Parallel Occlusion Culling for Interactive Walkthroughs using Multiple GPUs

Naga K. Govindaraju Avneesh Sud Sung-Eui Yoon Dinesh Manocha
University of North Carolina at Chapel Hill
{naga,sud,sungeui,dm}@cs.unc.edu
http://gamma.cs.unc.edu/switch
http://gamma.cs.unc.edu/switch/video.mpg

Abstract: *We present a new parallel occlusion culling algorithm for interactive display of large environments. It uses a cluster of three graphics processing units (GPUs) to compute an occlusion representation, cull away occluded objects and render the visible primitives. Moreover, our parallel architecture reverses the role of two of the GPUs between successive frames to lower the communication overhead. We have combined the occlusion culling algorithm with pre-computed levels-of-detail and use it for interactive display of geometric datasets. The resulting system has been implemented and applied to large environments composed of tens of millions of primitives. In practice, it is able to render such models at interactive rates with little loss in image fidelity. The performance of the overall occlusion culling algorithm is based on the graphics hardware computational power growth curve which has recently outperformed the Moore's Law for general CPU power growth.*

Keywords: Interactive display, parallel rendering, occlusion culling, cluster computing

1 Introduction

Many applications of CAD, virtual reality and simulation-based design generate large datasets composed of millions of primitives. While graphics processing units (GPUs) have been progressing at a fast rate, the complexity of these models appears to be growing even faster due to the advances in modeling systems and acquisition technologies. As a result, it is a major challenge to render these datasets at interactive rates, i.e. 20 frames a second or more, on current high-end graphics systems.

Given the complexity of these models, different approaches have been proposed for faster display. At a broad level they can be classified into:

- **Parallel Rendering:** These algorithms utilize multiple processors and graphics pipelines. Different algorithms based on shared memory multi-processor systems or clusters of PCs have been proposed.
- **Polygon Flow Minimization:** They attempt to minimize the number of primitives sent to the graphics processor during each frame. They are based on view frustum culling, model simplification, occlusion culling etc.

Most of the parallel rendering algorithms are based on sort-first or sort-last based approaches [MCEF94] or some hybrid combinations. They allocate the primitives among multiple graphics pipelines. However, their performance varies based on the distribution of primitives in the model, the underlying hardware as well as the communication bandwidth.

The polygon flow minimization methods attempt to avoid rendering the primitives that are not ultimately visible. These include primitives that are either not in the field of view or back-facing or are occluded by other objects or their projection is less than a few pixels on the screen. In practice, view frustum culling is used routinely and level-of-detail (LOD) based techniques are being increasingly used to discard primitives whose projection is smaller than a few pixels on the screen. However, no simple and general solutions are known for occlusion culling. Most of the earlier algorithms for occlusion culling fall into two categories. Some of them are specific to architectural or urban environments and not applicable to general environments. The more general approaches either require specialized hardware, extensive pre-processing or the presence of large, easily identifiable occluders in the scene. In fact, performing exact visibility computations on large, general datasets is considered expensive, and hard to achieve in real-time on current graphics systems [ESSS01].

Given the complexity of occlusion culling for general environments, different parallel approaches based on multiple graphics pipelines have been proposed. However, they either make assumptions about the environment or user's motion [WWS01] or are based on shared-memory multiprocessor, multi-pipeline graphics systems [BSGM02] such as SGI Reality Monster, which are quite expensive. On the other hand, there are many advantages of using a parallel occlusion culling algorithm based on commodity hardware components. It allows us to leverage the favorable price-to-performance ratio of PC graphics cards. Moreover, the recent cards including the NVIDIA's GeForce family or ATI's Radeon family have been progressing at a rate higher than Moore's law and it is relatively simple to replace or upgrade them.

Main Results: We present a new parallel occlusion culling algorithm that uses a cluster of three GPUs. The first two GPUs are used to compute an occlusion representation and cull away objects that are not visible from the current viewpoint based on that representation. The third GPU renders the visible geometry by selecting an appropriate level-of-detail. Moreover, the roles of first two GPUs are switched between successive frames to reduce the communication overhead. As compared to earlier image-space occlusion culling algorithms, our approach has lower bandwidth requirements and is able to cull a higher percentage of occluded primitives. It uses multiple GPUs in parallel and doesn't require high-end CPUs for interactive performance. Its overall performance is based on the graphics hardware computational power growth curve which has recently outperformed the general CPU power growth curve based on Moore's Law.

The overall rendering algorithm is general, automatic and applicable to large models. It makes no assumption about model representation or distribution of primitives. The combination of view frustum culling, occlusion culling and levels-

of-detail results in a bounded set of primitives that need to be rendered. In practice, it is almost independent of the input model size. We have implemented the resulting algorithm on a cluster of three PCs connected via Fast Ethernet. We demonstrate its performance on two large environments: a Powerplant model with more than 13 million triangles and a Double Eagle tanker with more than 82 million triangles.

