Interactive Visibility Culling in Complex Environments using Occlusion-Switches

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

Abstract: *We present occlusion-switches for interactive visibility culling in complex 3D environments. An occlusion-switch consists of two GPUs (graphics processing units) and each GPU is used to either compute an occlusion representation or cull away primitives not visible from the current viewpoint. Moreover, we switch the roles of each GPU between successive frames. The visible primitives are rendered in parallel on a third GPU. We utilize frame-to-frame coherence to lower the communication overhead between different GPUs and improve the overall performance. The overall visibility culling algorithm is conservative up to image-space precision. This algorithm has been combined with levels-of-detail and implemented on three networked PCs, each consisting of a single GPU. We highlight its performance on complex environments composed of tens of millions of triangles. In practice, it is able to render these environments at interactive rates with little loss in image quality.*

CR Categories and Subject Descriptors: I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling

Keywords: Interactive display, multiple GPUs, conservative occlusion culling, parallel rendering, levels-of-detail

1 Introduction

Interactive display and walkthrough of large geometric environments currently pushes the limits of graphics technology. Environments composed of tens of millions of primitives are common in applications such as simulation-based design of large man-made structures, architectural visualization, or urban simulation. In spite of the rapid progress in the performance of graphics processing units (GPUs), it is not possible to render such complex datasets at interactive rates, i.e., 20 frames a second or more, on current graphics systems.

Many rendering algorithms that attempt to minimize the number of primitives sent to the graphics processor during each frame have been developed. These are based on visibility culling, level-of-detail modeling, sample-based representations, etc. Their goal is to not render any primitives that the user will not ultimately see. These techniques have been extensively studied in computer graphics and related areas.

In this paper, we primarily deal with occlusion culling. Our goal is to cull away a subset of the primitives that are not visible from the current viewpoint. Occlusion culling has been well-studied in the literature and the current algorithms can be classified into different categories. Some are specific to certain types of models, such as architectural or urban environments. Others require extensive preprocessing of visibility, or the presence of large, easily identifiable occluders in the scene, and may not work well for complex environments. The most general algorithms use some combination of object-space hierarchies and image-space occlusion representation. These algorithms can be further classified into three categories:

1. **Specialized Architectures**: Some specialized hardware architectures have been proposed for occlusion culling [Greene et al. 1993; Greene 2001].

2. **Readbacks and Software Culling:** These algorithms read back the frame-buffer or depth-buffer, build a hierarchy, and perform occlusion culling in software [Greene et al. 1993; Zhang et al. 1997; Baxter et al. 2002]. However, readbacks can be expensive (e.g. 50 milliseconds to read back the $1K \times 1K$ depth-buffer on a Dell 530 Workstation with NVIDIA GeForce 4 card).

3. **Utilize Hardware Occlusion Queries:** Many vendors have been supporting image-space occlusion queries. However, their use can impose an additional burden on the graphics pipeline and can sometimes result in reduced throughput and frame rate [Klowoski and Silva 2001].

Overall, occlusion culling is considered quite expensive and hard to achieve in real-time for complex environments.

Main Contribution: We present a novel visibility culling algorithm based on *occlusion-switches*. An occlusion-switch consists of two graphics processing units (GPUs). During each frame, one of the GPUs renders the occluders and computes an occlusion representation, while the second GPU performs culling in parallel using an image-space occlusion query. In order to avoid any depth-buffer readbacks and perform significant occlusion culling, the two GPUs switch their roles between successive frames. The visible primitives computed by the occlusion-switch are rendered in parallel on a third GPU. The algorithm utilizes frame-to-frame coherence to compute occluders for each frame as well as lower the bandwidth or communication overhead between different GPUs. We have combined the occlusion-culling algorithm with static levels-of-detail (LODs) and used it for interactive walkthrough of complex environments. Our current implementation runs on three networked PCs, each consisting of a NVIDIA GeForce 4 graphics processor, and connected using Ethernet. We highlight the performance of our algorithm on three complex environments: a Powerplant model with more than 13 million triangles, a Double Eagle tanker with more than 82 million triangles and a part of a Boeing 777

airplane with more than 20 million triangles. Our system, SWITCH, is able to render these models at $10-20$ frames per second with little loss in image quality. However, our algorithm based on occlusion-switches introduces one frame of latency into the system.

As compared to earlier approaches, our overall occlusion culling and rendering algorithm offers the following advantages:

1. **Generality**: It makes no assumption about the scene and is applicable to all complex environments.

2. **Conservative Occlusion Culling:** The algorithm performs conservative occlusion up to screen-space image precision.

3. **Low Bandwidth:** The algorithm involves no depth-buffer readback from the graphics card. The bandwidth requirements between different GPUs varies as a function of the changes in the visible primitives between successive frames (e.g. a few kilobytes per frame).

4. **Significant Occlusion Culling:** As compared to earlier approaches, our algorithm culls away a higher percentage of primitives not visible from the current viewpoint.

5. **Practicality:** Our algorithm can be implemented on commodity hardware and only assumes hardware support for the occlusion query, which is becoming widely available. Furthermore, we obtain $2-3$ times improvement in frame rate as compared to earlier algorithms.

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 presents occlusion-switches and analyzes the bandwidth requirements. In Section 4, we combine our occlusion culling algorithm with pre-computed levels-of-detail and use it to render large environments. We describe its implementation and highlight its performance on three 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 occlusion culling and parallel rendering.

2.1 Occlusion Culling

The problem of computing portions of the scene visible from a given viewpoint has been well-studied in computer graphics and computational geometry. A recent survey of different algorithms is given in [Cohen-Or et al. 2001]. In this section, we give a brief overview of occlusion culling algorithms. These algorithms aim to cull away a subset of the primitives that are occluded by other primitives and, therefore, are not visible from the current viewpoint.

