Interactive Display of Complex Environments

Dinesh Manocha
Department of Computer Science
University of North Carolina at Chapel Hill
dm@cs.unc.edu
http://www.cs.unc.edu/~walk

Abstract

Intelligent systems and simulated environments require intuitive interfaces for man-machine interaction. One of the most important components of such a system is to interactively display the data in response to the user. Interactivity is defined as at least 20 updates per second for visual display.

In this paper, we give a brief overview of our work on interactive display of complex datasets. We use algorithms based on visibility culling that do not render the primitives that are not visible to the viewer from a viewpoint. We augment them with level-of-detail representations or multi-resolution modeling techniques which use different approximations of an object in the scene based on its distance from the viewpoint. We also utilize out-of-core techniques that use a limited memory footprint and predictive schemes to load only a portion of the scene that is used by the underlying rendering algorithm. We also present techniques to integrate these approaches and analyze the trade-offs. We have also developed parallel techniques that utilize multiple CPUs and GPUs (graphics processing units) for fast display. The resulting systems have been used for interactive walkthrough of complex datasets including a powerplant model composed of 13 million triangles, Double Eagle Tanker model composed of 82 million triangles and portions of a Boeing 777 model consisting of more than 470 million triangles.

About the Author

Dinesh Manocha is currently a professor of Computer Science at the University of North Carolina at Chapel Hill. He received his Ph.D. in Computer Science at the University of California at Berkeley 1992. He received Junior Faculty Award in 1992, Alfred P. Sloan Fellowship and NSF Career Award in 1995, Office of Naval Research Young Investigator Award in 1996, Honda Research Initiation Award in 1997, and Hettleman Prize for Scholarly Achievements at UNC Chapel Hill in 1998. He has also received best paper & panel awards at the ACM SuperComputing, ACM Multimedia, IEEE Visualization and Eurographics Conferences.

His research has been sponsored by ARO, DARPA, DOE, Ford, Honda, Intel, NSF, and ONR. He has published more than 140 papers in leading conferences and journals on computer graphics, geometric and solid modeling, robotics, symbolic and numeric computation, virtual environments and computational geometry. He has also served as a program committee member for many leading conferences in these areas. He was the program co-chair for the first ACM SIGGRAPH Workshop on Simulation and Interaction in Virtual Environments and program chair of the First ACM Workshop on Applied Computational Geometry. He has also served in the editorial boards of many leading journals.

Interactive Display of Complex Environments

Dinesh Manocha
Department of Computer Science
University of North Carolina at Chapel Hill
dm@cs.unc.edu
http://www.cs.unc.edu/~walk

1 Introduction

One of the fundamental problems in computer graphics, virtual and computer-simulated environments is representation and interactive display of complex environments on current graphics systems. Examples of such environments include:

- Large synthetic environments or CAD datasets corresponding to models of airplanes, submarines, powerplants or other man-made structures.
- Real-world models acquired using cameras, scanners or other sensors (e.g. LIDAR). Recent advances in acquisition technologies have resulted in accurate models of urban environments, indoor scenes, statues, human bodies, etc.
- Simulated datasets arising from tera-flop simulations of mathematical models on finite grids. These are multi-dimensional datasets represented using volumetric representations.

These models can have tens or hundreds of millions (or more) primitives. For example, the digital model of a Boeing 777 consists of almost 500 million triangles, models of statues of Michelangelo acquired using laser rangefinders consist of hundreds of millions of triangles and simulations used in the DOE Accelerated Strategic Computing Initiative (ASCI) result in volumetric datasets with billions of cells.

Interactivity: A driving goal of our research in this area is to build man-machine communication systems that are used to interact with complex environments. A key component of such a system is interactive display of the underlying model. Interactivity provides real-time feedback as the user moves through an environment. The faithful response to user spontaneity is what distinguishes an augmented or a virtual environment from excellent static graphics, which can take minutes or even hours per frame to calculate, and from pre-recorded video sequence. It is also extremely important for virtual reality applications to minimize the system latency in terms of generating the image(s) corresponding to the next frame. In our case, interactivity is defined as 20 updates or more per second.