Organization: The rest of the paper is organized in the following manner. We give a brief overview of previous work on parallel rendering and occlusion culling in Section 2. Section 3 describes our parallel occlusion culling algorithm and addresses bandwidth and latency requirements. In Section 4, we combine it with pre-computed levels-of-detail and use it to render large environments. We describe its implementation and highlight its performance on two complex environments in Section 5. Finally, we highlight areas for future research in Section 6.

2 Related Work

In this section, we give a brief overview of previous work on parallel rendering and occlusion culling algorithms.

2.1 Parallel Rendering

A number of parallel algorithms have been proposed in the literature to develop rendering algorithms on shared-memory systems or clusters of PCs. A sorting classification of different approaches for parallel rendering has been described in [MCEF94]. Some of the recent work on rendering large geometric datasets has focused on using PC clusters. These include techniques to assign different parts of the screen to different PCs [SFLS00] as well as distributed algorithms for scalable displays [HBEH00]. Other cluster-based approaches include WireGL, which allows a single serial application to drive a tiled display over a network [HEB$^+$01] as well parallel rendering with k-way replication [SFL01]. The performance of these algorithms varies with different environments as well as the underlying hardware. Most of these approaches are application independent and complimentary to our parallel occlusion algorithm that uses a cluster of three PCs for interactive display.

Parallel algorithms have also been designed for volume rendering [GP93] and ray tracing. These include interactive ray-tracing of volumetric and geometric models on a shared-memory multi-processor system [PMS$^+$99]. A fast algorithm for distributed ray-tracing of highly complex models has been described in [WSB01].

2.2 Occlusion Culling

The problem of computing portions of the scene visible from a given viewpoint is one of the fundamental problems in computer graphics, computational geometry and computer vision. 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. The goal of such algorithms is to cull away primitives that are occluded by other primitives, and therefore, not visible from the current viewpoint. In practice, these algorithms only cull away a subset of the primitives not visible from the current viewpoint and are different from hidden-surface removal algorithms that compute the visible surface.

Many occlusion 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.

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. These include region-based visibility algorithms, use of hardware-based visibility queries and using multiple graphics pipelines in parallel.

2.3 Region-based visibility algorithms

These algorithms 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.

2.4 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]. All these algorithms propose using the same GPU to perform visibility queries as well as render the visible geometry. As a result, only a fraction of a frame time is available for rasterizing the visible geometry. If a scene has no occluded primitives, this approach will slow down their performance.

2.5 Multiple Graphics Pipelines

The use of an additional graphics system as a visibility server has been used by [WWS01, BSGM02]. The approach presented by Wonka et al. [WWS01] 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. The resulting system, GigaWalk, uses a variation of two-pass HZB algorithm and combines it with hierarchies of levels-of-detail. It uses a shared-memory architecture and has been implemented on a SGI Reality Monster and uses two Infinite Reality pipelines and three CPUs. In Section 5, we compare the performance of our algorithm with GigaWalk.

3 Parallel Occlusion Culling

In this section, we present the parallel occlusion culling algorithm. It utilizes multiple graphics processing units (GPUs) and hardware-based visibility query. Our parallel architecture involves using three GPU's, where two GPU's are used to generate an occlusion representation and perform occlusion culling based on the representation, respectively, while the third GPU is used to render the visible geometry.

3.1 Occlusion Representation and Culling

An occlusion culling algorithm has three main components. These include:

1. Compute a set of occluders that correspond to an approximation of the visible geometry.
2. Compute an occlusion representation.
3. Cull away primitives that are not visible based on the occlusion representation.

Different culling algorithms perform these steps either explicitly or implicitly. We use an image-based occlusion representation, as it is able to perform "occlusion fusion" on possibly disjoint occluders [ZMHH97]. Some of the well-known image-based representations include HZB and HOM. But we don't use these hierarchical representations, because the current GPUs do not support these hierarchies in the hardware. Many two-pass occlusion culling algorithms rasterize the occluders, read back the frame-buffer or depth-buffer, and build the hierarchies in software [BSGM02, GKM93, ZMHH97]. However, reading back a high resolution frame-buffer or depth-buffer, say 1024×1024 pixels, can be slow on PC architectures. Moreover, building the hierarchy in software has additional overhead and introduces additional latency in the pipeline.

We use the hardware-based occlusion queries that are becoming common on current GPUs. These queries scan-convert the specified primitives (e.g. bounding boxes) to determine whether the depth of any pixels is affected. Different queries vary in their functionality. Some of the well-known occlusion queries based on the OpenGL culling extension include the HP_Occlusion_Query[1], and the NVIDIA OpenGL extension GL_NV_occlusion_query[2]. These queries can sometime stall the pipelines while waiting for the results. As a result, we dedicate one of the three GPUs to only perform these queries during each frame.