Many occlusion culling algorithms have been designed for specialized environments, including architectural models based on cells and portals [Airey et al. 1990; Teller 1992] and urban datasets composed of large occluders [Coorg and Teller 1997; Hudson et al. 1997; Schaufler et al. 2000; Wonka et al. 2000; Wonka et al. 2001]. 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 [Coorg and Teller 1997; Greene et al. 1993; Hudson et al. 1997; Klowoski and Silva 2001; Zhang et al. 1997]. On the other hand, the approximate algorithms include most of the visible objects but may also cull away some of the visible objects [Bartz et al. 1999; Klowoski and Silva 2000; Zhang et al. 1997]. Object space algorithms make use of spatial partitioning or bounding volume hierarchies; however, performing "occluder fusion" on scenes composed of small occluders with object space methods is difficult. Image space algorithms including the hierarchical Z-buffer (HZB) [Greene et al. 1993; Greene 2001] or hierarchical occlusion maps (HOM) [Zhang et al. 1997] 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 [El-Sana et al. 2001]. Some of the recent approaches are based on region-based visibility computation, hardware-based visibility queries and multiple graphics pipelines in parallel.

2.2 Region-based Visibility Algorithms

These algorithms pre-compute visibility for a region of space to reduce the runtime overhead [Durand et al. 2000; Schaufler et al. 2000; Wonka et al. 2000]. Most of them work well for scenes with large or convex occluders. Nevertheless, a trade-off occurs between the quality of the PVS estimation for a region and the memory overhead. These algorithms may be extremely conservative or unable to obtain significant culling on scenes composed of small occluders.

2.3 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. [Bartz et al. 1999; Klowoski and Silva 2001; Greene 2001; Meissner et al. 2002; Hillesl et al. 2002]. All these algorithms use the GPU to perform occlusion 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 and it is non-trivial to divide the time between performing occlusion queries and rendering the visible primitives. If a scene has no occluded primitives, this approach will slow down the overall performance. Moreover, the effectiveness of these queries varies based on the model and the underlying hardware.

2.4 Multiple Graphics Pipelines

The use of an additional graphics system as a visibility server has been used by [Wonka et al. 2001; Baxter et al. 2002]. The approach presented by Wonka et al. [2001] 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 [Wonka et al. 2000] 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.

Baxter et al. [2002] 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 that reads back the depth buffer and computes the hierarchy in software. GigaWalk 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.

2.5 Parallel Rendering

A number of parallel algorithms have been proposed in the literature to render large datasets on shared-memory systems or clusters of PCs. These algorithms include techniques to assign different parts of the screen to different PCs [Samanta et al. 2000]. Other cluster-based approaches include WireGL, which allows a single serial application to drive a tiled display over a network [Humphreys et al. 2001] as well as parallel rendering with k-way replication [Samanta et al. 2001]. The performance of these algorithms varies with different environments as well as the underlying hardware. Most of these approaches are application independent and complementary to our parallel occlusion algorithm that uses a cluster of three PCs for interactive display.

Parallel algorithms have also been proposed for interactive ray-tracing of volumetric and geometric models on a shared-memory multi-processor system [Parker et al. 1999]. A fast algorithm for distributed ray-tracing of highly complex models has been described in [Wald et al. 2001].

3 Interactive Occlusion Culling

In this section, we present occlusion-switches and use them for visibility culling. The resulting algorithm uses multiple graphics processing units (GPUs) with image-space occlusion query.

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. Use the occlusion representation to cull away primitives that are not visible.

Different culling algorithms perform these steps either explicitly or implicitly. We use an image-based occlusion representation because it is able to perform "occluder fusion" on possibly disjoint occluders [Zhang et al. 1997]. Some of the well-known image-based hierarchical representations include HZB [Greene et al. 1993] and HOM [Zhang et al. 1997]. However, 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 [Baxter et al. 2002; Greene et al. 1993; Zhang et al. 1997].

However, reading back a high resolution frame-buffer or depth-buffer can be slow on PC architectures. Moreover, constructing the hierarchy in software incurs additional overhead.

We utilize the hardware-based occlusion queries that are becoming common on current GPUs. These queries scan-convert the specified primitives (e.g. bounding boxes) to check whether the depth of any pixels changes. Different queries vary in their functionality. Some of the well-known occlusion queries based on the OpenGL culling extension include the HP_Occlusion_Query (`http://oss.sgi.com/projects/ogl-sample/registry/HP/occlusion\_test.txt`) and the NVIDIA OpenGL extension GL_NV_occlusion_query (`http://oss.sgi.com/projects/ogl-sample/registry/NV/occlusion\_query.txt`). These queries can sometime stall the pipelines while waiting for the results. As a result, we use a specific GPU during each frame to perform only these queries.

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. The *occlusion switches* are used to compute the occlusion representation and perform these queries.

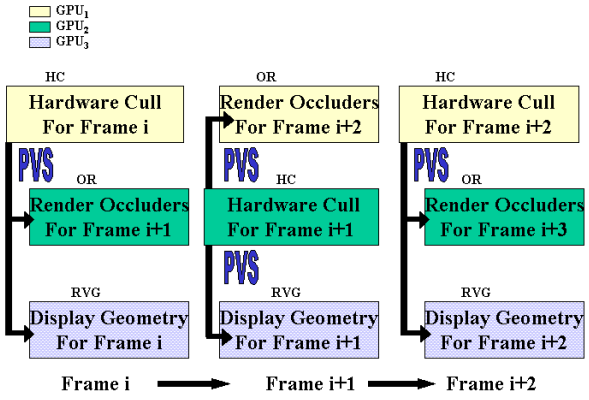


Figure 1: *System Architecture: Each color represents a separate GPU. Note that GPU_1 and GPU_2 switch their roles each frame with one performing hardware culling and other rendering occluders. GPU3 is used as a display client.*

3.2 Occlusion-Switch

An occlusion-switch takes the camera for frame $i+1$ as input and transmits the potential visible set and camera for frame i as the output to the renderer. The occlusion-switch is composed of two GPUs, which perform the following functions, each running on a separate GPU in parallel:

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

- **Hardware Culling (HC)**: Enable the occlusion query state on the GPU and render the bounding boxes corresponding to the scene geometry. Use the image-space occlusion query to determine the visibility of each bounding box and compute the PVS. Moreover, we disable modifications to the depth buffer while performing these queries.

