Thirteenth Eurographics Workshop on Rendering (2002)
P. Debevec and S. Gibson (Editors)

GigaWalk: Interactive Walkthrough of Complex Environments

William V. Baxter III Avneesh Sud Naga K. Govindaraju Dinesh Manocha

University of North Carolina at Chapel Hill
{baxter,sud,naga,dm}@cs.unc.edu
http://gamma.cs.unc.edu/GigaWalk

Abstract

We present a new parallel algorithm and a system, GigaWalk, for interactive walkthrough of complex, gigabyte-sized environments. Our approach combines occlusion culling and levels-of-detail and uses two graphics pipelines with one or more processors. GigaWalk uses a unified scene graph representation for multiple acceleration techniques, and performs spatial clustering of geometry, conservative occlusion culling, and load-balancing between graphics pipelines and processors. GigaWalk has been used to render CAD environments composed of tens of millions of polygons at interactive rates on systems consisting of two graphics pipelines. Overall, our system's combination of levels-of-detail and occlusion culling techniques results in significant improvements in frame-rate over view-frustum culling or either single technique alone.

Keywords: Interactive display systems, parallel rendering, occlusion culling, levels-of-detail, Engineering Visualization.

1. Introduction

Users of computer-aided design and virtual reality applications often create and use geometric models of large, complex 3D environments. Gigabyte-sized datasets representing power plants, ships, airplanes, submarines and urban scenes are not uncommon. Simulation-based design and design review of such datasets benefits significantly from the ability to generate user-steered interactive displays or *walkthroughs* of these environments. Yet, rendering these environments at interactive rates and with high fidelity has been a major challenge.

Many acceleration techniques for interactive display of complex datasets have been developed. These include visibility culling, object simplification and the use of image-based or sampled representations. They have been successfully combined to render certain specific types of datasets at interactive rates, including architectural models [15], terrain datasets [25], scanned models [33] and urban environments [42]. However, there has been less success in displaying more general complex datasets due to several challenges facing existing techniques:

Occlusion Culling: While possible for certain environments, performing exact visibility computations on large, general datasets is difficult to achieve in real time on current graphics systems [11]. Furthermore, occlusion culling alone will not sufficiently reduce the load on the graphics pipeline when many primitives are actually visible.

Object Simplification: Object simplification techniques alone have difficulty with high-depth-complexity scenes, as they do not address the problems of overdraw and fill load on the graphics pipeline.

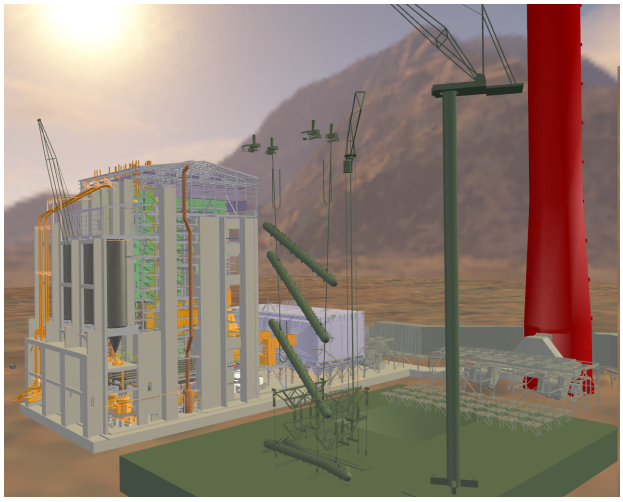


Figure 1: Coal-Fired Power plant: This 1.7 gigabyte environment consists of over 13 million triangles and 1200 objects. GigaWalk can display it 12-37 frames per second on an SGI Onyx workstation using two IR2 graphics pipelines and three 300MHz R12000 CPUs.

Image-based Representations: There are some promising image-based algorithms, but generating complete samplings of large complex environments automatically and efficiently remains a difficult problem. The use of image-based methods can also lead to popping and aliasing artifacts.

1.1. Main Results

We present a parallel architecture that enables interactive rendering of complex environments comprised of many tens of millions of polygons. Initially, we precompute geometric levels-of-detail (LODs) and represent the dataset using a scene graph. Then at runtime we compute a *potentially visible set* (PVS) of geometry for each frame using a combination of view frustum culling and a two-pass hierarchical Z-buffer occlusion culling algorithm [19] in conjunction with the pre-computed LODs. The system runs on two graphics rasterization pipelines and one or more CPU processors. Key features of our approach include:

1. A parallel rendering algorithm that is general and automatic, makes few assumptions about the model, and places no restrictions on user motion through the scene.
2. A unified scene graph hierarchy that is used for both geometric simplification and occlusion culling.
3. A parallel, image-precision occlusion culling algorithm based on the hierarchical Z-buffer [19, 20]. It uses *hierarchical occluders* and can perform conservative as well as approximate occlusion culling.
4. A parallel rendering algorithm that balances the computational load between two rendering pipelines and one or more processors.
5. An interactive system, GigaWalk, to render large, complex environments with good fidelity on two-pipeline graphics systems. The graphics pipelines themselves require only standard rasterization capabilities.

We demonstrate the performance of our system on two complex CAD environments: a coal-fired power plant (Fig. 1) composed of 13 million triangles, and a Double Eagle Tanker (Plate 1) composed of over 82 million triangles. GigaWalk is able to render models such as these at $11-50$ frames a second with little loss in image quality on an SGI Onyx workstation using two IR2 pipelines. The end-to-end latency of this implementation is typically $50-150$ milliseconds. We have also developed a preliminary implementation of GigaWalk on a pair of networked PCs.

1.2. Organization

The rest of the paper is organized as follows. We give a brief survey of previous work in Section 2. Section 3 gives an overview of our approach. In Section 4 we describe the scene representation and preprocessing steps. Section 5 presents the parallel algorithm for interactive display. We describe the system implementation and highlight its performance on complex models in Section 6.

2. Prior Work

In this section, we present a brief overview of previous research on interactive rendering of large datasets, including geometric simplification and occlusion culling algorithms, and other systems that have combined multiple rendering acceleration techniques.

2.1. Geometric Simplification

Simplification algorithms compute a reduced-polygon approximation of a model while attempting to retain the shape of the original. A recent survey of simplification algorithms is presented in [30].

Algorithms for simplifying large environments can be classified as either static (view-independent) or dynamic (view-dependent). Static approaches pre-compute a discrete series of levels-of-detail (LODs) in a view-independent manner [9, 17, 32, 35]. Erikson et al. [10] presented an approach to large model rendering based on the hierarchical use of static LODs, or HLODs. We also use LODs and HLODs in our system.

At run-time, rendering algorithms for static LODs choose an appropriate LOD to represent each object based on the viewpoint. Selecting the LODs requires little run-time computation, and rendering static LODs on contemporary graphics hardware is also efficient.

View-dependent, dynamic algorithms pre-compute a data structure that encodes a continuous range of detail. Examples include progressive meshes [23, 24, 44] and hierarchies of decimation operations [29, 12]. Selection of the appropriate LOD is based on view-parameters such as illumination and viewing position. Overall, view-dependent LODs can provide better fidelity than static LODs and work well for large connected datasets such as terrain and spatially large objects. However, the run-time overhead is higher compared to static LODs, since all level-of-detail selection is done at the individual feature level (vertex, edge, polygon), rather than the object level.

2.2. Occlusion Culling

Occlusion culling methods attempt to quickly determine a PVS for a viewpoint by excluding geometry that is occluded. A recent survey of different algorithms is presented in [6].