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 [Funkhouser et al. 1996], terrain datasets [Hoppe 1998], scanned models [Rusinkiewicz and Levoy 2000], and urban environments [Wonka et al. 2001]. 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 [Greene et al. 1993; Cohen-Or et al. 2001; El-Sana et al. 2001]. 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 [Cohen et al. 1996; Erikson and Manocha 1999; Garland and Heckbert 1997; Luebke et al. 2002] alone have difficulty with high-depth-complexity scenes, as they do not address the problems of overdraw and fill load on the graphics pipeline. Furthermore, the view-dependent simplification algorithms can have a high runtime overhead.

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 [Aliaga et al. 1999; Aliaga and Lastra 1999; Darsa et al. 1998; Decoret et al. 1999; Jeschke and Wimmer 2002; Maciel and Shirley 1995; Schaufler and Sturzlinger 1996; Shade et al. 1996; Sillion et al. 1997; Wilson et al. 2001; Wilson and Manocha 2003].

Main Results: We present novel algorithms for interactive display of complex datasets. These include automatic preprocessing and computing a suitable scene graph representation and multiresolution representations of the model. At runtime, the algorithm traverses the scene graph and performs view-frustum and occlusion culling using occlusion queries on current graphics processors and selects an appropriate multiresolution representation for each object or portion of the environment. We present algorithms based on selecting among static LODs or performing view-dependent simplification of the model. We also present techniques for out-of-core rendering, which only use a finite memory footprint. Finally, we render the visible primitives using vertex-arrays on PC workstations.

We highlight the application of our algorithm to two complex environments. These include a power plant model with more than 1200 objects and 12.2 million triangles and and a Double Eagle Tanker composed of over 82 million triangles. Our system is able to render these datasets at interactive rates with little loss in image quality on commodity graphics systems.

2 Scene Representation

In this section, we give an overview of our pre-processing algorithm used to compute a scene graph representation of the complex environment. The scene graph is used by visibility culling and level-of-detail (LOD) selection algorithms at runtime. We initially present a partitioning-clustering-partitioning scheme that is used on large CAD datasets. Next, we present a scheme to compute a hierarchy of LODs that is used by the static LOD selection algorithm at runtime. Finally, we present a scheme for generating a view-dependent and clustering hierarchy that is used by view-dependent runtime algorithms.

Many 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 uses 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.

2.1 Unified Scene Hierarchy

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.

2.1.1 Criteria for Hierarchy

A good hierarchical representation of the scene graph is crucial for the performance of occlusion culling and the overall 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.

1. **Localization**: A good hierarchy should have a good localized representation of the nodes to ensure high fidelity HLODs (hierarchical LODs). Also, HLODs from localized geometry also serve as good aggregate occluders.

2. **Tree Depth**: Balanced hierarchies with lower depth outperform the ones based on unbalanced tree structure. This is due to the relatively fewer number of occlusion tests on potential occludees in the balanced hierarchies.

3. **Non-overlapping Bounding boxes:** The performance of occlusion tests and culling algorithms improves if all the bounding boxes at the same level in the scene graph have little or no overlap. Furthermore, they should provide a tight fit to the underlying geometry. It reduces the repetition of occlusion tests for the same pixel in the occlusion representation.

4. **Object Size:** The objects associated with the nodes in the scene graph are used as occluders as well as potential occludees by different processes in the rendering algorithm. This imposes conflicting constraints on the object size. The occluder selection algorithm performs better on scenes composed of large occluders as they can cull away significant portions of the scene. At the same time, the performance of the occlusion culling algorithm improves if the model consists of small occludees with tight bounding boxes. Moreover, the HLOD based rendering algorithm tends to split large objects, so that the entire object is not rendered at high detail when the viewer is close to the object.

5. **Simplicity:** For interactive runtime performance, the hierarchy should allow for quick LOD selection and culling tests.

In practice, 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. We achieve some of the benefits of localization and bottom-up hierarchies by clustering and partitioning the objects before top-down hierarchy computation.