During a frame, each GPU in the occlusion-switch performs either OR or HC and at the end of the frame the two GPUs inter-change their function. The depth buffer computed by OR during the previous frame is used by HC to perform the occlusion queries during the current frame. Moreover, the visible nodes computed by HC correspond to the PVS. The PVS is rendered in parallel on a third GPU and is used by the OR for the next frame to compute the occlusion representation. The architecture of the overall system is shown in Fig. 1. The overall occlusion algorithm involves no depth buffer readbacks from the GPUs.

3.3 Culling Algorithm

The occlusion culling algorithm uses an occlusion-switch to compute the PVS and renders them in parallel on a separate GPU. Let GPU_1 and GPU_2 constitute the occlusion-switch and GPU_3 is used to render the visible primitives (RVG). In an occlusion-switch, the GPU performing HC requires OR for occlusion tests. We circumvent the problem of transmitting occlusion representation from the GPU generating OR to GPU performing hardware cull tests by "switching" their roles between successive frames 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, GPU_1 generates the OR for frame $i+2$ and GPU_3 displays the visible primitives.

3.4 Incremental Transmission

The HC process in the occlusion culling algorithm computes the PVS for each frame and sends it to the OR and RVG. To minimize the communication overhead, we exploit frame-to-frame coherence in the list of visible primitives. All the GPUs keep track of the visible nodes in the previous frame and the GPU performing HC uses this list and only transmits the changes to the other two GPUs. The GPU performing HC sends the visible nodes to OR and RVG, and therefore, it has information related to the visible set on HC. Moreover, the other two processes, OR and RVG, maintain the visible set as they receive visible nodes from HC. To reduce the communication bandwidth, we transmit only the difference in the visible sets for the current and previous frames. Let V_i represent the potential visible set for frame i and $\delta_{j,k} = V_j - V_k$ be the difference of two sets. During frame i, HC transmits $\delta_{i,i-1}$ and $\delta_{i-1,i}$ to OR and RVG, respectively. We reconstruct V_i at OR and RVG based on the following formulation:

$$V_i = (V_{i-1} - \delta_{i-1,i}) \cup \delta_{i,i-1}.$$

In most interactive applications, we expect that the size of the set $\delta_{i-1,i} \cup \delta_{i,i-1}$ is much smaller than that of V_i.

3.5 Bandwidth Requirements

In this section, we discuss the bandwidth requirements of our algorithm for a distributed implementation on three different graphics systems (PCs). Each graphics system consists of a single GPU and they are connected using a network. In particular, we map each node of the scene by the same node identifier across the three different graphics systems. We transmit this integer node identifier across the network from the GPU performing HC to each of the GPUs performing OR and RVG. This procedure is more efficient than sending all the triangles that correspond to the node as it requires relatively smaller bandwidth per visible node (i.e. 4 bytes per node). So, if the number of visible nodes is n, then GPU performing HC must send $4n$ bytes per frame to each OR and RVG client. Here n refers to the number of visible objects and not the visible polygons. We can reduce the header overhead by sending multiple integers in a packet. However, this process can introduce some extra latency in the pipeline due to buffering. Moreover, the size of camera parameters is 72 bytes; consequently, the bandwidth requirement per frame is $8n + nh/b + 3(72 + h)$ 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 $8nf + nhf/b + 216f + 3hf$. If we send visible nodes by incremental transmission, then n is equal to the size of $\delta_{i,i-1} \cup \delta_{i-1,i}$.

4 Interactive Display

In this section, we present our overall rendering algorithm for interactive display of large environments. We use the occlusion culling algorithm described above and combines it with pre-computed static levels-of-detail (LODs) to render large environments. We represent our environment using a scene graph, as described in [Erikson et al. 2001]. We describe the scene graph representation and the occlusion culling algorithm. We also highlight many optimizations used to improve the overall performance.

4.1 Scene Graph

Our rendering algorithm uses a scene graph representation along with pre-computed static LODs. Each node in the scene graph stores references to its children as well as its parent. In addition, we store the bounding box of each node in the scene graph, which is used for view frustum culling and occlusion queries. This bounding box may correspond to an axis-aligned bounding box (AABB) or an oriented bounding box (OBB). We pre-compute the LODs for each node in the scene graph along with hierarchical levels-of-detail (HLODs) for each intermediate node in the scene graph [Erikson et al. 2001]. Moreover, each LOD and HLOD is represented as a separate node in the scene graph and we associate 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 corresponding LOD or HLOD. 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 to compute the level at which it must be rendered.*

4.2 Culling Algorithm

At runtime, we traverse the scene graph and cull away portions of geometry that are not visible. The visibility of a node is computed by rendering its bounding box against the occlusion representation and querying if it is visible or not. Testing the visibility of a bounding box is a fast and conservative way to reject portions of the scene that are not visible. If the bounding box of the node is visible, we test whether any of the LODs or HLODs associated with that node meet the pixel-deviation error-bound. If one of the LODs or HLODs is selected, we include that node in the PVS and send it to the GPU performing OR for the next frame as well as to the GPU performing RVG for the current frame. If the node is visible and none of the HLODs associated with it satisfy the simplification error bound, we traverse 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. The image-space occlusion query is used to perform view frustum culling as well as occlusion culling on the

bounding volume.

4.3 Occluder Representation Generation

At runtime, if we are generating OR for frame $i+1$, we receive camera $i+1$ from RVG and set its parameters. We also clear its depth and color buffer. While OR receives 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.

4.4 Occlusion-Switch Algorithm

We now describe the algorithm for the "switching" mechanism described in Section 3. The two GPU's involved in the occlusion-switch toggle or interchange their roles of performing HC and generating OR. We use the algorithms described in sections 4.2 and 4.3 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 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 implementation of occlusion-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.*

4.5 Render Visible Geometry

The display client, RVG, receives the camera for the current frame from HC. In addition, it receives the visible nodes in the scene graph and renders them at the appropriate level-of-detail. Moreover, the display client transmits the camera information to the GPU's involved in occlusion-switch based on user interaction. The colormask and depthmask are set to true during initialization.