Several effective algorithms have been developed for specific environments. Examples include cells and portals for architectural models [2, 39] and algorithms for urban datasets or scenes with large, convex occluders [7, 22, 36, 41, 42]. In this section, we restrict the discussion to occlusion culling algorithms for general environments.

Algorithms for occlusion culling can be broadly classified based on whether they are conservative or approximate, whether they use object space or image space hierarchies, and whether they compute visibility from a point (*from-point*) or a region (*from-region*). Conservative algorithms compute a PVS that includes all the visible primitives, plus a small number of potentially occluded primitives [7, 19, 22, 28, 45]. The approximate algorithms identify most of the visible objects but may incorrectly cull some objects [5, 27, 45].

Object space algorithms can perform culling efficiently and accurately given a small set of large occluders, but it

is difficult to perform the “occluder fusion" necessary to effectively cull in scenes composed of many small occluders. For these types of scenes, the image space algorithms typified by the hierarchical Z-buffer (HZB) [19, 20] or hierarchical occlusion maps (HOM) [45] are more effective.

From-region algorithms pre-compute a PVS for each region of space to reduce the run-time overhead [8, 36, 41]. This works well for scenes with large occluders, but the amount of geometry culled by a given occluder diminishes as the region sizes are increased. Thus there is a trade-off between the quality of the PVS estimation for each region and the memory overhead. These algorithms may be overly conservative and have difficulty obtaining significant culling in scenes including only small occluders. In contrast, from-point algorithms generally provide more accurate culling, but they have a higher run-time cost.

2.3. Parallel Approaches

A number of parallel approaches based on multiple graphics pipelines have been proposed. These can provide scalable rendering on shared-memory systems or clusters of PCs. These approaches can by classified mainly as either object-parallel, screen-space-parallel, or frame-parallel [21, 37]. Specific examples include distributing primitives to different pipelines by the screen region into which they fall (screen-space-parallel), or rendering only every Nth frame on each pipeline (frame-parallel).

Another parallel approach to large model rendering that shows promise is interactive ray tracing [4, 40]. The algorithm described in [40] is able to render the Power Plant model at 4-5 frames a second with 640×480 pixel resolution on a cluster of seven dual processor PCs.

Garlick et al. [16] presented a system for performing view-frustum culling on multiple CPUs in parallel with the rendering process. Their observation that culling can be performed in parallel to improve overall system performance is the fundamental concept behind our approach as well.

Wonka et al. [42] presented a “visibility server” that performed occlusion culling to compute a PVS at run-time in parallel on a separate machine. Their system works well for urban environments; however, it relies on the *occluder shrinking* algorithm [41] to compute the region-based visibility. This approach is effective only if the occluders are large and volumetric in nature.

2.4. Hybrid Approaches

The literature reports several systems that combine multiple techniques to accelerate the rendering of large models. For example, The BRUSH system [34] used LODs with hierarchical representation for large mechanical and architectural models. The UC Berkeley Architecture Walkthrough system [15] combined hierarchical algorithms with object-space visibility computations [39] and LODs for architectural models.

More recently, Anjudar et al. [3] presented a framework that integrates occlusion culling and LODs. The crux of the approach is to estimate the degree of visibility of each object in the PVS and use that value both to select appropriate LODs and to cull. The method relies on decomposing scene objects into overlapping convex pieces (axis-aligned boxes) that then serve as individual “synthetic occluders”. Thus the effective maximum occluder size depends on the largest axis-aligned box that will fit inside each object.

Another recent integrated approach uses a prioritized-layered projection visibility approximation algorithm with view-dependent rendering [11]. The resulting rendering algorithm seems a promising approach when approximate (non-conservative) visibility is acceptable.

The UNC Massive Model Rendering (MMR) system [1] combined LODs with image-based impostors and occlusion culling to deliver interactive walkthroughs of complex models. A more detailed comparison with this system will be made later in Section 6.5.

Various proprietary systems exist as well, such as the one Boeing created in the 1990’s to visualize models of large passenger jets. However, to the best of our knowledge, no detailed descriptions of this system are available, so it is difficult to make comparisons.

3. Overview

In this section, we give a brief overview of the main components of our approach. These components are simplification, occlusion culling, and a parallel architecture.

3.1. Model Simplification

Given a large environment, we generate a scene graph by clustering small objects, and partitioning large objects to create a spatially coherent, axis-aligned bounding box (AABB) hierarchy. The hierarchy construction will be discussed in more detail in Section 4.

3.2. Parallel Occlusion Culling

At run-time, our algorithm performs occlusion culling, in addition to view frustum culling, based on the pre-computed AABB scene graph hierarchy. We use a two-pass version of the hierarchical Z-buffer algorithm [19] with a two-graphics-pipeline parallel architecture. In this architecture, occluders are rendered on one pipeline while the final interactive rendering of visible primitives takes place on the second pipeline. A separate software thread performs the actual culling using the Z-buffer that results from the occluder rendering. The architecture will be presented in detail in Section 5.

We chose to use the hierarchical Z-buffer (HZB) because of its good culling performance, minimal restrictions on the

type of occluders, and for its ability to perform occluder fusion. Moreover, it can be made to work well without extra preprocessing or storage overhead by exploiting temporal coherence. The preprocessing and storage cost of GigaWalk is thus the same as that of an LOD-only system.

Occluder Selection: A key component of any occlusion culling algorithm is occluder selection, which can be accomplished in a number of ways. A typical approach uses solid angles and spatial distributions of objects to estimate a small set of good occluders [45, 27]. However, occluders selected according to such heuristics are not necessarily optimal in terms of the number of other objects they actually occlude. The likelihood of obtaining good occlusion can be increased by making the occluder set larger, but computational costs usually demand the set be as small as possible.

Our parallel approach, on the other hand, allows us to take advantage of the temporal coherence based occluder selection algorithm presented by Greene et al.[19], which treats *all* the visible geometry from the previous frame as occluders for the current frame. This method makes use of frame-to-frame coherence and provides a good approximation to the foreground occluders for the current frame.

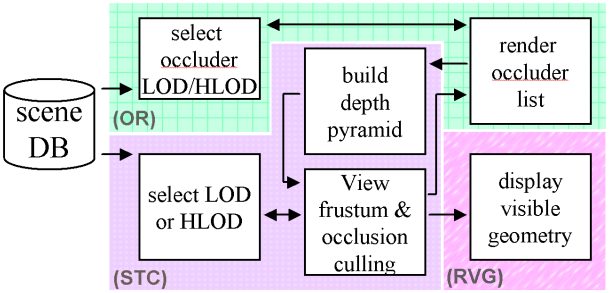


Figure 2: System Architecture: Each shaded region represents a separate process. The OR and RVG processes are associated with separate graphics pipelines, whereas the STC uses one or more processors.

3.3. GigaWalk Architecture

Fig. 2 presents the overall architecture of our run-time system. It shows the three processes that run in parallel:

1. **Occluder Rendering (OR):** Using all the visible geometry from a previous frame as the occluder set, this process renders that set into a depth buffer. It runs on the first graphics pipeline.
2. **Scene Traversal, Culling and LOD Selection (STC):** This process computes the HZB using the depth buffer computed by OR. It traverses the scene graph, computes the visible geometry and selects appropriate LODs based on the user-specified error tolerance. The visible geometry is used by RVG for the current frame and OR for the next frame. It runs on one or more processors.
3. **Rendering Visible Scene Geometry (RVG):** This process renders the visible scene geometry computed by STC. It uses the second graphics pipeline.

More details of the run-time system are given in Sections 5 and 6.