Our algorithm uses the visible geometry from frame i as an approximation to the occluders for frame $i+1$. The occlusion representation implicitly corresponds to the depth buffer after rasterizing all these occluders. The occlusion tests are performed using hardware-based occlusion queries and the system involves no readbacks.

3.2 System Architecture

In this section, we present the overall architecture for parallel occlusion culling. Our algorithm uses three GPUs connected to each other via a LAN. These GPUs perform the following functions, each running as a separate process:

- **Occlusion Representation (OR)**: Render the occluders to compute the occlusion representation. The occluders for frame $i+1$ correspond to the visible geometry from frame i.

[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

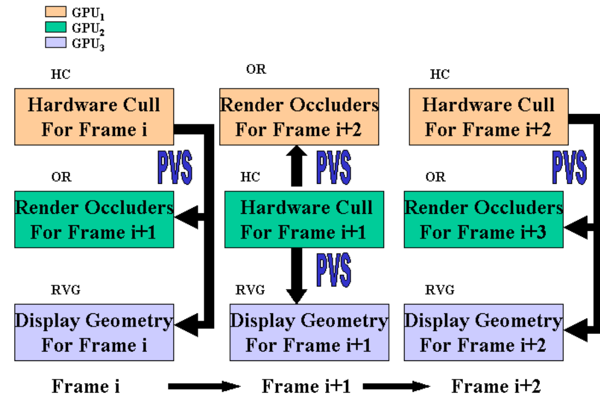


Figure 1: *System Architecture: Each color represents a separate GPU. Note that GPU_1 and GPU_2 switch their roles each frame with one performing hardware culling and other rendering occluders. GPU3 is used as a display client.*

- **Hardware Culling (HC)**: Enable the occlusion query state on the GPU and render the bounding boxes of the scene geometry. Query for the result and send the visible nodes to other two processes. Moreover, we disable modifications to the depth buffer while performing the tests.
- **Render Visible Geometry (RVG)**: Render the visible nodes for the current frame.

Each of these tasks is performed using a separate GPU for each frame. The output of OR and HC is used by other processes. The depth buffer computed by OR is used by HC to perform the occlusion queries. Moreover, the visible nodes or primitives computed by HC are passed onto the RVG (to be used for the current frame) and OR (to be used for the next frame). A key issue in the design and implementation of such an architecture is to minimize the communication between different process. Moreover, each of these tasks is performed in parallel.

In our architecture, we use a GPU to perform occlusion queries for the current frame, another GPU to generate occlusion representation by rendering occluders for the next frame and the third GPU to render the visible primitives for the current frame. We circumvent the problem of transmitting the occlusion representation (OR) from GPU generating OR to GPU performing hardware cull (HC) tests by "switching" their roles between successive frame as shown in Fig. 1. For example, GPU_1 is performing HC for frame i and sending visible nodes to GPU_2 (to be used to compute OR for frame $i+1$) and GPU_3 (to render visible geometry for frame i). For frame $i+1$ GPU_2 has previously computed OR for frame $i+1$. As a result, GPU_2 performs HC and GPU_1 generates the OR for frame $i+2$ and GPU_3 displays the visible primitives. In this case, GPU_1 and GPU_2 form a "switch".

3.3 Bandwidth requirements

In this section, we discuss the bandwidth requirements of our system. In our implementation, we map each node of the scene by the same node identifier across the three different PC's. We transmit this integer node identifier across the network from GPU performing HC to each of GPUs performing OR and RVG if it is visible. This has the advantage over sending the node geometry as it requires relatively smaller bandwidth. So, if the number of visible nodes are n, then GPU performing HC would need to send $4n$ bytes per frame to each of OR and RVG client. Here n refers to number of visible objects and not visible polygons. We can reduce the header overhead by send-

ing multiple integers in a packet - however this incurs some latency due to buffering. The size of camera parameters is 72 bytes and therefore, the bandwidth requirement per frame is $8(nh/b) + 3(72 + h)$ bytes, where h is the size of header in bytes and buffer size b is the number of node ids in a packet. If the frame rate is f frames per second, the total bandwidth required is $8nhf/b + 216f + 3hf$.

3.4 System Latency

A key component of any parallel algorithm implemented using a cluster of PCs is the network latency introduced in terms of transmitting the results from one PC to another during each frame. The performance of our system is dependent on the latency involved in receiving camera parameters by the GPU performing HC and the GPU generating OR for the next frame. There is also a latency in sending camera parameters from GPU performing HC to GPU performing RVG. Moreover, there is latency involved in sending the visible nodes across the network to RVG and OR. We handle the latency problem in receiving the camera parameters by the GPU performing HC using the switching mechanism.