4.6 Incremental Traversal and Front Tracking

The traversal of scene graph defines a cut that can be partitioned into a visible front and an occluded front.

- **Visible Front**: Visible front is composed of all the visible nodes in the cut. In addition, each node belonging to the visible front satisfies the screen space error metric while its parent does not.

- **Occluded Front**: Occluded front is composed of all the occluded nodes in the cut. Also, note that an occluded node may not satisfy the screen space error metric.

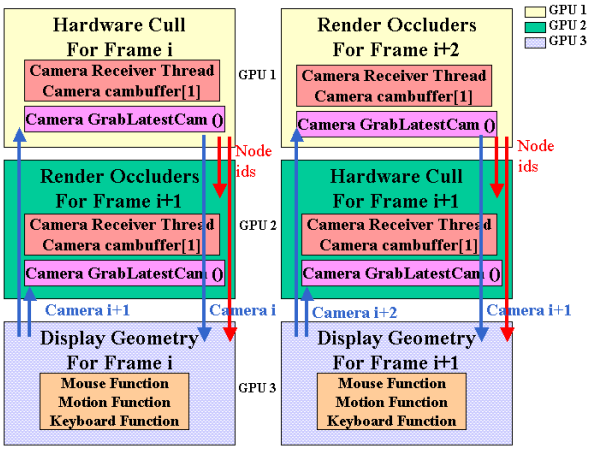


Figure 2: *System Overview: Each color represents a separate GPU with GPU_1 and GPU_2 forming a switch and GPU_3 as the display client. Each of GPU_1 and GPU_2 has a camera-receiver thread and receives camera parameters when the client transmits them due to user's motion and stores them in a camera buffer 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.*

We reduce the communication overhead by keeping track of the visible and occluded fronts from the previous frame at each GPU. Each node in the front is assigned one of the following states:

- **Over-refined**: Both the node and its parent satisfy the silhouette deviation metric in screen space.

- **Refined**: The node satisfies the silhouette deviation metric while the parent does not.

- **Under-refined**: The node does not satisfy the silhouette deviation metric.

Each node in the front is updated depending upon its state. If the node is *Over-refined*, we traverse up the scene graph to reach a parent node which is *Refined*. If the node is *Under-refined*, we traverse down the scene graph generating a set of *Refined* children nodes. At the beginning of each frame, both OR and RVG update the state of each node in the visible front before rendering it.

We also render each node in $\delta_{i,i-1}$ at OR and RVG. At the end of the frame, the visible nodes for the current frame are reconstructed as described in Section 3.4. The update of the state of each node is important for maintaining the conservative nature of the algorithm.

At the GPU performing HC, we also maintain the occluded front in addition to the visible front of previous frame. This enables us to compute $\delta_{i,i-1}$ efficiently by performing culling on the occluded front before the visible front. A node in the occluded front is refined only if it is in the Over-refined state. Each of the occluded fronts and visible fronts is refined before performing culling algorithm on the refined fronts. Moreover, $\delta_{i,i-1}$ is a part of the refined occluded front.

4.7 Optimizations

We use a number of optimizations to improve the performance of our algorithms, including:

- **Multiple Occlusion Tests**: Our culling algorithm performs multiple occlusion tests using GL_NV_occlusion_query; this avoids immediate readback of occlusion identifiers, which can stall the pipeline. More details on implementation are described in section 4.7.1.

- **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 4.7.2.

4.7.1 Multiple Occlusion Tests

Our rendering algorithm performs several optimizations to improve the overall performance. The GL_NV_occlusion_query on NVIDIA GeForce 3 and GeForce 4 cards allows multiple occlusion queries at a time and query the results at a later time. We traverse the scene graph in a breadth first manner and perform all possible occlusion queries for the nodes at a given level. This traversal results in an improved performance. Note that certain nodes may be occluded at a level and are not tested for visibility. After that we query the results and compute the visibility of each node. Let L_i be the list of nodes at level i which are being tested for visibility as well as pixel-deviation error. We generate the list L_{i+1} that will be tested at level $i+1$ by pushing the children of a node $n \in L_i$ only if its bounding box is visible, and it does not satisfy the pixel-deviation error criterion. 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.

4.7.2 Visibility for LOD Selection

The LODs in a scene graph are associated with a screen space projection error. We traverse the scene graph until each LOD satisfies the pixels-of-error metric. However, this approach can be too conservative if the object is mostly occluded. We therefore utilize the visibility information in selecting an appropriate LOD or HLOD of the object.

The number of visible pixels for a bounding box of a node provides an upper bound on the number of visible pixels for its geometry. The GL_NV_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 do not traverse the scene graph any further at that node. This additional optimization is very useful if only a very small portion of the bounding box is visible, and the node has a very high screen space projection error associated with it.

4.8 Design Issues

Latency and reliability are two key components considered in the design of our overall rendering system. In addition to one frame of latency introduced by an occlusion-switch, our algorithm introduces additional latency due to the transfer of camera parameters and visible node identifiers across the network. We also require reliable transfer of data among different GPUs to ensure the correctness of our approach.

4.8.1 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 the camera parameters by the GPUs involved in occlusion-switch. In addition,

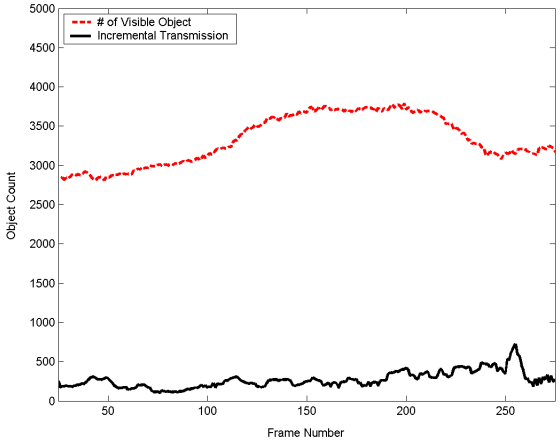


Figure 3: *Comparison of number of nodes transmitted with and without incremental transmission (described in section 4.6) for a sample path on Double Eagle Tanker model. Using incremental transmission, we observe an average reduction of 93% in the number of nodes transmitted between the GPUs.*

there is latency in terms of sending the camera parameters from the GPU performing HC to the GPU performing RVG. Moreover, latency is also introduced in sending the visible nodes across the network to RVG and OR. We eliminate the latency problem in receiving the camera parameters by the GPU performing HC using the switching mechanism.