2.2 Hierarchy Generation

In order to generate uniformly-sized objects, our pre-processing algorithm first redefines the objects using a combination of partitioning and clustering algorithms (see Fig. 1). 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 (axis-aligned bounding box) bounding volume hierarchy on the final redefined set of objects (top-down).

The partitioning (stages 1 and 3) 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 2) was adapted from a computer vision technique for image segmentation [Felzenszwalb and Huttenlocher 1998]. The algorithm uses minimum spanning trees (MST) to represent clusters and is similar to *Kruskal's* algorithm [Kruskal 1956]. More details on the partitioning and clustering algorithm as well as hierarchy computation are given in [Baxter et al. 2002].

2.2.1 HLOD Generation

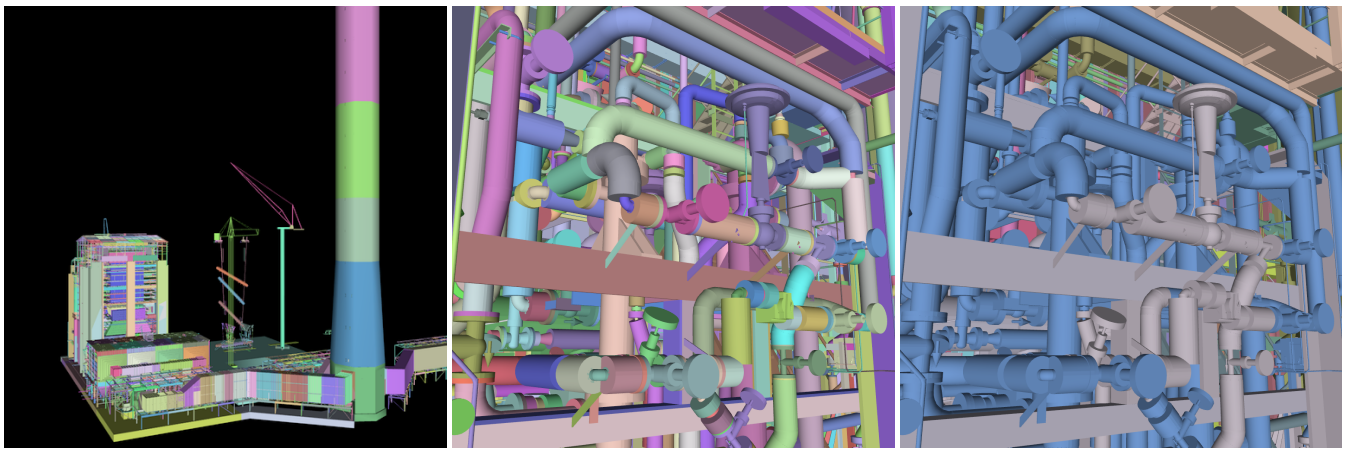
Given the AABB-based scene graph representation, the algorithm computes a series of LODs for each node. It generates a sequence of static LODs and HLODs and the runtime algorithm selects a subset depending on the location of the viewer. The HLODs are computed in a bottom-up manner. The HLODs of the leaf 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 the node's children [Erikson et al. 2001]. We use the GAPS [Erikson and Manocha 1999] 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.

2.3 HLODs as Hierarchical Occluders

Our occlusion culling algorithm uses LODs and HLODs of nodes as occluders to compute the occlusion representation (presented in the next section). 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



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

Figure 1: *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.*

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.

3 View-Dependent Simplification

The rendering algorithms based on static LODs are relatively simple to implement. The resulting algorithms have very little runtime overhead and can efficiently use vertex arrays and display lists. However, switching between different LODs can lead to popping artifacts at runtime. Dynamic simplification (or *view-dependent rendering* (VDR)) algorithms represent an environment using a hierarchy of simplification operations (e.g. vertex hierarchy). The rendering algorithm traverses the hierarchy in an incremental manner and computes a front that satisfies the error bound based on the viewing parameter. VDR algorithms offer several benefits over static LOD-based systems. First, the level of mesh refinement can vary over the surface of an object to provide consistent error in the screen space. This alleviates the popping artifacts that occur when an LOD changes. Furthermore, view information not available during a preprocess can be used to preserve effects such as silhouette edges and specular highlights.