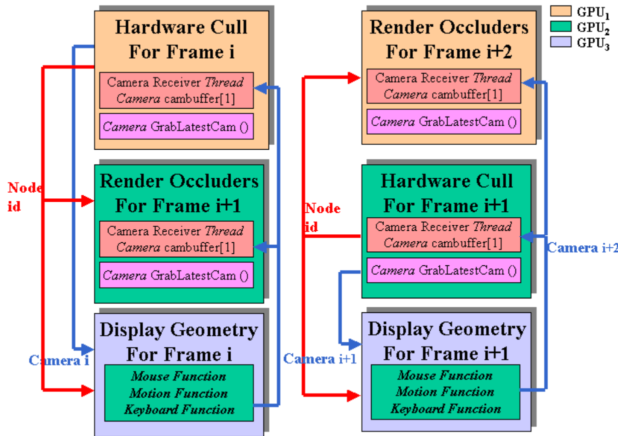


Figure 2: *System Overview: Each color represents a separate GPU with GPU_1 and GPU_2 forming a switch and GPU_3 as the display client. Each of GPU_1 and GPU_2 have a camera-receiver thread and receives camera parameters when the client transmits them due to user's motion and stores them in a camera buffer of size one. The GPU performing OR grabs the latest camera from this thread as the camera position for the next frame. Notice that in this design, the GPU performing HC doesn't have any latency in terms of receiving the camera parameters.*

Let GPU_1 and GPU_2 constitute a switch. GPU_1 performs HC for frame i and GPU_2 generates OR for frame $i + 1$. For frame $i + 1$, GPU_1 generates OR for frame $i + 2$ and GPU_2 performs HC for frame $i + 1$. We note that since GPU_2 has already rendered occluder geometry for frame $i + 1$, it already has the correct camera parameters for performing HC for frame $i + 1$. As a result, there is no additional latency in terms of HC receiving the camera parameters. However, the GPU performing OR requires the camera-parameters from GPU performing RVG. This introduces some latency in terms of receiving the camera parameters. However, since HC requires time to perform Hardware Cull tests before transmitting the first visible node to GPU performing OR, this latency is usually hidden. We reduce the latency in transmitting camera parameters from HC to RVG by sending them in the beginning of frame. Figure 2 illustrates the basic camera transfer routines between the 3 GPU's.

There is some delay introduced by the network. This includes the protocol dependent buffering delays and the hardware level transfer delay. It is important to note that the algorithm would be applicable even if all the GPU's are on the same PC and this would reduce the latencies involved in sending nodes and camera's over network.

3.5 Reliability

The correctness and conservativity of our algorithm is dependent upon the reliable transmission of camera parameters and the visible nodes. Our system is synchronized based on transmission of an end of frame (EOF) packet. This requires us to have reliable transmission of camera parameters from GPU performing HC to GPU performing RVG, Also we require reliable transmission of node ids and EOF from GPU performing HC to each of GPUs performing OR and RVG. We used TCP/IP to transfer data across the network, as it provides a reliable transfer mechanism.

4 Interactive Display

In this section, we present our approach to interactive display of large environments based on the occlusion culling algorithm described above. In particular, we integrate with precomputed static levels-of-detail (LODs) to render large environments. We represent our environment by a scene graph described in [EMB01]. In addition, we store the bounding box of each node in the scene graph, which is used for view frustum culling and occlusion tests. We pre-compute levels-of-detail for each node in the scene graph and also compute hierarchical levels-of-detail HLODs for each intermediate node in the scene graph [EMB01]. Each LOD and HLOD of a node in the scene graph is associated with an error deviation metric that approximately corresponds to the Hausdorff distance between the original model and the simplified object. At runtime, we project this error metric to the screen space and compute the maximum deviation in the silhouette of the original object and its LOD. Our rendering algorithm uses an upper bound on the maximum silhouette deviation error and selects the lowest resolution LOD or HLOD that satisfies the error bound.

```
HardwareCull(Camera *cam)
1    queue = root of scene graph
2    disable color mask and depth mask
3    while( queue is not empty)
4    do
5        node = pop(queue)
6        visible= OcclusionTest(node)
7        if(visible)
8            if(error(node) < pixels of error)
9                Send node to OR and RVG
10           else
11               push children of node to end of queue
12           endif
13       end if
14   end do
```

ALGORITHM 4.1: *Pseudo code for Hardware cull (HC). OcclusionTest renders the bounding box and returns either the number of visible pixels or a boolean depending upon the implementation of query. The function error(node) returns the screen space projection error of the node. Note that if the occlusion test returns the number of visible pixels, we could use it in determining the level at which it needs to be rendered. A detailed explanation is provided in section 4.5.2*