Let GPU_1 and GPU_2 constitute an occlusion-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 $i+1$. Given that GPU_2 has already rendered the occluders for frame $i+1$, it already has the correct camera parameters for performing HC for frame $i+1$. As a result, no additional latency occurs in terms of HC receiving the camera parameters. However, the GPU performing OR requires the camera-parameters from the GPU performing RVG. This introduces latency in terms of receiving the camera parameters. Because HC takes some 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 each frame. Figure 2 illustrates the basic protocol for transferring the camera parameters among the three GPU's. We enumerate other sources of network latency in Section 5.2.

4.8.2 Reliability

The correctness and conservativity of our algorithm depends on the reliable transmission of camera parameters and the visible nodes between the GPUs. Our system is synchronized based on transmission of an end of frame (EOF) packet. This protocol 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 GPU performing OR and RVG. We used reliable transfer protocols (TCP/IP) to transfer the data across the network.

5 Implementation and Performance

We have implemented our parallel occlusion culling algorithm on a cluster of three 2.2 GHz Pentium-4 PCs, each having 4 GB of RAM (on an Intel 860 chipset) and a GeForce 4 Ti 4600 graphics card. Each runs Linux 2.4, with bigmem option enabled giving 3.0 GB user addressable memory. The

Model	Pixels of Error	Average FPS SWITCH	Distributed GigaWalk	GigaWalk
PP	5	14.17	6.2	5.6
DE	20	10.31	4.85	3.50
B-777	15	13.01	5.82	

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,* **B-777** *= Boeing 777 model*

Model	Pixels of Error	Number of Polygons SWITCH	GigaWalk	Exact Visibility
PP	5	91550	119240	7500
DE	20	141630	173350	10890

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*

PCs are connected via 100 Mb/s Ethernet. We typically obtain a throughput of $1-2$ million triangles per second in immediate mode using triangle strips on these graphics cards. Using NVIDIA OpenGL extension GL_NV_occlusion_query, we perform an average of around $50,000$ queries per second.

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 over Ethernet. Synchronization between the PCs is maintained by sending a sentinel node over the node sockets to mark an end of frame(EOF).

We compare the performance of the implementation of our algorithm (called SWITCH) with the following algorithms and implementations:

- **GigaWalk**: A fast parallel occlusion culling system which uses two IR2 graphics pipelines and three CPUs [Baxter et al. 2002]. 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 PCs with NVIDIA GeForce 4 GPUs. One of the PCs 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 based on TCP/IP stream sockets.

We compared the performance of the three systems on three complex environments: a coal fired Power Plant (shown in the color plate) composed of 13 million polygons and 1200 objects, a Double Eagle Tanker (shown in the color plate) composed of 82 million polygons and $127K$ objects, and part of a Boeing 777 (shown in the color plate) composed of 20 million triangles and $52K$ objects. Figures 4, 5(a) and 5(b) illustrate the performance of SWITCH on a complex path in the Boeing 777, Double Eagle and Powerplant models, respectively (as shown in the video). Notice that we are able to obtain $2-3$ times speedups over earlier systems.

We have also compared the performance of occlusion culling algorithm in terms of the number of objects and polygons rendered as compared to the number of objects and polygons exactly visible. *Exact visibility* is defined as the number of primitives actually visible up to the screen-space and depth-buffer resolution from a given viewpoint. The exact visibility is computed by drawing each primitive in a different color to an "itembuffer" and counting the number of colors visible. Figures 6(a) and 6(b) show the culling performance of our algorithm on the Double Eagle Tanker model.

Model	Pixels of Error	Number of Objects SWITCH	GigaWalk	Exact Visibility
PP	5	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*

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 at an object level and not at the polygon level. Consequently, 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 the number of visible objects n typically ranges in the order of 100 to 4000 depending upon scene complexity and the viewpoint. If we render at most 30 frames per second (fps), header size h (for TCP, IP and ethernet frame) is 50 bytes and buffer size b is 100 nodes per packet, then we require a maximum bandwidth of 8.3 Mbps. Hence, our system is not limited by the available bandwidth on fast ethernet. However, the variable window size buffering in TCP/IP [Jacobson 1988], introduces network latency. The incremental transmission algorithm greatly lowers the communication overhead between different GPUs. Figure 3 shows the number of node identifiers transmitted with and without incremental transmission for a sample path in the Double Eagle Tanker model. We observe a very high frame-to-frame coherence and an average reduction of 93% in the bandwidth requirements. During each frame, the GPUs need to transmit pointers to a few hundred nodes, which adds up to a few kilobytes. The overall bandwidth requirement is typically a few megabytes per second.

5.2 Performance Analysis

In this section, we analyze different factors that affect the performance of occlusion-switch based culling algorithm. One of the key issues in the design of any distributed rendering algorithm is system latency. In our architecture, we may experience latency due to one of the following reasons:

1. **Network** : Network latencies mainly depend upon the implementation of transport protocol used to communicate between the PCs. The effective bandwidth varies depending on the packet size. Implementations like TCP/IP inherently buffer the data and may introduce latencies. Transmission of a large number of small size packets per second can cause packet loss and re-transmission introduces further delays. Buffering of node ids reduces loss but increases network latency.

2. **Hardware Cull** : The occlusion query can use only a limited number of identifiers before the results of pixel count are queried. Moreover, rendering a bounding box usually requires more resources in terms of fill-rate as compared to rasterizing the original primitives. If the application is fill-limited, HC can become a bottleneck

in the system. In our current implementation, we have observed that the latency in HC is smaller as compared to the network latency. Using a front based ordered culling, as described in section 4.6, reduces the fill-requirement involved in performing the queries and results in a better performance.