Most view-dependent rendering algorithms use a vertex hierarchy built from an original triangulated mesh. The interior nodes are generated by applying a simplification operation such as an edge collapse or vertex clustering to a set of vertices. The result of the operation is a new vertex that is the parent of the vertices to which the operator was applied. Successive simplification operations build a hierarchy that is either a single tree or a forest of trees. At runtime the mesh is refined to satisfy an error bound specified by the user.

We use the edge collapse operator as the basis for our vertex hierarchies and allow virtual edges so that disjoint parts of the model can be merged. We store an error value corresponding to the local Hausdorff distance from the original mesh with each vertex. This value is used to refine the mesh at runtime by projecting it to screen space where the deviation can be measured in pixels, which is referred to as "pixels of error."

A mesh "fold-over" occurs when a face normal flips during a vertex split or edge collapse. Vertex splits can be applied in a different order at runtime than during the hierarchy generation. This means that even though no fold overs occur during hierarchy generation, they may occur at runtime [El-Sana and Varshney 1999; Hoppe 1997; Xia et al. 1997]. To detect this situation we use a neighborhood test. The face neighborhood is stored for each edge collapse and vertex split operation when creating the hierarchy. At runtime, an operation is considered fold-over safe only if its current neighborhood is identical to the stored neighborhood.

The vertex hierarchy can be interpreted as a fine-grained bounding volume hierarchy. Vertices have bounding volumes enclosing all faces adjacent when the vertex is created during simplification. However, such a bounding volume hierarchy is not well suited for occlusion culling because each bounding volume is small and can occlude only a few primitives. Furthermore, the culling algorithm will have to perform a very high number of occlusion tests.

To address this problem, we partition the vertex hierarchy into clusters and represent them as a cluster hierarchy. Each cluster contains a portion of the vertex hierarchy, as shown in Fig. 2. All vertex relationships from the vertex hierarchy are preserved so that a vertex node may have a child or parent in another cluster. The relationships of the cluster hierarchy are based on those of the vertex hierarchy, so that at least one vertex in a parent cluster has a child vertex in a child cluster.

We characterize clusters based on their *error ratio* and *error range* [Yoon et al. 2003]. The error ratio is defined as the ratio of the maximum error value associated with a vertex in the cluster to that of the minimum. The error range is the difference between the maximum and minimum error values in a cluster.

We use a novel clustering algorithm that traverses the vertex hierarchy to create clusters that are used for occlusion culling. It is based on some of the criteria for good hierarchy for occlusion culling listed in the previous section. The performance of the occlusion culling algorithm depends highly upon the properties of these clusters.

3.1 Cluster Hierarchy Generation

Our clustering algorithm works directly on an input vertex hierarchy without utilizing a spatial subdivision such as an octree. We assume that the vertex hierarchy from which the cluster hierarchy is generated exhibits high spatial coherence and is constructed in a bottom-up manner using edge collapses or vertex merges.

We descend the vertex hierarchy from the roots while creating clusters. An active vertex front is maintained and vertices on the front are added to clusters. When a vertex is added to a cluster, it is removed from the front and replaced with its children. We do not add a vertex to a cluster if it cannot be split in a fold-over safe manner. Thus, the construction of such a cluster will have to wait until dependent vertices are added to other clusters. For this reason, we use a cluster queue and place a cluster at the back of the queue when we attempt to add a vertex that is not fold-over safe. Then, the cluster at the front of queue is processed.