4.1 Culling Algorithm

At run-time, we traverse the scene graph and cull away portions of geometry that are not visible. The visibility of a node is determined by rendering its bounding box against the oc-

clusion representation and querying if it is visible or not. It is important to note that the visibility of a bounding box is a very fast way of rejecting geometry which is not visible. If the bounding box of the node is visible, we test if any of the LOD or HLOD associated with that node meets the error-bound expressed in terms of pixels of error deviation in the silhouette. If a LOD or HLOD is selected, we send the node to the GPU performing OR for the next frame as well as the GPU performing RVG to render the node at the appropriate level-of-detail. If the node is visible but none of the HLOD associated with it satisfy the simplification error bound, we recurse down the scene graph and apply the procedure recursively on each node. On the other hand, if the bounding box of the node is not visible, we do not render that node or any node in the sub-tree rooted at the current node. The pseudocode for the algorithm is described in Algorithm 4.1.

4.2 Occluder Representation Generation

In this section, we describe our algorithm to generate the occlusion representation. At run-time, if GPU performing RVG is processing for frame i, grab camera for frame $i + 1$ from RVG and clear its depth and color buffer. Set the camera parameters for frame $i + 1$. While we receive nodes from GPU performing HC, we render them at the appropriate level of detail. An end-of-frame identifier is sent from HC to notify that no more nodes need to be rendered for this frame.

4.3 SWITCH Algorithm

We now describe the algorithm for "switching" mechanism described in Section 3. The two GPU's involved in the SWITCH toggle their roles of performing HC and generating OR. We use the algorithms described in sections 4.1 and 4.2 to perform HC and OR respectively. The pseudocode for the resulting algorithm is shown in Algorithm 4.2.

```
1   if GPU is generating OR
2       camera=grabLatestCam()
3    end if
4   Initialize the colormask and depth mask to true.
5    if GPU is performing HC
6       Send Camera to RVG
7    else /*GPU needs to render occluders */
8       Clear the color and depth buffer
9    end if
10    Set the camera parameters
11    if GPU is performing HC
12        HardwareCull(camera)
13        Send end of frame to OR and RVG
14    else /* Render occluders */
15        int id= end of frame +1 ;
16        while(id!=end of frame)
17        do
18            id=receive node from HC
19            render(id, camera);
20        end do
21    end if
22    if GPU is performing HC
23        do OR for next frame
24    else
25        do HC for next frame
26    end if
```

ALGORITHM 4.2: *The main algorithm for the GPU's forming the switch. Note that we send the camera parameters to the RVG client at the beginning of HC (on line 6) in order to reduce latency.*

4.4 Render Visible Geometry

The display client receives the camera from HC and sets it for the current frame. In addition, it receives the nodes of scene graph which are determined as visible by the HC and renders them at the appropriate level of detail. Also, the display client transmits the camera information to the GPU's involved in SWITCH based on user interaction. The colormask and depthmask are set to true during initialization. The algorithm for display routine is shown in Algorithm 4.3

```
1   Receive camera from HC
2   Set the camera parameters
3   clear depthbit and colorbit of framebuffer.
4   int id= end of frame +1 ;
5   while(id!=end of frame)
6   do
7       id=receive node from HC
8       render(id, camera);
9   end do
10    end if
```

ALGORITHM 4.3: *Algorithm for the display routine for GPU performing RVG*

4.5 Optimizations

We now describe the optimizations in maximizing the performance of our algorithms. The following are the optimizations:

- **Multiple Occlusion Tests**: Our culling algorithm performs multiple occlusion tests using GL_NV_occlusion_query and this avoids immediate readback of occlusion identifiers, which can stall the pipeline. More details on implementation are described in section 4.5.1.

- **Visibility for LOD Selection**: We utilize the number of visible pixels of geometry queried using GL_NV_occlusion_query in selecting the appropriate LOD. Details are discussed in section 4.5.2.

