our algorithm. In general, our algorithm will result in improved performance, if occlusion detection can cull a higher percentage of the triangles in the view frustum than the percentage reduction in triangle throughput.

We have shown that for a complex model such as the power plant, the potentially visible set determined by our algorithm on average is 18% of the geometry in the view frustum for a uniform grid method and 9% for a nested grid method. These measures are far less than the triangle throughput utilization of 72% and 41%, and therefore, result in an overall performance increase.

5 Conclusion

We have shown how to effectively use a hardware z-query to accelerate the rendering of models with high depth complexity. The presented data also illustrates the effectiveness of our scheme in terms of achieving the goal of rendering only visible triangles, keeping the pipeline full, and the overhead costs associated with our method. We believe that using our approach, a flat grid will be suitable for most scenarios.

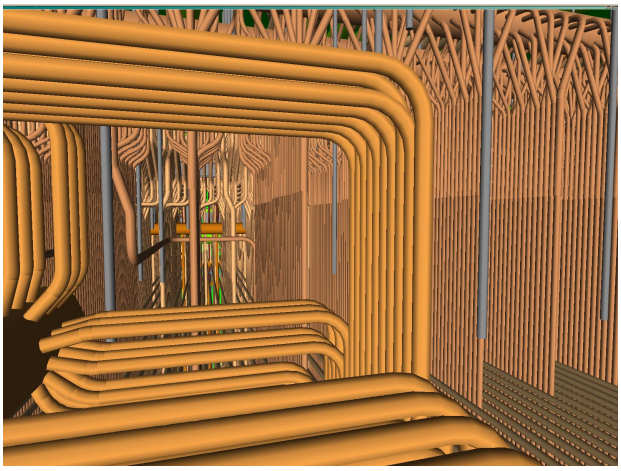

Figure 8: *Powerplant Model: Internal View from our path*

6 Future Work

There are many avenues for future work. We would like to exploit frame-to-frame coherence, perform approximate occlusion culling, and integrate different approaches for LODs. We can further the reduce pipeline stalls caused by waits for query results, and wish to pursue more effective techniques for keeping the pipeline full. Our current algorithm does not address large amounts of visible geometry. We would like to incorporate LODs as discussed in Section 3.5.

Currently, all cells in the view frustum are checked for occlusion corresponding to uniform spatial subdivision. For the nested grid, we check all the cells in the view frustum that are in the top level of the hierarchy. It should be possible to terminate the traversal of the grid in regions of the screen, as they are filled.

Another feature of the GL_NV_occlusion_query extension is the ability to return the number of fragments that actually pass the depth test. This could be used in an approximate occlusion culling scheme where rendering priority would be influenced by the number of fragments passing the z-test. It can be used to select an appropriate static LOD, as suggested in the current OpenGL extension specification. We would also like to extend our occlusion culling algorithm to dynamic environments, which would basically involve an incremental update of the spatial subdivision hierarchies at runtime.

References

[ACW+99] D. Aliaga, J. Cohen, A. Wilson, H. Zhang, C. Erikson, K. Hoff, T. Hudson, W. Stuerzlinger, E. Baker, R. Bastos, M. Whitton, F. Brooks, and D. Manocha. Mmr: An integrated massive model rendering system using geometric and image-based acceleration. In *Proc. of ACM Symposium on Interactive 3D Graphics*, 1999.

[ARB90] J. Airey, J. Rohlf, and F. Brooks. Towards image realism with interactive update rates in complex virtual building environments. In *Symposium on Interactive 3D Graphics*, pages 41–50, 1990.

[ASVNB00] C. Andujar, C. Saona-Vazquez, I. Navazo, and P. Brunet. Integrating occlusion culling and levels of detail through hardly-visibly sets. In *Proceedings of Eurographics*, 2000.

[BMH99] D. Bartz, M. Meibner, and T. Huttner. Opengl assisted occlusion culling for large polygonal models. *Computer and Graphics*, 23(3):667–679, 1999.