Each cluster in this cluster queue has an associated vertex pri-

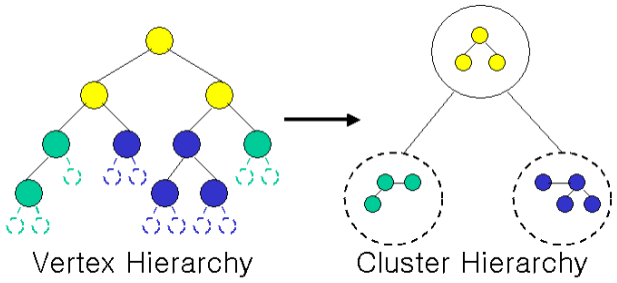


Figure 2: *Construction of the Cluster Hierarchy : On the left is the input vertex hierarchy. The vertices are colored based on the cluster to which they are assigned. The nodes drawn with dotted lines represent the candidate vertices for the clusters, which reside in the vertex priority queue. The two clusters within dotted circles are still in the cluster queue, while the cluster inside the solid circle is finished processing.*

ority queue sorted based on error values. A cluster's vertex queue contains its candidate vertices on the active front. Initially, the cluster queue contains a single cluster. The vertex priority queue associated with this initial cluster contains the roots of the vertex hierarchy. Since candidate vertices for a cluster are processed in order of decreasing error value, it is never the case that a vertex split is dependent upon a split in its own vertex queue.

While the cluster queue is not empty the following steps are performed:

1. Dequeue the cluster, C, at the front of the cluster queue.
2. Dequeue the vertex, v, with highest error from the vertex priority queue.
3. If splitting v is not fold-over safe, return it to the vertex priority queue, place C at the back of the cluster queue and go back to Step 1.
4. If adding v to C makes the error ratio of C too large or increases its vertex count beyond the target:
   (a) Create two children clusters C_l and C_r of C in the cluster queue.
   (b) Partition the vertex priority queue and assign the two resulting queues to C_l and C_r.
   (c) Go back to Step 1 without placing C in the back of the cluster queue; no more vertices will be added to this cluster.
5. Add v to C, update the number of vertices and the error ratio associated with C.
6. Replace v on the active vertex front by its children and enqueue the children in the vertex priority queue associated with C. Go back to Step 2.

4 Visibility and Occlusion Culling

In this section, we present the visibility and occlusion culling algorithm used for interactive display of large and complex environments. We perform occlusion culling using multiple graphics processing units (GPUs) and refer to them as *occlusion-switches* [Govindaraju et al. 2003].

4.1 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.

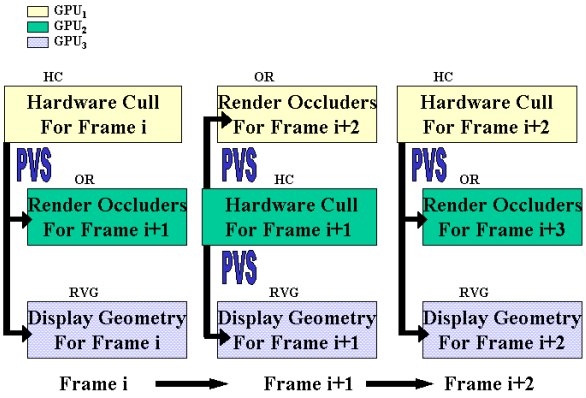


Figure 3: *System Architecture: Each color represents a separate GPU. Note that GPU_1 and GPU_2 switch their roles each frame with one performing hardware culling and other rendering occluders. GPU3 is used as a display client.*

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, say 1024×1024 pixels, can be slow on PC architectures and can take up to 20 milliseconds during each frame. Moreover, constructing the hierarchy in software has 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.

4.2 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 occlud-

ers 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 interchange 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. 3. The overall occlusion algorithm involves no depth buffer readbacks from the GPUs.

4.3 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. 3. 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.

4.4 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.

4.5 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 $\delta_{i,i-1} \cup \delta_{i-1,i}$.

5 Runtime Traversal and Out-of-Core Rendering

Given a scene graph representation based on static LODs, the runtime rendering algorithm traverses the scene graph from the root node during each frame. When it reaches a node , it performs any subset of culling techniques and based on their outcome traverses the scene graph recursively. These include:

- **View-Frustum culling**: Check whether the node's bounding box lies in the viewing frustum.
- **Simplification Culling**: For static-LOD based representations, the algorithm checks whether any object associated with that node satisfies the user-specified error bound. Among all objects associated with the node, the algorithm chooses the coarsest object that meets the screen-space error criteria.
- **Occlusion Culling**: Check whether the node is occluded by other objects. Our current implementation does not perform occlusion culling, though the out-of-core rendering algorithm can be easily extended to handle it.