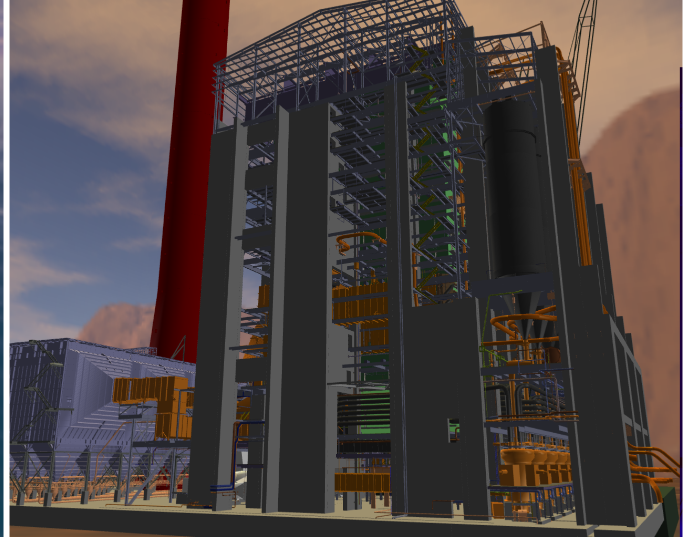
4.5.1 Multiple Occlusion Tests

Our rendering algorithm performs several optimizations to improve the overall performance of the system. The GL_NV_occlusion_query on Nvidia GeForce 3 and GeForce 4 cards allow the programmer to execute multiple occlusion queries at a time and query the results at a later time. We can traverse the scene graph in a breadth first manner and perform all possible occlusion queries for nodes at a level. This would result in a higher performance. Note that certain nodes may be occluded at a level and they are not tested for visibility. We then query the results and determine the visibility of each node. Therefore, if L_i is the list of nodes at level i which are being tested for visibility and pixels of error criteria, we generate the list L_{i+1} to be tested for level $i + 1$ by pushing the children of a node $n \in L_i$ only if its bounding box is visible and it doesn't meet the pixels of error. We use an occlusion identifier for each node in the scene graph and exploit the parallelism available in GL_NV_occlusion_query by performing multiple occlusion queries at each level.

4.5.2 Visibility for LOD Selection

The number of visible pixels of a bounding box of a node provides an upper bound on the number visible pixels of its geometry. The GL_NV_occlusion_query occlusion query also returns the number of pixels visible when the geometry is rendered. We compute the visibility of a node by rendering the bounding box of the node and the query returns the number of visible pixels corresponding to the box. If the number of visible pixels is less than the pixels of error specified by a bound, we don't traverse the scene graph any further at that node. This

(a) Double Eagle tanker rendered at 20 pixels of error



(b) Powerplant rendered at 10 pixels of error

Figure 3: *Both the models are rendered at* $10-20$ *frames per second on a Nvidia GeForce* 4 *GPU at* 1024×1024 *screen resolution.*

Model	Pixels of Error	Average FPS		
		SWITCH	Distributed GigaWalk	GigaWalk
PP	10	10.0	6.2	5.6
DE	20	11.5	4.85	3.50

Table 1: *Average frame rates obtained by different acceleration techniques over the sample path.* **FPS** = *Frames Per Second,* **DE** = *Double Eagle Tanker model,* **PP** = *Power Plant model*

Model	Pixels of Error	Number of Polygons (in 10^5)		
		SWITCH	GigaWalk	Exact Visibility
PP	10	0.9155	1.1924	0.0750
DE	20	1.4163	1.7335	0.1089

Table 2: *Comparison of number of polygons rendered to the actual number of visible polygons by the two implementations.* **DE** = *Double Eagle Tanker model,* **PP** = *Power Plant model*

additional optimization is very useful if only a very small portion of the geometry is visible and the node has a very high screen space projection error associated with it.

5 Implementation and Performance

We have implemented our parallel occlusion culling algorithm on a cluster of three 2.2 GHz Pentium 4 PCs each with 4 GB of RAM and a GeForce 4 Ti 4600 graphics card, running linux and connected via a 100 Mbps switched ethernet.

The scene database is replicated on each PC. Communication of camera parameters and visible node ids between each pair of PCs is handled by a separate TCP/IP stream socket. Synchronization between PCs is maintained by sending a sentinel node over the node sockets to mark an end of frame(EOF).

We compare the performance of our implementation (SWITCH) with the following implementations:

- **GigaWalk**: It is a fast parallel occlusion culling system which uses two IR2 graphics pipelines and three CPUs [BSGM02]. OR and RVG are performed in parallel on two separate graphics pipelines while occlusion culling is performed in parallel using a software based hierarchical Z-buffer. All the interprocess communication is handled using the shared memory.

- **Distributed GigaWalk**: We have implemented a distributed version of GigaWalk on two high end PC's with Nvidia GeForce 4 GPUs. One of the PC's serves as the occlusion server implementing OR and occlusion culling in parallel. The other PC is used as a display client. The occlusion culling is performed in software similar to GigaWalk. Interprocess communication between PCs is done using TCP/IP stream sockets.