4. Scene Representation

In this section, we give an overview of our pre-processing algorithm used to compute a scene graph representation of the geometric environment.

CAD datasets often consist of a large number of objects which are organized according to a functional, rather than spatial, hierarchy. By "object" we mean simply the lowest level of organization in a model or model data structure above the primitive level. The size of objects can vary dramatically in CAD datasets. For example, in the Power Plant model a large pipe structure, which spans the entire model and consists of more than 6 million polygons, is one object. Similarly, a relatively small bolt with 20 polygons is another object. Our rendering algorithm computes LODs, selects them, and performs occlusion culling at the object level; therefore, the criteria used for organizing primitives into objects has a serious impact on the performance of the system. Our first step, then, is to redefine objects in a dataset based on criteria that will improve performance.

4.1. Unified Scene Hierarchy

Our rendering algorithm performs occlusion culling in two rendering passes: Pass 1 renders occluders to create a hierarchical Z-buffer to use for culling, Pass 2 renders the objects that are deemed visible by the HZB culling test. Given this two-pass approach, we could consider using a separate representation for occluders in Pass 1 than for displayed objects in Pass 2 [22, 45]. Using different representations has the advantage of allowing different criteria for partitioning and clustering of each hierarchy. Moreover, it gives us the flexibility to use an alternate error metric for creating simplified occluders, one optimized to preserve occlusion properties rather than visual fidelity.

Despite these potential advantages, we use a single, unified hierarchy for occlusion culling and LOD-based rendering. A single hierarchy offers several benefits. First, using the same representation decreases the storage overhead and the overall preprocessing cost. Second, it leads to a conservative occlusion culling algorithm. Our rendering algorithm treats the visible geometry from the previous frame as the occluder set for the current frame. In order to guarantee conservative occlusion culling, it is sufficient to ensure that exactly the same set of nodes and LODs in the unified scene graph are used by each process.

4.1.1. Criteria for Hierarchy

A good hierarchical representation of the scene graph is crucial for the performance of occlusion culling and the over-

all rendering algorithm. We use the same hierarchy for view frustum culling, occluder selection, occlusion tests on potential occludees, hierarchical simplification, and LOD selection. Though there has been considerable work on spatial partitioning and bounding volume hierarchies, including top-down and bottom-up strategies and spatial clustering, none of them seem to have addressed all the characteristics desired by our rendering algorithm. These include good spatial localization, object size, balance of the hierarchy, and minimal overlap between the bounding boxes of sibling nodes in the tree.

Bottom-up hierarchies lead to better localization and higher fidelity LODs. However, it is harder to use bottom-up techniques to compute hierarchies that are both balanced and have minimal spatial overlap between nodes. On the other hand, top-down schemes are better at ensuring balanced hierarchies and bounding boxes with little or no overlap between sibling nodes. Given their respective benefits, we use a hybrid approach that combines both top-down partitioning and hierarchy construction with bottom-up clustering.

4.2. Hierarchy Generation

In order to generate uniformly-sized objects, our preprocessing algorithm first redefines the objects using a combination of partitioning and clustering algorithms. The partitioning algorithm takes large objects and splits them into multiple objects. The clustering step groups objects with low polygon counts based on their spatial proximity. The combination of these steps seems to result in a redistribution of geometry with good localization and emulates some of the benefits of pure bottom-up hierarchy generation. The overall algorithm proceeds as follows:

1. Partition large objects into sub-objects in the initial database (top-down)
2. Organize disjoint objects and sub-objects into clusters (bottom-up)
3. Partition again to eliminate any uneven spatial clusters (top-down)
4. Compute an AABB bounding volume hierarchy on the final redefined set of objects (top-down).

The partitioning (stages 2 and 4) uses standard top-down techniques that group polygons based on an object's center or center-of-mass, along with several heuristics for selecting the split axis. The clustering algorithm (stage 3) was adapted from a computer vision technique for image segmentation [14]. The algorithm uses minimum spanning trees (MST) to represent clusters and is similar to *Kruskal's* algorithm [26]. Plate 2 shows the results of clustering and partitioning on the Power Plant and Double Eagle models. More details on the partitioning and clustering algorithm as well as hierarchy computation are given in [38].

4.3. HLOD Generation

Given the AABB-based scene graph representation, the algorithm computes a series of LODs for each node. The HLODs are computed in a bottom-up manner. The HLODs of the leaf nodes are the same as static LODs, while the HLODs of intermediate nodes are computed by combining the LODs of the nodes with the HLODs of node's children[10]. We use the GAPS [9] simplification algorithm, which can merge disjoint objects.

The majority of the pre-computation time is spent in LOD and HLOD generation. The HLODs of an internal node depend only on the LODs of the children, so by keeping only the LODs of the current node and its children in main memory, HLOD generation is accomplished within a small memory footprint. Specifically, the memory usage is given by

$$\begin{aligned} main\_memory\_footprint \;\leq\; & sizeof(AABBHierarchy) \\ & + \max_{N_i \in SG} (sizeof(N_i) + \\ & \sum_{C_j \in Child(N_i)} sizeof(C_j)), \end{aligned}$$

where SG corresponds to the scene graph.

4.4. HLODs as Hierarchical Occluders

Our occlusion culling algorithm uses LODs and HLODs of nodes as occluders to compute the HZB. They are selected based on the maximum screen-space pixel deviation error on object silhouettes.

The HLODs used by the rendering algorithm can also be regarded as "hierarchical occluders". A hierarchical occluder associated with a node N_i is an approximation of a group of occluders contained in the subtree rooted at N_i. The approximation provides a lower polygon count representation of a collection of object-space occluders. It can also be regarded as object-space occluder fusion.

5. Interactive Display

In this section, we present our parallel rendering architecture for interactive display of complex environments. Here we describe in detail the operations performed by each of the two graphics pipelines and each of the three processes: occluder rendering (OR), scene traversal and culling (STC) and rendering visible geometry (RVG), which run synchronously in parallel (as shown in Fig. 3).

5.1. Run-time Architecture

The relationship between different processes and the tasks performed by them is shown in Fig. 2.

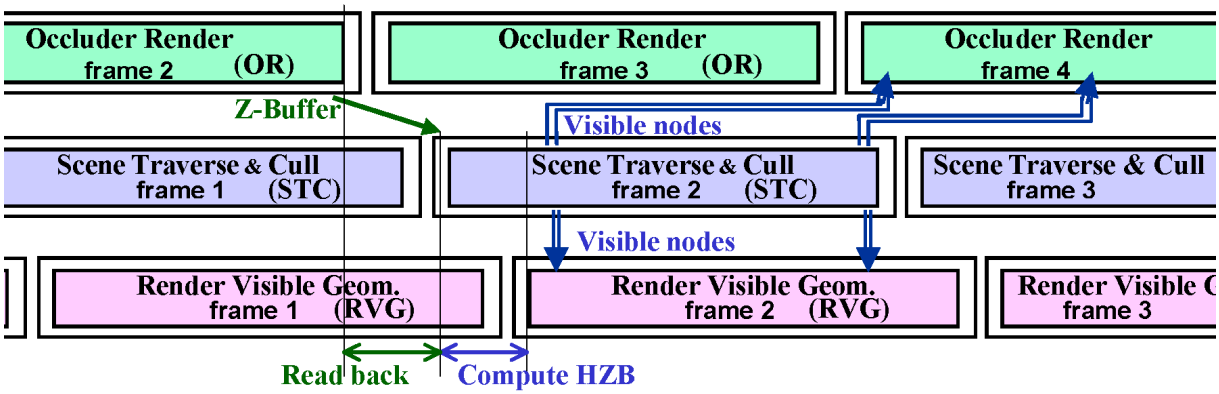


Figure 3: Timing relationship between different processes. The arrows indicate data passed between processes during the computation of frame 2. Along with the other data indicated, viewpoints also travel through the pipeline according to the frame numbers. This diagram demonstrates the use of occluders from frame $i-2$ rather than $i-1$ (see Section 5.2).