At the end of the traversal, the algorithm computes a list of objects that need to be rendered during the current frame. We refer to the resulting set of objects as the *front*. The front represents the working set of objects for the current frame and corresponds to a subset of a cut of the scene graph (as shown in Fig 6). A variation of this approach for view-dependent simplification algorithm [Yoon et al. 2003].

5.1 View-Dependent Rendering with Occlusion Culling

The scheme highlighted above is limited to scene graphs with static-LOD based representation. In the view-dependent simplification algorithm, the active vertex front or list and active face list are divided among the clusters so that each cluster maintains its own portion of the active lists. Only clusters that contain vertices on the active front need to be considered during refining and rendering. These clusters are stored as an *active cluster list*.

Prior to rendering a cluster, its active face and vertex lists are updated to reflect viewpoint changes since the last frame. We traverse its active vertex list and use the aforementioned vertex error value to compute which vertices need to be split or collapsed. The error value is projected onto the screen and used as a bound on the deviation of the surface in screen pixels. Vertex splits are performed recursively on front vertices that do not satisfy the bound. For sibling pairs that meet the error bound, we recursively check whether their parent vertex also meets the error bound and if so, collapse the edge (or virtual edge) between the vertex pair.

5.1.1 Maintaining the Active Cluster List

A vertex that is split may have children that belong to a different cluster. The children vertices are activated in their containing clusters and these clusters are added to the active cluster list if they were not previously active. Similarly, during an edge collapse operation,

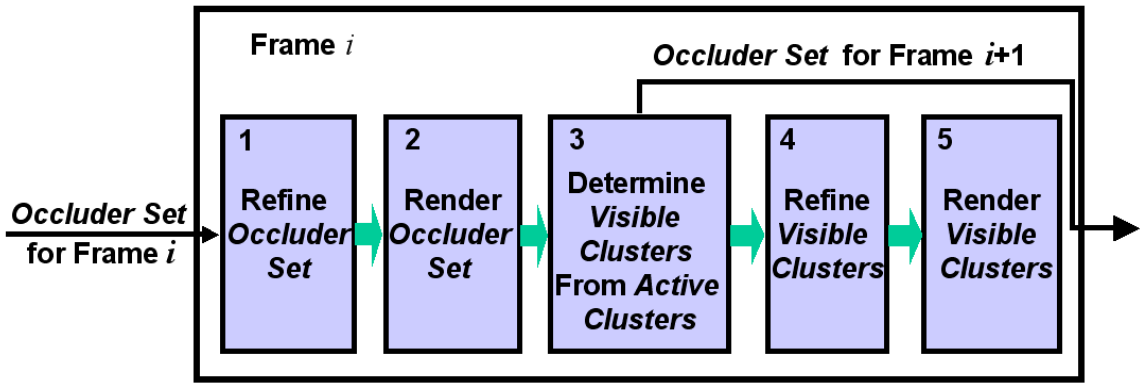


Figure 4: *Runtime System Architecture for View-Dependent Rendering: In each frame the clusters visible in the previous frame are used as an occluder set. In Phases 1 and 2, the occluder set is refined and then rendered in to create a depth map in the z-buffer. Phase 3 tests bounding boxes of all the active clusters against this depth map using occlusion queries. The clusters passing the test are refined and rendered in Phases 4 and 5 and also used as occluders for the next frame.*

the parent vertex is activated in its containing cluster and that cluster is added to the active cluster list. When the last vertex of a cluster is deactivated, the cluster is removed from the active cluster list.

5.1.2 Rendering Algorithm

Our rendering algorithm exploits frame-to-frame coherence in occlusion culling, by using the visible set of clusters from the previous frame as the *occluder set* for the current frame. The algorithm proceeds by rendering the occluder set to generate an occlusion representation in the depth-buffer. Then, it tests all the clusters in the active cluster list for occlusion. Meanwhile, the occluder set is updated for the next frame. An architecture of the runtime algorithm is shown in Fig. 4. Different phases of the algorithm are numbered in the upper left of each box.