Model	Pixels of Error	Number of Objects		
		SWITCH	GigaWalk	Exact Visibility
PP	10	1557	2727	850
DE	20	3313	4036	1833

Table 3: *Comparison of number of objects rendered to the actual number of visible objects by the two implementations.* **DE** = *Double Eagle Tanker model,* **PP** = *Power Plant model*

We compared the performance of the three systems on two complex environments: a coal fired Power Plant (shown in Fig 3(b)) composed of 13 million polygons and 1200 objects, and a Double Eagle Tanker (shown in Fig. 3(a)) composed of 82 million polygons and 127 thousand objects. Figures 4(b) and 4(a) illustrate the interactive performance of our algorithm on a reasonably complex path for Powerplant and Double Eagle models respectively. We have also compared the performance of occlusion culling in terms of the number of objects and polygons rendered and the number of objects and polygons exactly visible. Exact visibility is defined as the number of primitives actually visible from a given viewpoint. It is determined upto the Z-buffer resolution by drawing each primitive in a different color to an "itembuffer" and counting the number of colors visible. Figures 5(a) and 5(b) show our culling performance on the Double Eagle Tanker model. The average speedup in frame rate for the sample paths is shown in Table 1. Tables 2 and 3 summarize the comparison of the primitives rendered by SWITCH and GigaWalk with the exact visibility for polygons and objects respectively. As the scene graph of

the model is organized in terms of objects and we perform visibility tests for objects (not polygons) as primitives, we observe a discrepancy in the ratios of number of primitives rendered to the exact visibility for objects and polygons.

5.1 Bandwidth Estimates

In our experiments, we have observed that number of visible objects n typically ranges in the order of 100 to 4000 depending upon scene complexity. If we render atmost 30 frames per second (fps), header size h (for TCP, IP and ethernet frame) is 50 bytes, buffer size b is 10, then we require a maximum bandwidth of 39 Mbps. Hence, our system is not limited by the available bandwidth on standard ethernet. However, due to the variable window size buffering in TCP/IP [Jac88], we experience network delays under TCP. With UDP, the network delays are significantly lower, but does not guarantee reliability and correctness of our occlusion algorithm.

5.2 Limitations

Our parallel occlusion culling algorithm introduces a frame latency due to multipass rendering.This does not decrease the frame rate as the second pass is performed in parallel. However, it introduces an end-to-end latency and is best suited for latency-tolerant applications.

In addition, a distributed implementation of the algorithm may suffer from network delays, depending upon the implementation of network transmission protocol used. However, our approach is general and is independent of the underlying networking protocol.

6 Conclusions and Future Work

We have presented a new parallel occlusion culling algorithm using multiple GPUs for rendering massive models at interactive rates. In particular, it uses three GPUs in parallel and reverses the role of two of the GPUs between successive frames. We have illustrated the performance of the algorithm on two complex environments and demonstrated its efficiency with two fast parallel occlusion culling implementations.

There are many avenues for future work. These are main related to network communication between different GPUs. A low latency network implementation is highly desirable to maximize the performance achieved by our parallel occlusion culling system. We are working on implementing the system using *Myrinet* which is a low latency network implementation. We are also interested in exploring other low latency implementations like ATM. Finally, we will like to apply our algorithm to more complex environments.

References

[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.

[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. Govindaraju, and D. Manocha. Gigawalk: Interactive walkthrough of complex 3D environments. Technical Report TR02-013, Department of Computer Science, University of North Carolina, 2002. *http://gamma.cs.unc.edu/GigaWalk*

[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.

[EMB01] C. Erikson, D. Manocha, W. Baxter. HLODs for Fast Display of Large Static and Dynamic Environments. *Proceedings of ACM Symposium on Interactive 3D Graphics*, 2001.

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

[GP93] C. Giertsen and J. Peterson. Parallel volume rendering on a network of workstations. *IEEE Computer Graphics and Applications*, pages 16–23, November 1993.

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

[HBEH00] Greg Humphreys, Ian Buck, Matthew Eldridge, and Pat Hanrahan. Distributed rendering for scalable displays. In *Supercomputing*, 2000.

[HEB+01] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan. WireGL: A scalable graphics system for clusters. *Proc. of ACM SIGGRAPH*, 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.

[Jac88] V. Jacobson. Congestion avoidance and control. *Proc. of ACM SIGCOMM*, pages 314–329, 1988

[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.

[MCEF94] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, (4):23–31, 1994.

[PJ01] K. Perrine and D. Jones. Parallel graphics and interactivity with the scaleable graphics engine. *IEEE Supercomputing*, 2001.

[PMS+99] S. Parker, W. Martic, P. Sloan, P. Shirley, B. Smits, and C. Hansen. Interactive ray tracing. *Symposium on Interactive 3D Graphics*, 1999.

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

[SFL01] R. Samanta, T. Funkhouser, and K. Li. Parallel rendering with k-way replication. *IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, 2001.

[SFLS00] R. Samanta, T. Funkhouser, K. Li, and J. P. Singh. Hybrid sort-first and sort-last parallel rendering with a cluster of PCs. *Eurographics/SIGGRAPH workshop on Graphics Hardware*, pages 99–108, 2000.