5.1.1. Occluder Rendering

The first stage for a given frame is to render the occluders. The occluders are simply the visible geometry from a previous frame. By using this temporal coherence strategy, the load on the two graphics pipelines is essentially balanced, since they render the exact same set of primitives, just shifted in time. The culling and LOD selection performed for displaying frame i results in an occluder set for frame $i+1$ that has manageable size. A brief pseudo-code description is given in Algorithm 5.1.

Occluder_Render(δ,frame *i*)
- get current instantaneous camera position (camera *i*)
- while (more nodes on node queue from STC (*i*-1))
  - pop next node off the queue
  - select LOD/HLOD for the node according to error tolerance, δ, using camera *i*
  - render that LOD/HLOD into Z-buffer *i*
- read back Z-buffer *i* from graphics hardware
- push Z-buffer *i* onto queue for STC *i*
- push camera *i* onto queue for STC *i*

ALGORITHM 5.1: Occluder_Render.

Since the list of visible geometry for rendering comes from the culling stage (STC), and STC gets its input from this process (OR), a start-up procedure is required to initialize the pipeline and resolve this cyclic dependency. During startup, the OR stage is bypassed on the first frame, and STC generates its initial list of visible geometry without occlusion culling.

5.1.2. Scene Traversal, Culling and LOD Selection

The STC process first computes the HZB from the depth buffer output from OR. It then traverses the scene graph, performing view-frustum culling, occlusion culling and LOD error-based selection in a recursive manner. The LOD selection proceeds exactly as in [10, 43]: recursion terminates at nodes that are either culled, or which meet the user-specified pixel-error tolerance. A pseudo-code description is given in Algorithm 5.2. The occlusion culling is performed by comparing the bounding box of the node with the HZB. It can be performed in software or can make use of hardware-based queries as more culling extensions become available.

Scene_Traversal_Cull(ε, frame *i*)
- get Z-buffer *i* from OR *i* via Z-buffer queue
- build HZB *i*
- get camera *i* from OR *i* queue
- push copy of camera *i* onto queue for RVG *i*
- set NodeList[*i*] = Root(SceneGraph)
- while (NotEmpty(NodeList[*i*]))
  - node = First(NodeList[*i*])
  - set NodeList[*i*] = Delete(NodeList[*i*],node)
  - if (View_Frustum_culled(node)) then next;
  - if (Occlusion_Culled(node)) then next;
  - if HLOD_Error_Acceptable(ε,node) then
    - push node onto queue for OR $i+1$;
    - push node onto display queue for RVG *i*;
  - else
    set NodeList[*i*] = Add(NodeList[*i*], Children(node));

ALGORITHM 5.2: Scene_Traversal_Cull.

5.1.3. Rendering Visible Scene Geometry

All the culling is performed by STC, so the final render loop has only to rasterize the nodes from STC as they are placed in the queue. See Algorithm 5.3.

5.2. Occluder Selection

Ideally, the algorithm uses the visible geometry from the previous frame ($i-1$) as the occluders for the current frame to get the best approximation to the current foreground geometry. However, using the previous frame's geometry can lead

Render_Visible_Scene_Geometry(frame *i*)
- get camera *i* from STC *i* queue
- while (more nodes on queue from STC *i*)
  - pop node off queue
  - render node

ALGORITHM 5.3: Render_Visible_Scene_Geometry.

to bubbles in the pipeline, because of the dependency between the OR and STC stages: STC must wait for OR to finish rendering the occluders before it can begin traversing the scene graph and culling. Fortunately, using the visible geometry from two frames previous can eliminate that dependency, and still provides a good approximation to the visible geometry for most interactive applications.

5.3. Trading Fidelity for Performance

The user can trade off fidelity for better performance in a number of ways. The primary control for achieving higher frame rates is the allowable LOD pixel error (see Plate 3).

Our system has been designed primarily to offer conservative occlusion culling, and we report all of our results based on this mode of operation. Our system can guarantee conservative culling results for two reasons: 1) the underlying HZB algorithm used is itself conservative, and 2) for a given frame i we choose the exact same set of LODs for both OR and STC stages. By choosing the same LODs, we ensure that the Z-buffer used for culling is consistent with the geometry it is used to cull. Without this selection algorithm, conservativity is not guaranteed.

We have also modified our run-time pipeline in a number of ways to optionally increase frame rate or decrease latency, by allowing the user to relax the restriction that occlusion culling be performed conservatively:

• **Asynchronous rendering pipeline:** Rather than waiting for the next list of visible geometry from the culling stage (STC) to render frame $i+1$, the render stage (RVG) can instead proceed to render another frame, still using the geometry from frame i, but using the most recent camera position, corresponding to the user's most up-to-date position. This modification eliminates the extra frame of latency introduced by our method. The main drawback is that it may introduce occlusion errors that, while typically brief, are potentially unbounded when the user moves drastically.

• **Nth Farthest Z Buffer Values:** The occlusion culling can be modified to use not the farthest Z values in building the depth pyramid, but the Nth farthest [20], thereby allowing for approximate "aggressive" culling.

• **Lower HZB resolution for occluder rendering:** The pixel resolution of the OR stage can be set smaller than that of the RVG stage. If readback from the depth buffer or HZB computation is relatively slow, this can improve the performance. However, using a lower resolution source for HZB allows for the possibility of depth buffer aliasing artifacts that can manifest themselves as small occlusion errors. In practice, however, we have not been able to visually detect any such errors when using OR depth buffers with as little as half the RVG resolution.

6. Implementation and Performance

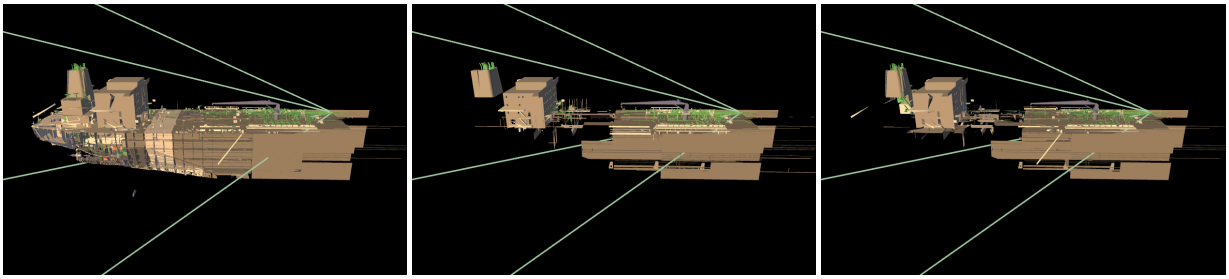
We have implemented our parallel rendering algorithm on two hardware systems. The first is a shared-memory multiprocessor machine with dual graphics rasterization pipelines: an SGI Onyx workstation with 300MHz R12000 MIPS processors, Infinite Reality (IR2e) graphics boards, and 16GB of main memory. Our algorithm uses three CPUs and two graphics pipelines of this machine. We have also made a preliminary port of the system to a pair of networked, dual processor PCs: both are Dell Precision Workstations with GeForce 4 graphics cards, 2GHz processors, and 2GB of main memory.