[BSGM02] B. Baxter, A. Sud, N. Govindraju, and D. Manocha. Gigawalk: Interactive walkthrough of complex 3d environments. Technical Report TR02-013, Department of Computer Science, University of North Carolina, 2002.

[COCS01] D. Cohen-Or, Y. Chrysanthou, and C. Silva. A survey of visibility for walkthrough applications. *SIGGRAPH Course Notes # 30*, 2001.

[CT97] S. Coorg and S. Teller. Real-time occlusion culling for models with large occluders. In *Proc. of ACM Symposium on Interactive 3D Graphics*, 1997.

[DDTP00] F. Durand, G. Drettakis, J. Thollot, and C. Puech. Conservative visibility preprocessing using extended projections. *Proc. of ACM SIGGRAPH*, pages 239–248, 2000.

[ESSS01] J. El-Sana, N. Sokolovsky, and C. Silva. Integrating occlusion culling with view-dependent rendering. *Proc. of IEEE Visualization*, 2001.

[FKST96] T.A. Funkhouser, D. Khorramabadi, C.H. Sequin, and S. Teller. The ucb system for interactive visualization of large architectural models. *Presence*, 5(1):13–44, 1996.

[GKM93] N. Greene, M. Kass, and G. Miller. Hierarchical z-buffer visibility. In *Proc. of ACM SIGGRAPH*, pages 231–238, 1993.

[Gre01] N. Greene. Occlusion culling with optimized hierarchical z-buffering. In *ACM SIGGRAPH COURSE NOTES ON VISIBILITY, # 30*, 2001.

[HMC+97] T. Hudson, D. Manocha, J. Cohen, M. Lin, K. Hoff, and H. Zhang. Accelerated occlusion culling using shadow frusta. In *Proc. of ACM Symposium on Computational Geometry*, pages 1–10, 1997.

[KCCO00] V. Koltun, Y. Chrysanthou, and D. Cohen-Or. Virtual occluders: An efficient intermediate pvs representation. *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, pages 59–70, 2000.

[KS00] J. Klowoski and C. Silva. The prioritized-layered projection algorithm for visible set estimation. *IEEE Trans. on Visualization and Computer Graphics*, 6(2):108–123, 2000.

[KS01] J. Klowoski and C. Silva. Efficient conservative visiblity culling using the prioritized-layered projection algorithm. *IEEE Trans. on Visualization and Computer Graphics*, 7(4):365–379, 2001.

[MBH+02] M. Meissner, D. Bartz, T. Huttner, G. Muller, and J. Einighammer. Generation of subdivision hierarchies for efficient occlusion culling of large polygonal models. *Computer and Graphics*, 2002. To appear.

[RL00] S. Rusinkiewicz and M. Levoy. Qsplat: A multiresolution point rendering system for large meshes. *Proc. of ACM SIGGRAPH*, 2000.

[SBM+94] B. Schneider, P. Borrel, J. Menon, J. Mittleman, and J. Rossignac. Brush as a walkthrough system for architectural models. In *Fifth Eurographics Workshop on Rendering*, pages 389–399, July 1994.

[SDDS00] G. Schaufler, J. Dorsey, X. Decoret, and F. Sillion. Conservative visibility preprocessing using extended projections. *Proc. of ACM SIGGRAPH*, pages 229–238, 2000.

[Tel92] S. J. Teller. *Visibility Computations in Densely Occluded Polyheral Environments*. PhD thesis, CS Division, UC Berkeley, 1992.

[WSB01] I. Wald, P. Slusallek, and C. Benthin. Interactive distributed ray-tracing of highly complex models. In *Rendering Techniques*, pages 274–285, 2001.

[WWS00] P. Wonka, M. Wimmer, and D. Schmalstieg. Visibility preprocessing with occluder fusion for urban walkthroughs. In *Rendering Techniques*, pages 71–82, 2000.

[WWS01] P. Wonka, M. Wimmer, and F. Sillion. Instant visibility. In *Proc. of Eurographics*, 2001.

[ZMHH97] H. Zhang, D. Manocha, T. Hudson, and K. Hoff. Visibility culling using hierarchical occlusion maps. *Proc. of ACM SIGGRAPH*, 1997.