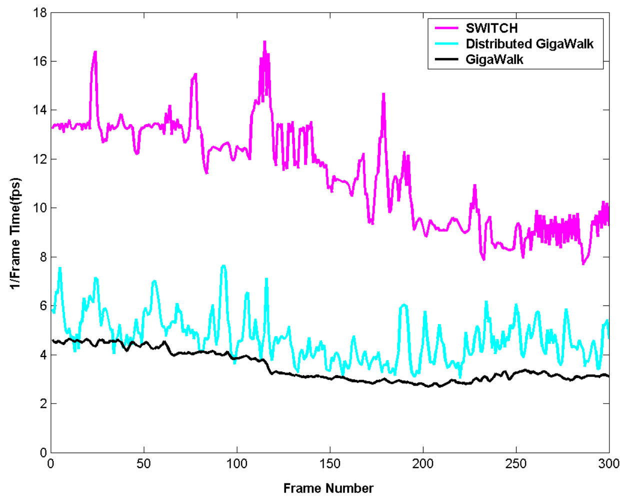
[Tel92] S. J. Teller. *Visibility Computations in Densely Occluded Polyheral Environments*. PhD thesis, CS Division, UC Berkeley, 1992.

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

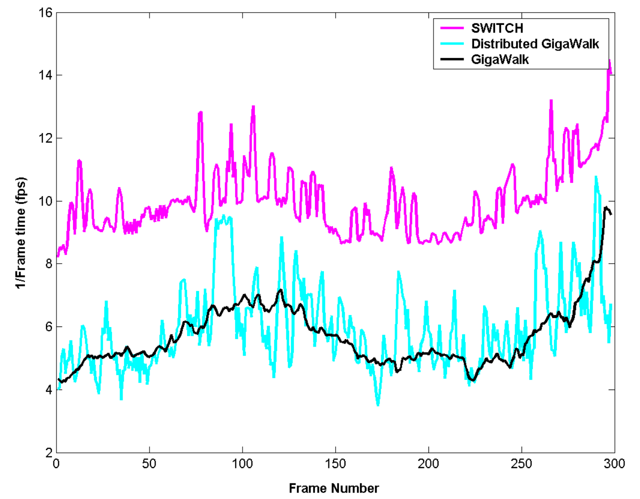
[WWS00] P. Wonka, M. Wimmer, and D. Schmalstieg. Visibility preprocessing with occluder fusion for urban walkthroughs. In *Rendering Techniques*, pages 71–82, 2000.

[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.

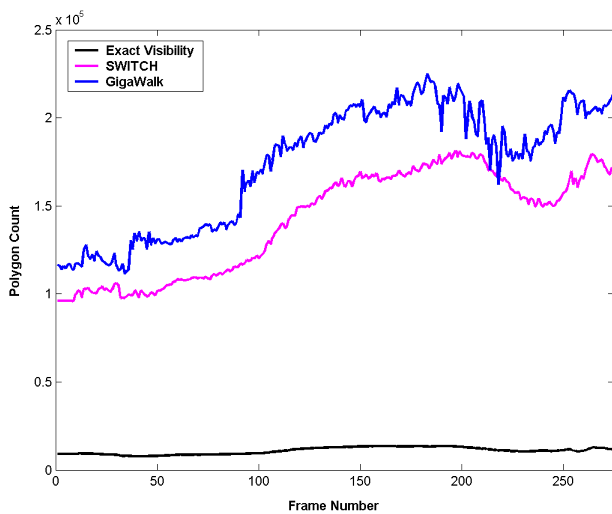


(a) Double Eagle Tanker model at 20 pixels of error

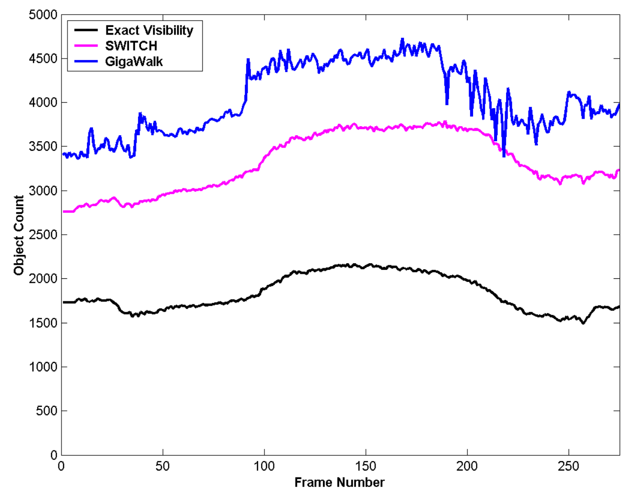


(b) Powerplant model at 10 pixels of error

Figure 4: *Frame rate comparison between SWITCH, GigaWalk and Distributed GigaWalk at* 1024×1024 *screen resolution.*



(a) At polygon level



(b) At object level

Figure 5: *Comparison with exact visibility between SWITCH, GigaWalk and Distributed GigaWalk at* 20 *pixels of error at* 1024×1024 *screen resolution.*