5.1.3 Vertex Arrays

On current graphics processors display lists and vertex arrays are significantly faster than immediate mode rendering [Woo et al. 1997]. The changing nature of the visible primitives and dynamically generated LODs in a VDR system are not well suited for using display lists. Thus, we use vertex arrays stored in the graphics processor unit (GPU) memory to accelerate the rendering.

We use a memory manager when the size of the vertices in the active clusters is less than the amount of the memory allocated on the GPU (e.g. 100 MB). Using a least recently used replacement policy, we keep the vertices in GPU memory over successive frames. When the front size exceeds the memory requirement, we still use GPU memory, but do not attempt to keep clusters in this memory for more than one frame.

In many rendering applications all or most of the vertices in a vertex array are used to render faces. But in our case only a fraction of the vertices for a cluster, the active vertices, are used for rendering. This increases the number of bytes per rendered vertex that are transferred to the GPU when using vertex arrays stored in GPU memory. To obtain maximum throughput, we use a minimum ratio of active vertices to total vertices, and any active cluster that does not meet this threshold is rendered in immediate mode.

5.2 Out-of-Core Rendering

The complex environments are represented using gigabyte-sized models. Furthermore, the level-of-detail modeling and visibility culling algorithms compute additional data structures for faster rendering. All this additional information results in a much higher storage complexity for these environments. Any in-core algorithm for interactive display of such datasets needs many gigabytes of main memory.

Given the size of these environments, many out-of-core algorithms have been proposed that limit the runtime memory footprint. Typically these algorithms load only a portion of the environment into the main memory that is needed for the current frame and use prefetching techniques to load portions of the model that may be rendered during subsequent frames. They have been used for architectural models, represented using cells and portals [Funkhouser 1996], large environments that can be easily decomposed into spatial cells [Aliaga et al. 1999], view-dependent simplification based on multi-resolution hierarchies [El-Sana and Chiang 2000], out-of-core simplification of large models [Lindstrom 2000; Lindstrom and Silva 2001; Shaffer and Garland 2001; Bernadini et al. 1999; Cignoni et al. 2000], for GIS applications [Kofler et al. 2000], etc. However, these approaches are not directly applicable to very large, general and complex environments that are composed of tens or hundreds of millions of primitives.

5.3 Front Computation and Prediction

Some nodes in a cut of the scene graph may not be visible from the current viewpoint, and are therefore, not part of the front. The front is not merely a collection of nodes in the scene graph, but includes only one of the objects associated with each node. The index of the object selected for each node represents an additional LOD dimension. As the viewpoint moves, the front changes in many ways. These include different events [Varadhan and Manocha 2002]:

1. **LOD Switching Events:**
   - An object that was in the front may be replaced by a coarser or finer object associated with the same node.
   - An object that was in the front may get replaced either by an object belonging to an ascendant node in the scene graph or by a set of objects from the descendant nodes in the scene graph.

   These events occur when the user zooms in or out of the scene.

2. **Visibility Events**:
   - An object that was in the front may disappear because the corresponding node is no longer visible.
   - An object that wasn't present earlier may appear because the corresponding node has become visible.

   These events occur when the user pans across the scene or because of occlusion events.

Our out-of-core algorithm takes advantage of the fact that the relative changes in the front between successive frames are typically small, and therefore utilizes spatial and temporal coherence in designing an out-of-core rendering algorithm.