All of the inter-process communication is implemented using a templated producer-consumer queue data structure. For the SGI implementation, this uses shared memory to pass data between processes. On the PC, the queue class was re-implemented to pass data over TCP/IP sockets. Each stage (OR,STC,RVG) is connected with the others using one or more instances of this queue data structure. Synchronization between the processes is accomplished by pushing sentinel nodes onto the shared queues to delimit the data at the end of a frame. The scene graph resides in shared memory in the SGI version, and is simply replicated on both PCs for the PC version. The overall run-time system is about $6,000$ lines of C++ code.

We have tested the performance of GigaWalk on two complex environments, a coal-fired Power Plant (shown in Fig. 1) and a Double Eagle Tanker (shown in Plate 1). The details about these environments are shown in Table 1. In addition to the model complexity, the table also lists the object counts after the clustering and partitioning steps. Unless otherwise noted, performance results from this point on will refer to the SGI implementation.

6.1. Improvement in Frame Rate

GigaWalk is able to render our two example complex environments at interactive rates from most viewpoints. The frame rate varies from 11 to 50 FPS. It is more than 20 frames a second from most viewpoints in the scene. We have recorded and analyzed some example paths through these models, as shown on the video available at the WWW site: *http://gamma.cs.unc.edu/GigaWalk*. In Fig. 5, we show the improvement in frame rate for each environment. The graphs compare the frame rate for each individual rendering acceleration technique alone and for the combination. Table 2 shows the average speed-ups obtained by each technique over the same path. The comparison between the techniques for a given viewpoint is shown in Fig. 4.



(a) Polygon Count = 202666 (b) Polygon Count = 3578485 (c) Polygon Count = 61771

Figure 4: Comparison between different acceleration techniques from the same viewpoint. (a) Rendered with only HLODs. (b) Rendered with only HZB occlusion culling. (c) Rendered with GigaWalk using HLODs and HZB occlusion culling.

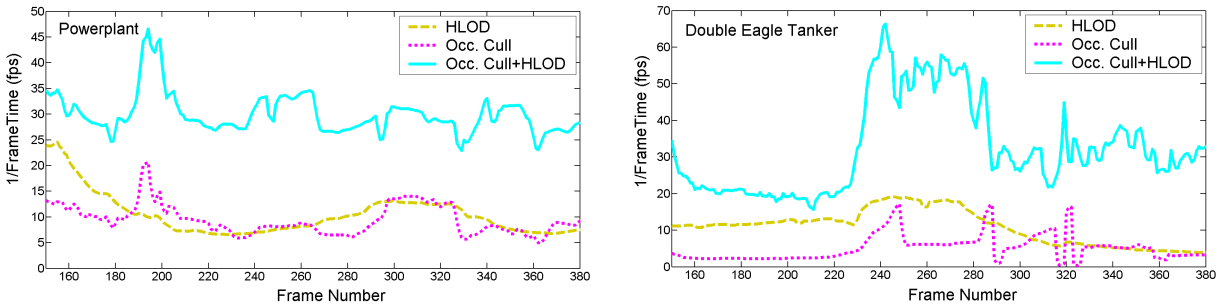


Figure 5: Comparison of acceleration techniques on a path in the Power Plant model at 10 pixels of error (left) and Double Eagle at 30 pixels of error (right), on the SGI Workstation using two IR pipelines. The Y-axis shows the instantaneous frame rate. The combination of HLODs + occlusion culling results in 2-5 times improvement over using only one of them. Display resolution was 640×480.

The networked PC implementation achieved an average frame rate of 10 frames per second on the Powerplant model at 1024×1024 resolution with at most 10 pixels of screen-space error, and about 11.5 frames per second on the Double Eagle tanker model rendered at 1024×1024 resolution with at most 20 pixels of screen-space error. It compares favorably with the shared memory implementation, but has much higher variance. This increase in variance is due to latency incurred from TCP/IP network buffering.

6.2. Culling Performance

Figure 6 shows the number of objects and polygons rendered for each frame on a path through the Power Plant and a path through the Double Eagle. It is clear from the left graphs that most of the reduction in object count comes from occlusion culling. The differences between the exact visibility counts and GigaWalk's are explained by GigaWalk's HZB occlusion algorithm, which culls based on objects' axis-aligned screen-space bounding rectangles rather than actual object polygons. On average for these paths, GigaWalk draws about twice many objects as a perfect object-level visibility algorithm, and about ten times as many polygons as a perfect polygon-level visibility algorithm.

	Object Count				
Env	**Poly** $\times 10^6$	**Init** $\times 10^4$	**Part**^{1} $\times 10^4$	**Clust** $\times 10^3$	**Part**^{2} $\times 10^5$
PP	12.2	0.12	6.95	3.33	0.38
DE	82.4	12.7	2.21	2.31	1.2

Table 1: A breakdown of the complexity of each environment. **Poly** is the polygon count. **Init** is the number of objects in the original dataset. The algorithm first partitions (**Part**^{1}) objects into sub-objects, then generates clusters (**Clust**), and finally partitions large uneven spatial clusters **Part**^{2}. The table shows the object count after each step.

6.3. System Latency

Our algorithm introduces a frame of latency to rendering times. Latency can be a serious issue for many interactive applications like augmented reality. Our approach is best suited for latency-tolerant applications, namely walkthroughs of large synthetic environments on desktop or projection displays. The end-to-end latency in the shared-memory imple-

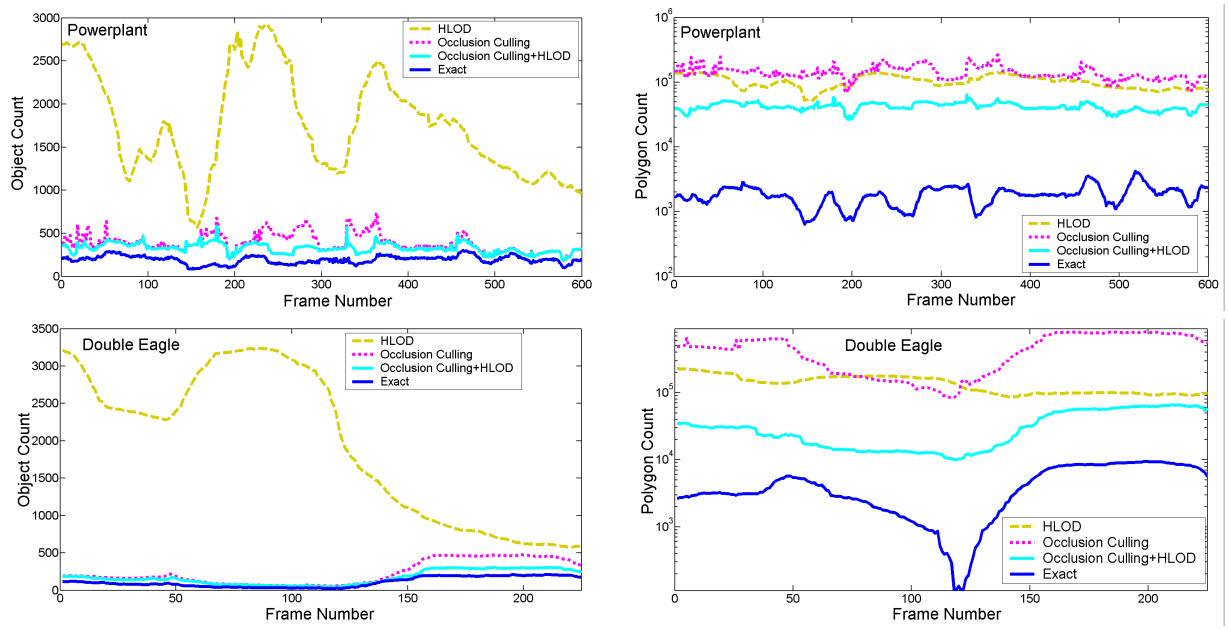


Figure 6: Comparison of object counts (left column) and polygon counts (right column) for different acceleration techniques. Top row is a path on the Powerplant model at 10 pixels of error. Bottom row is a path on the Double Eagle model at 30 pixels of error. The Y-axis shows number of objects or polygons drawn. "Exact" indicates what would be drawn by a perfect visibility algorithm using HLODs. Display resolution was 640x480.