3. **OR and RVG** : OR and RVG can become bottlenecks when the number of visible primitives in a given frame is very high. In our current implementation, HC performs culling at the object level. As a result, the total number of polygons rendered by OR or RVG can be quite high depending upon the complexity of the model, the LOD error threshold and the position of the viewer. We can reduce this number by selecting a higher threshold for the LOD error.

The overall performance of algorithm is governed by two factors: culling efficiency for occlusion culling and the overall frame-rates achieved by the rendering algorithm.

- **Culling Efficiency**: Culling efficiency is measured in terms of the ratio of number of primitives in the potential visible set to the number of primitives visible. The culling efficiency of occlusion-switch depends upon the occlusion-representation used to perform culling. A good selection of occluders is crucial to the performance of HC. The choice of bounding geometric representation used to determine the visibility of an object affects the culling efficiency of HC. In our current implementation, we have used rectangular bounding box as the bounding volume because of its simplicity. As HC is completely GPU-based, we can use any other bounding volume (e.g. a convex polytope, k-dop) and the performance of the query will depend on the number of triangles used to represent the boundary of the bounding volume.

- **Frame Rate**: Frame rate depends on the culling efficiency, load balancing between different GPUs and the network latency. Higher culling efficiency results in OR and RVG rendering fewer number of primitives. A good load balance between the occlusion-switch and the RVG would result in maximum system throughput. The order and the rate at which occlusion tests are performed affects the load balance across the GPUs. Moreover, the network latency also affects the overall frame rate. The frame rate also varies based on the LOD selection parameter.

With faster GPUs, we would expect higher culling efficiency as well as improved frame rates.

5.3 Comparison with Earlier Approaches

We compare the performance of our approach with two other well-known occlusion culling algorithms: HZB [Greene et al. 1993] and HOM [Zhang et al. 1997]. Both of these approaches use a combination of object-space and image-space hierarchies and are conservative to the image precision. Their current implementations are based on frame-buffer readbacks and performing the occlusion tests in software. The software implementation incurs additional overhead in terms of hierarchy construction. Moreover, they project the object's bounding volume to the screen space and compute a 2D screen space bounding rectangle to perform the occlusion test. As a result, these approaches are more conservative as compared to occlusion-switch based culling algorithm. Further, the frame-buffer or depth-buffer readbacks can be expensive as compared to the occlusion queries, especially on current PC systems. In practice, we obtained almost three

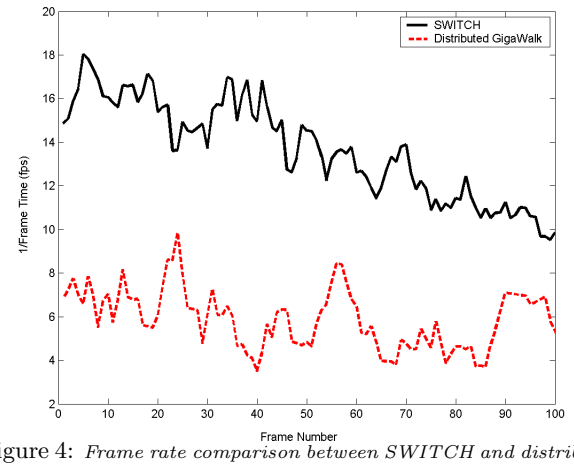


Figure 4: *Frame rate comparison between SWITCH and distributed Gigawalk at* 1024×1024 *screen resolution and* 15 *pixels of error on Boeing model.*

times speed-up over an implementation of HZB on two PCs (Distributed GigaWalk).

Our algorithm also utilizes the number of visible pixels parameter returned by GL_NV_occlusion_query for LOD selection. This bound makes our rendering algorithm less conservative as compared to earlier LOD-based rendering algorithms, which only compute a screen space bound from the object space deviation error.

5.4 Limitations

Occlusion-switch based culling introduces an extra frame of latency in addition to double-buffering. The additional latency does not decrease the frame rate as the second pass is performed in parallel. However, it introduces additional latency into the system; the overall algorithm 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. Our overall approach is general and independent of the underlying networking protocol.

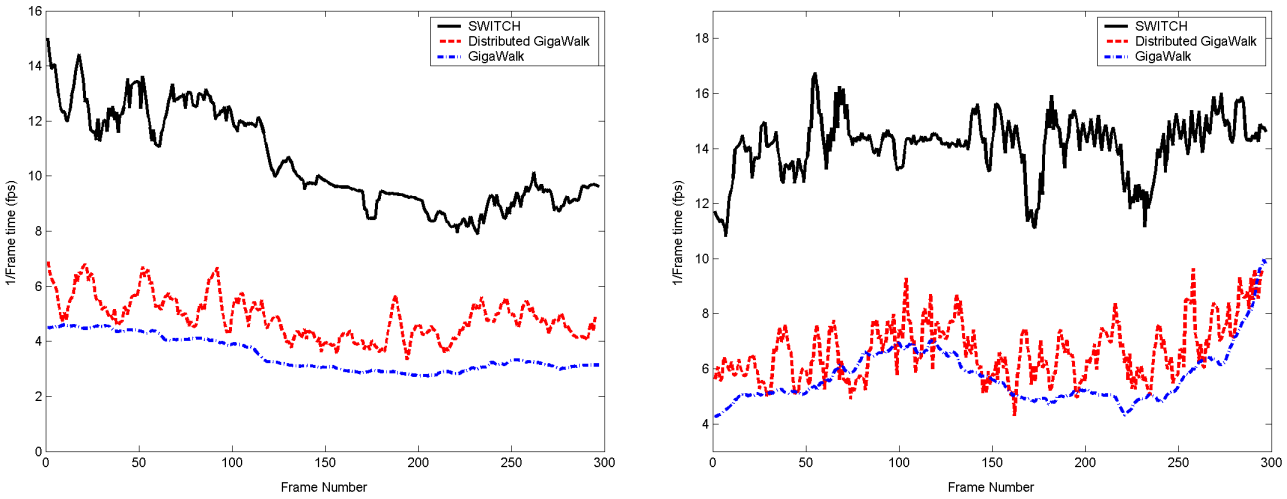
Our occlusion culling algorithm also assumes high spatial coherence between successive frames. If the camera position changes significantly from one frame to the next, the visible primitives from the previous frame may not be a good approximation to the occluder set for the current frame. As a result, the culling efficiency may not be high.