Model	Average FPS			
	OCH	HLOD+VFC	OCC+VFC	VFC
PP	30.67	9.55	9.48	1.15
DE	29.43	9.76	3.27	0.02

Table 2: Average frame rates obtained by different acceleration techniques over the sample path. **FPS** = Frames Per Second, **HLOD** = Hierarchical levels of detail, **OCH** = Occlusion culling with HLOD, **OCC** = Occlusion Culling, **VFC** = View Frustum Culling

mentation varies with the frame rate. It is typically in the range 50 – 150 ms. The high end of this range is achieved when the frame rate dips close to 10 frames a second. This latency is within the range that most users can easily adapt to (less than 300 ms) without changing their interaction mode with the model [13].

In many interactive applications, the dominant component of latency is the frame rendering time [13]. Through the use of our two-pass occlusion culling technique, our rendering algorithm improves the frame rate by a factor of 3-4. As a result, the overall system latency is decreased, in contrast to an algorithm that does not use occlusion culling.

6.4. Preprocessing

This section reports the amount of time and memory used by our preprocessing.

6.4.1. Time and Space Requirements

The preprocessing was done on a single-processor 2GHz Pentium 4 PC with 2GB RAM. The preprocessing times for the Double Eagle model were: 177 min for hierarchy generation (partitioning/clustering), and 32.5 hours for out of core HLOD generation. The size of the final HLOD scene graph representation is 7.6GB which is less than 2 times the original data size. The AABB hierarchy skeleton occupies 7MB of space, though this could easily be further reduced.

The main memory requirement for partitioning and clustering is bounded by the size of the largest object/cluster. For the Double Eagle it was less than 200MB for partitioning, 1GB for clustering and 300MB for out of core HLOD generation.

6.5. Comparison with Earlier Approaches

A number of algorithms and systems have been proposed for interactive display of complex environments. These include specialized approaches for architectural, terrain and urban environments, as highlighted in Section 2. Given low depth complexity scenes, or scenes composed of large or convex occluders (e.g, architectural or urban models), our general

approach is not likely to perform any better than special-purpose algorithms designed specifically to exploit such features.

Of the previous systems which do handle general environments, however, few have been able to reduce both depth complexity (e.g. by using occlusion culling) and screen-space complexity (e.g. by using LODs). It is worth making the comparison with one previous system which was designed to do both, the MMR[1].

The MMR system combined LODs and occlusion culling for near-field geometry with image-based textured meshes to approximate the far-field, in a cell-based framework. While the combination of techniques proved capable of achieving interactive frame rates, the system had some drawbacks. First, the creation of cells required user intervention. For instance, some hand-selected model-dependent features were used in the Powerplant to define viewpoint cells. In contrast, the preprocessing and scene graph computation in GigaWalk is fully automatic. Second, the image-based far-field representations used in the MMR system resulted in dramatic popping and distortion when switching between different cells. Third, the memory overhead and preprocessing cost of creating six meshes and textures per cell was quite quite high. Finally, since the MMR used just a single rendering pipeline, it could only afford to use a few objects as occluders, rather than all the visible objects from the previous frame. The occluders had to be pre-selected offline using a heuristic which could not always find good candidates.

In MMR's favor, however, the cell based spatial decomposition allowed for a simple out-of-core prefetching and rendering algorithm. In contrast, GigaWalk currently assumes that the entire scene graph and the LODs and HLODs are loaded into main memory.

7. Conclusions and Lessons Learned

We have presented an approach to rendering interactive walkthroughs of complex 3D environments. The algorithm features a novel integration of conservative occlusion culling and levels-of-detail using a parallel algorithm. We have demonstrated a new parallel rendering architecture that integrates these acceleration techniques on two graphics pipelines, and highlighted its performance on two complex CAD environments. To the best of our knowledge, GigaWalk is the first system that can render such complex environments at interactive rates with this level of fidelity.

There are many complex issues with respect to the design and performance of systems for interactive display of complex environments. These include load balancing, extent of parallelism and scalability of the resulting approach, the effectiveness of occlusion culling and issues related to loading and managing large datasets.

7.1. Load Balancing

There is a trade-off between the depth of scene graph, which is controlled by the choice of minimum cluster size, and the culling efficiency. Smaller bounding boxes lead to better culling since more boxes can be rejected, so less geometry is sent to RVG. On the other hand, more boxes increases the cost of scene graph traversal and culling in STC. In our system, scene traversal and object culling (STC) operate in parallel with rendering (OR and RVG). If our performance bottleneck is the rendering processes (RVG and OR), we can shift the load back to the culling (STC) process by creating a finer partitioning. Conversely, we can use a coarser partitioning to move the load back to RVG and OR. Thus, the system can achieve load balancing between different processes running on the CPUs or graphics pipelines by changing the granularity of partitioning.

7.2. Parallelism

Parallel graphics hardware is increasingly being used to improve the rendering performance of walkthrough systems. Generally, though, the speed-up obtained from using N pipelines is no more than a factor of N. Using a second pipeline for occlusion culling (i.e. $N = 2$), however, enables GigaWalk to achieve more than two times speed-up for scenes with high depth complexity. For low depth complexity scenes there is little or no speed-up, but there is no loss in frame rate as the occlusion culling is performed using a separate pipeline. However, our parallel algorithm does introduce a frame of latency.

Note also that other parallel approaches [21, 37] are fundamentally orthogonal to our approach, and could potentially be used in conjunction with our architecture as black-box replacements for the OR and RVG rendering pipelines.

7.3. Load Times

One of the considerations in developing a walkthrough system to render gigabyte datasets is the time taken to load gigabytes of data from secondary storage, which can be many hours. To speed up the system we have implemented an on-demand loading system. Initially the system takes a few seconds to load the skeletal representation of the scene graph with just bounding boxes. Once loaded, the user commences the walkthrough while a fourth, asynchronous background process automatically loads the geometry for the nodes in the scene graph that are visible. We have found that adding such a feature is very useful in terms of system development and testing its performance on new complex environments.

8. Limitations and Future Work

Our current implementation of GigaWalk has many limitations. The current system works only for static environments, and it would be desirable to extend it to dynamic environments as well, perhaps with a strategy similar to that proposed in Erikson et al.[10].

The memory overhead of GigaWalk can be high. In the current implementation, viewing an entire model requires loading the scene graph and HLODs. Vardahan and Manocha[43] have recently developed an out-of-core algorithm that renders massive datasets using view-frustum culling and LOD/HLOD based selection which we may be able to benefit from.

The preprocessing time for our largest dataset, the Double Eagle, was also higher than desired. Since most nodes in the scene graph are non-overlapping, the LODs can be generated independently. Thus the algorithm could compute the LODs and HLODs in parallel, using multiple threads. This could improve the preprocessing performance considerably, reducing the 32.5 hours spent on the Double Eagle to a few hours.