Our algorithm performs culling at an object level and does not check the visibility of each triangle. As a result, its performance can vary based on how the objects are defined and represented in the scene graph.

6 Summary and Future Work

We have presented a new occlusion culling algorithm based on occlusion-switches and used it to render massive models at interactive rates. The occlusion-switches, composed of two GPUs, make use of the hardware occlusion query that is becoming widely available on commodity GPUs. We have combined the algorithm with pre-computed levels-of-detail and highlighted its performance on three complex environments. We have observed $2-3$ times improvement in frame rate over earlier approaches. The culling performance of the algorithm is further improved by using a sub-object hierarchy and it is used for interactive shadow generation [Govindaraju et al. 2003].

Many avenues for future work lie ahead. A low latency network implementation is highly desirable to maximize the performance achieved by our parallel occlusion culling algorithm. One possibility is to use raw GM sockets over



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

(b) Powerplant model at 5 pixels of error

Figure 5: *Frame rate comparison between SWITCH, GigaWalk and Distributed GigaWalk at* 1024×1024 *screen resolution. We obtain* $2-3$ *times improvement in the frame rate as compared to Distributed GigaWalk and GigaWalk.*

Myrinet. We are also exploring the use of a reliable protocol over UDP/IP. Our current implementation loads the entire scene graph and object LODs on each PC. Due to limitations on the main memory, we would like to use out-of-core techniques that use a limited memory footprint. Moreover, the use of static LODs and HLODs can lead to popping artifacts as the rendering algorithm switches between different approximations. One possibility is to use view-dependent simplification techniques to alleviate these artifacts. Finally, we would like to apply our algorithm to other complex environments.

Acknowledgments

Our work was supported in part by ARO Contract DAAD19-99-1-0162, NSF award ACI 9876914, ONR Young Investigator Award (N00014-97-1-0631), a DOE ASCI grant, and by Intel Corporation.

The Double Eagle model is courtesy of Rob Lisle, Bryan Marz, and Jack Kanakaris at NNS. The Power Plant environment is courtesy of an anonymous donor. The Boeing 777 model is courtesy of Tom Currie, Bill McGarry, Marie Murray, Nik Prazak, and Ty Runnels at the Boeing Company. We would like to thank David McAllister, Carl Erikson, Brian Salomon and other members of UNC Walkthrough group for useful discussions and support.

References

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

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

Baxter, B., Sud, A., Govindaraju, N., and Manocha, D. 2002. Gigawalk: Interactive walkthrough of complex 3d environments. *Proc. of Eurographics Workshop on Rendering*.

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

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

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

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

Erikson, C., Manocha, D., and Baxter, B. 2001. Hlods for fast display of large static and dynmaic environments. *Proc. of ACM Symposium on Interactive 3D Graphics*.

Govindaraju, N., Lloyd, B., Yoon, S., Sud, A., and Manocha, D. 2003. Interactive shadow generation in complex environments. Tech. rep., Department of Computer Science, University of North Carolina.

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

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

Hillesl, K., Salomon, B., Lastra, A., and Manocha, D. 2002. Fast and simple occlusion culling using hardware-based depth queries. Tech. Rep. TR02-039, Department of Computer Science, University of North Carolina.

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

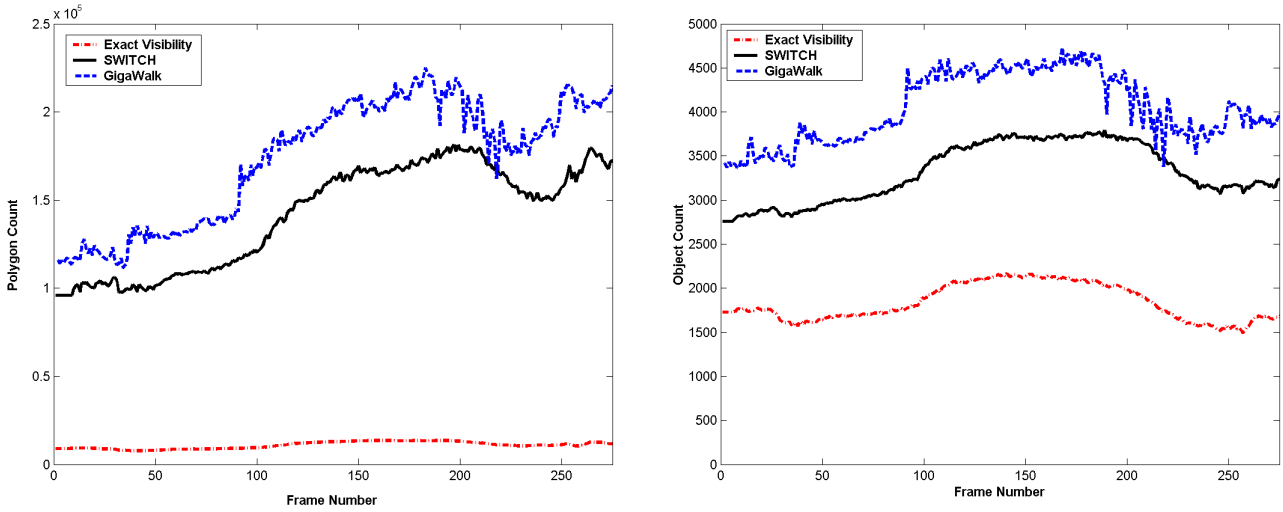
Humphreys, G., Eldridge, M., Buck, I., Stoll, G., Everett, M., and Hanrahan, P. 2001. Wiregl: A scalable graphics system for clusters. *Proc. of ACM SIGGRAPH*.