The algorithm described in this paper guarantees image quality in terms of a bound on screen-space LOD error. First, we recognize that this is far from an ideal image-quality metric, and better metrics which are suitable for interactive display are desired. Second, our system gives no guarantees on the frame rate. The current system would be improved by the addition of a target-frame-rate rendering mode. Furthermore, the current system's use of static LODs and HLODs leads to some popping when switching between different levels. We would like to explore view-dependent or hybrid view-dependent/static LOD-based simplification approaches that can improve the fidelity of our geometric approximations without increasing the polygon count.

Finally, while the current PC implementation shows promise, we need to lower the networking latency in the system. Our implementation indicates that the bandwidth is sufficient with commodity TCP/IP over Ethernet, but to reduce latency it may be necessary to move to a lightweight protocol like UDP or even use specialized low-latency network hardware like Myrinet. We are also interested in using new hardware occlusion culling extensions on PC graphics cards to accelerate GigaWalk. Govindaraju et al.[18] recently devised one approach which uses three PCs and three GPUs.

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. We would like to thank Carl Erikson, Brian Salomon and other members of UNC Walkthrough group for their useful discussions and support. Special thanks to Sungeui Yoon for porting GigaWalk to the PC, and to Dorian Miller for help measuring end-to-end latencies with his latency meter [31].

References

1. D. Aliaga, J. Cohen, A. Wilson, H. Zhang, C. Erikson, K. Hoff, T. Hudson, W. Stuerzlinger, E. Baker, R. Bastos, M. Whitton, F. Brooks, and D. Manocha. MMR: An integrated massive model rendering system using geometric and image-based acceleration. In *Proc. of ACM Symposium on Interactive 3D Graphics*, 1999. 3, 10
2. 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. 2
3. C. Andujar, C. Saona-Vazquez, I. Navazo, and P. Brunet. Integrating occlusion culling and levels of detail through hardly-visibly sets. In *Proceedings of Eurographics*, 2000. 3
4. J. Alex and S. Teller. Immediate-mode ray-casting. Technical report, MIT LCS Technical Report 784, 1999. 3
5. D. Bartz, M. Meibner, and T. Huttner. OpenGL assisted occlusion culling for large polygonal models. *Computer and Graphics*, 23(3):667–679, 1999. 2
6. D. Cohen-Or, Y. Chrysanthou, and C. Silva. A survey of visibility for walkthrough applications. *SIGGRAPH Course Notes # 30*, 2001. 2
7. S. Coorg and S. Teller. Real-time occlusion culling for models with large occluders. In *Proc. of ACM Symposium on Interactive 3D Graphics*, 1997. 2
8. F. Durand, G. Drettakis, J. Thollot, and C. Puech. Conservative visibility preprocessing using extended projections. *Proc. of ACM SIGGRAPH*, pages 239–248, 2000. 3
9. C. Erikson and D. Manocha. GAPS: General and automatic polygon simplification. In *Proc. of ACM Symposium on Interactive 3D Graphics*, 1999. 2, 5
10. C. Erikson, D. Manocha, and W. Baxter. HLODs for fast display of large static and dynamic environments. *Proc. of ACM Symposium on Interactive 3D Graphics*, 2001. 2, 5, 6, 10
11. J. El-Sana, N. Sokolovsky, and C. Silva. Integrating occlusion culling with view-dependent rendering. *Proc. of IEEE Visualization*, 2001. 1, 3
12. J. El-Sana and A. Varshney. Generalized view-dependent simplification. *Computer Graphics Forum*, pages C83–C94, 1999. 2
13. S. Ellis, M. Young, B. Adelstein, and S. Ehrlich. Discrimination of changes of latency during voluntary hand movement of virtual objects. In *Proc. of the Human Factors and Ergonomics Society*, 1999. 9
14. P. Felzenszwalb and D. Huttenlocher. Efficiently computing a good segmentation. In *Proceedings of IEEE CVPR*, pages 98–104, 1998. 5
15. T.A. Funkhouser, D. Khorramabadi, C.H. Sequin, and

S. Teller. The UCB system for interactive visualization of large architectural models. *Presence*, 5(1):13–44, 1996. 1, 3

16. B. Garlick, D. Baum, and J. Winget. Interactive Viewing of Large Geometric Databases Using Multiprocessor Graphics Workstations. In *SIGGRAPH '90 Course Notes (Parallel Algorithms and Architectures for 3D Image Generation), volume 28*, 1990. 3
17. M. Garland and P. Heckbert. Surface simplification using quadric error bounds. *Proc. of ACM SIGGRAPH*, pages 209–216, 1997. 2
18. N. Govindaraju, A. Sud, S. Yoon, and D. Manocha. Parallel Occlusion Culling for Interactive Walkthroughs using Multiple GPUs TR02-27, Dept. of Computer Science, UNC-Chapel Hill, 2002. 11
19. N. Greene, M. Kass, and G. Miller. Hierarchical Z-buffer visibility. In *Proc. of ACM SIGGRAPH*, pages 231–238, 1993. 2, 3, 4
20. N. Greene. Occlusion culling with optimized hierarchical Z-buffering. In *ACM SIGGRAPH COURSE NOTES ON VISIBILITY, # 30*, 2001. 2, 3, 7
21. 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. 3, 10
22. 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. 2, 4
23. H. Hoppe. Progressive meshes. In *Proc. of ACM SIGGRAPH*, pages 99–108, 1996. 2
24. H. Hoppe. View dependent refinement of progressive meshes. In *ACM SIGGRAPH Conference Proceedings*, pages 189–198. ACM SIGGRAPH, 1997. 2
25. H. Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *IEEE Visualization Conference Proceedings*, pages 35–42, 1998. 1
26. J.B. Kruskal On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of American Mathematical Society*, 7:48–50, 1956. 5
27. 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. 2, 4
28. J. Klowoski and C. Silva. Efficient conservative visibility culling using the prioritized-layered projection algorithm. *IEEE Trans. on Visualization and Computer Graphics*, 7(4):365–379, 2001. 2
29. D. Luebke and C. Erikson. View-dependent simplification of arbitrary polygon environments. In *Proc. of ACM SIGGRAPH*, 1997. 2
30. D. Luebke. A developer's survey of polygon simplification algorithms. *IEEE CG & A*, pages 24–35, May 2001. 2
31. D. Miller and G. Bishop. Latency meter: A device for easily monitoring VE delay. In *Proceedings of SPIE*, Vol. #4660 Stereoscopic Displays and Virtual Reality Systems IX, San Jose, CA, January 2002. 11
32. J. Rossignac and P. Borrel. Multi-resolution 3D approximations for rendering. In *Modeling in Computer Graphics*, pages 455–465. Springer-Verlag, June–July 1993. 2
33. S. Rusinkiewicz and M. Levoy. QSplat: A multiresolution point rendering system for large meshes. *Proc. of ACM SIGGRAPH*, 2000. 1
34. B. Schneider, P. Borrel, J. Menon, J. Mittleman, and J. Rossignac. BRUSH as a walkthrough system for architectural models. In *Fifth Eurographics Workshop on Rendering*, pages 389–399, July 1994. 3
35. W. Schroeder. A topology modifying progressive decimation algorithm. In *Proceedings of Visualization'97*, pages 205–212, 1997. 2
36. G. Schaufler, J. Dorsey, X. Decoret, and F. Sillion. Conservative volumetric visibility with occluder fusion. *Proc. of ACM SIGGRAPH*, pages 229–238, 2000. 2, 3
37. 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. 3, 10
38. A. Sud, N. Govindaraju, and D. Manocha. Partitioning and Clustering Large Environments for Interactive Walkthroughs TR02-29, Dept. of Computer Science, UNC-Chapel Hill, 2002. 5
39. S. J. Teller. *Visibility Computations in Densely Occluded Polyhedral Environments*. PhD thesis, CS Division, UC Berkeley, 1992. 2, 3
40. I. Wald, P. Slusallek, and C. Benthin. Interactive distributed ray-tracing of highly complex models. In *Rendering Techniques*, pages 274–285, 2001. 3
41. P. Wonka, M. Wimmer, and D. Schmalstieg. Visibility preprocessing with occluder fusion for urban walkthroughs. In *Rendering Techniques*, pages 71–82, 2000. 2, 3
42. P. Wonka, M. Wimmer, and F. Sillion. Instant visibility. In *Proc. of Eurographics*, 2001. 1, 2, 3
43. G. Varadhan and D. Manocha. Out-of-Core Rendering of Massive Geometric Environments TR02-28, Dept. of Computer Science, UNC-Chapel Hill, 2002. To appear in Proc. of IEEE Visualization, 2002. 6, 11
44. J. Xia, J. El-Sana, and A. Varshney. Adaptive real-time level-of-detail-based rendering for polygonal models. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):171–183, June 1997. 2
45. H. Zhang, D. Manocha, T. Hudson, and K. Hoff. Visibility culling using hierarchical occlusion maps. *Proc. of ACM SIGGRAPH'97*, 1997. 2, 3, 4

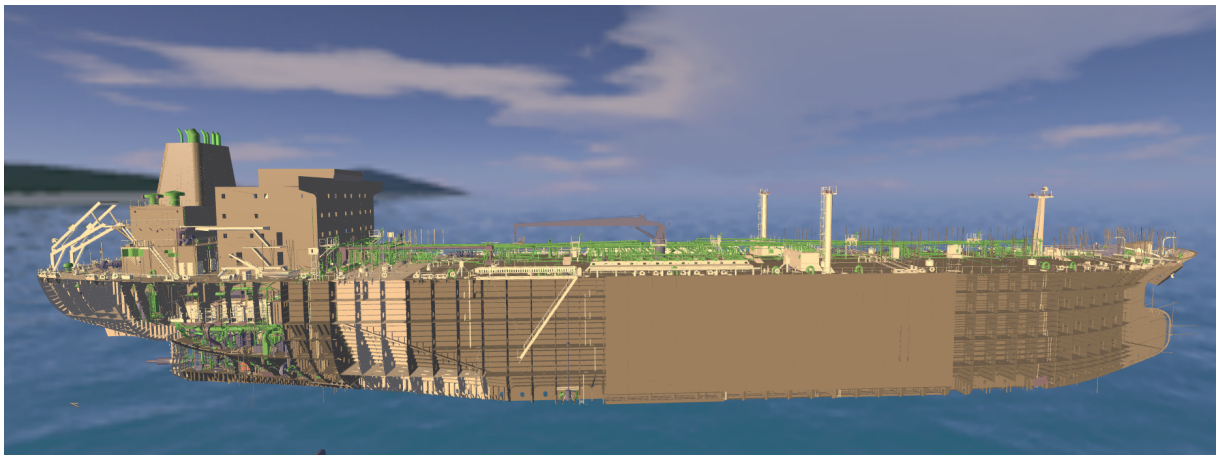
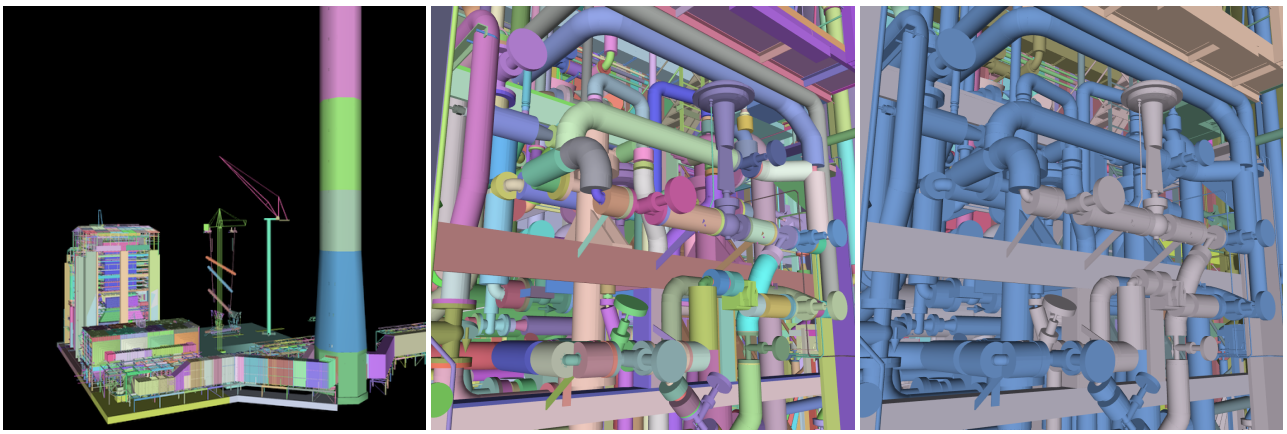
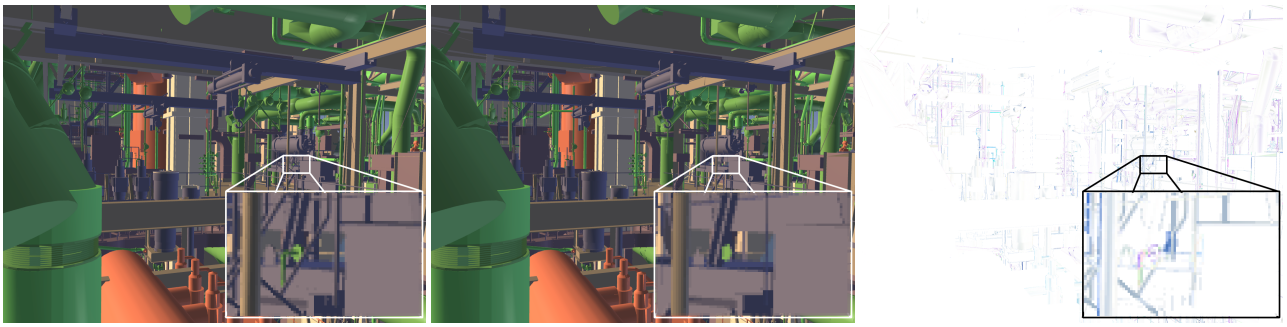


***Plate 1**: Double Eagle Tanker: This 4 gigabyte environment consists of more than 82 million triangles and 127 thousand objects. Our algorithm can render it 11-50 frames per second on an SGI system with two IR2 graphics pipelines and three 300MHz R12000 CPUs.*



(a) Partitioning & Clustering on Power Plant (b) Original Objects in Double Eagle (c) Partitioning & Clustering on Double Eagle

***Plate 2**: The image on the left shows the application of the partitioning and clustering algorithm to the Power Plant model. The middle image shows the original objects in the Double Eagle tanker model with different colors. The right image shows the application of the clustering algorithm on the same model. Each cluster is shown with a different color.*



(a) Pixel Error = 0 (b) Pixel Error = 20 (c) Difference Image

***Plate 3**: The Engine Room in the Double Eagle Tanker displayed without and with HLODs. The inset shows a magnification of one region. Original resolution 1280×960.*