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

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

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

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



(a) At polygon level

(b) At object level

Figure 6: *Double Eagle Tanker: Comparison of exact visibility computation with SWITCH and GigaWalk at* 20 *pixels of error at* 1024×1024 *screen resolution. SWITCH is able to perform more culling as compared to GigaWalk. However, it renders one order of magnitude more triangles or twice the number of objects as compared to exact visibility.*

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

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

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

Schaufler, G., Dorsey, J., Decoret, X., and Sillion, F. 2000. Conservative volumetric visibility with occluder fusion. *Proc. of ACM SIGGRAPH*, 229–238.

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

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

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

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

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

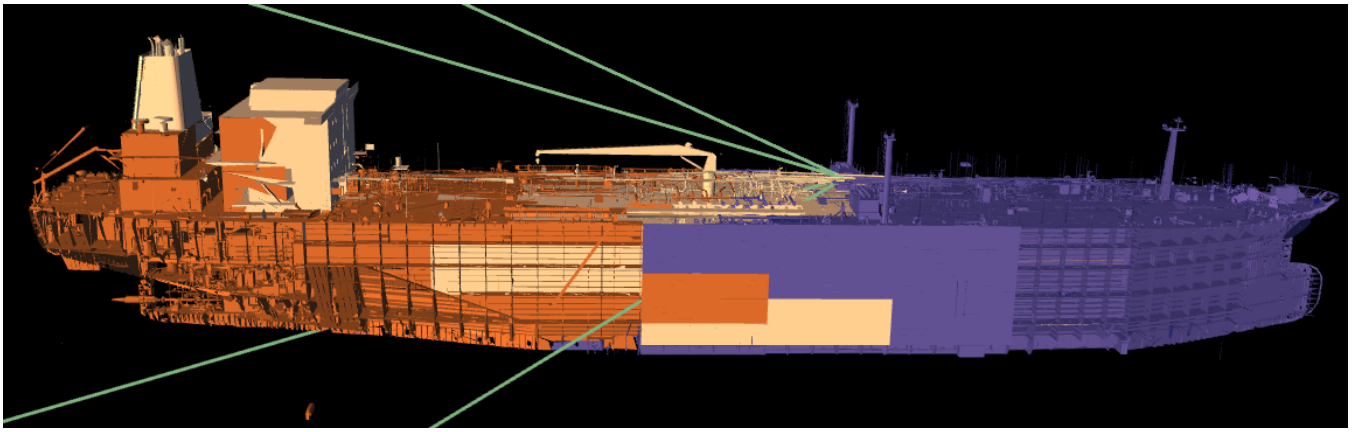
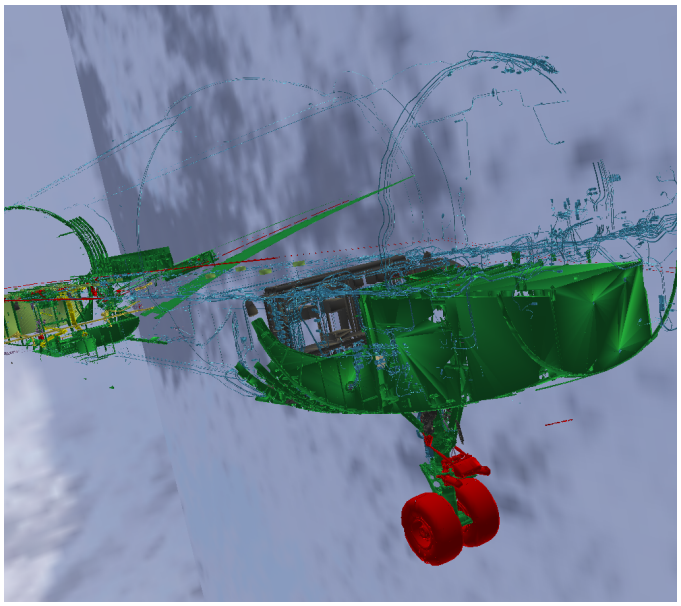
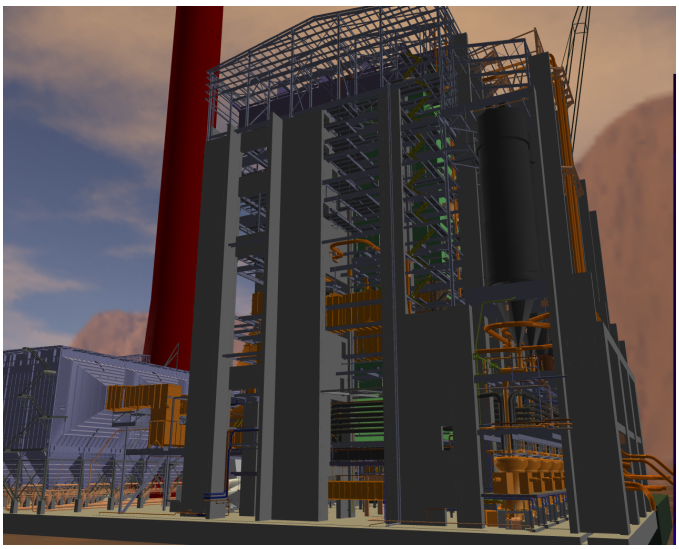


Figure 1: *Performance of occlusion-switch algorithm on the DoubleEagle Tanker model: This environment consists of more than* 82 *million triangles and our algorithm renders it a t* $9-15$ *fps on a cluster of 3 PCs, each consisting of an NVIDIA GeForce 4 GPU. Occlusion-switch culls away most occluded portions of the model and renders around* $200K$ *polygons in the view shown. Objects are rendered in following colors - visible: yellow; view-frustum culled: violet; and occlusion-culled: orange.*



(a) *Portion of a Boeing* 777 *model rendered at* 15 *pixels of error. Our system, SWITCH, is able to render it at* $11-18$ *frames per second on a 3-PC cluster.*



(b) *Powerplant model composed of more than* 12.7 *million triangles. SWITCH can render it at* $11-19$ *frames per second using* 5 *pixels of deviation error.*

Figure 2: *Performance of Occlusion-switch on complex CAD models: Both the models are rendered at* 1024×1024 *screen resolution using NVIDIA GeForce 4 cards.*