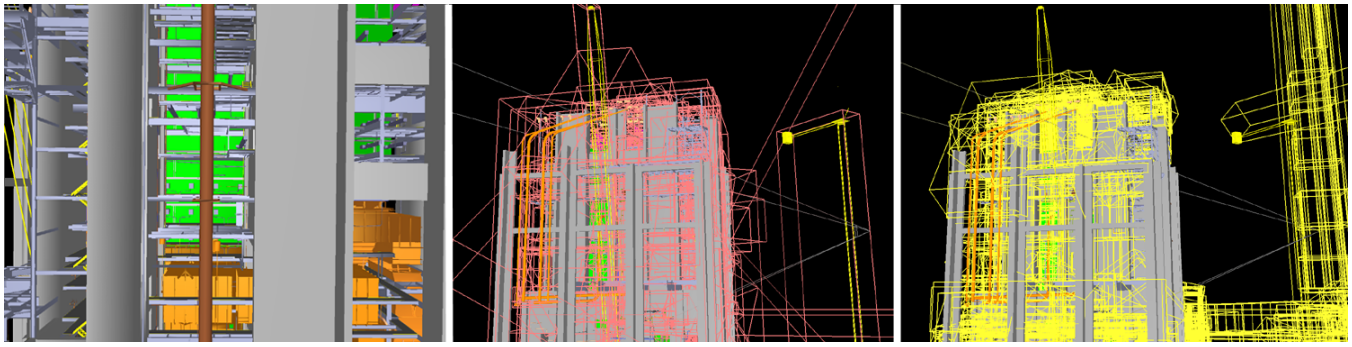


Figure 5: *Occlusion culling in the Power Plant. The left image shows a first person view. The middle image shows a third person view with the bounding boxes of visible clusters shown in pink and the view frustum in white. The right image is from the same third person view with the bounding boxes of occluded clusters in yellow.*

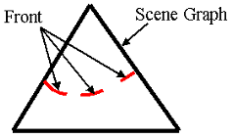


Figure 6: *Front: The exterior black triangle represents the scene graph. Front (in red) is a subset of the cut of the scene graph.*

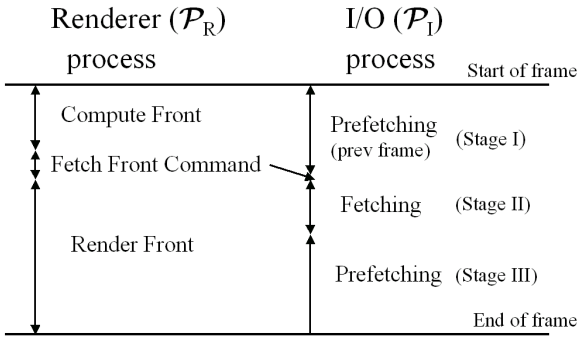


Figure 7: *Parallel Processes: Our algorithm uses two processes, one for rendering and one for I/O. The figure shows the tasks performed by each of the two processes in a given frame time*

5.4 Scene Graph Skeleton & Out-of-Core Representation

In order to traverse the scene graph and compute the front, our rendering algorithm only needs the scene graph *skeleton*. The skeleton includes the nodes and connectivity information like parent-child relationships, as well as additional data structures including bounding boxes and error metrics associated with objects used by different culling algorithms. It stores all the objects on the disk and loads them on the fly based on front computation and the fetching algorithm. The resulting skeleton typically takes only a small fraction of the overall model representation. This division of the model into in-core and out-of-core representations ensures that the main memory overhead is almost equal to the size of the skeleton. The rendering algorithm accepts a memory footprint size as input and we ensure that its memory usage cannot exceed this limit. Moreover, we assume that the given memory footprint is large enough to hold the skeleton.

5.5 Parallel Rendering & I/O Management

Our runtime algorithm uses two synchronous processes, one for rendering and the second one manages the disk I/O. We also use a novel prefetching algorithm that takes into account changes in the front between successive frames and uses a prioritized scheme to handle very large datasets.

Our algorithm uses two main processes: one for scene graph traversal and rendering, and the second one for I/O management and prefetching. Both of them run in parallel and operate synchronously (see Fig 7).

The rendering process traverses the scene graph and computes the front based on the current viewpoint and scene graph skeleton. During this time, the I/O process continues to perform prefetching for the the previous frame (Stage I). Once the rendering process finishes the front computation, it sends a fetch command to the I/O process. On receiving a fetch command, I/O gets synchronized with the rendering process and fetches the objects for the current frame. The fetch command has information about the set of objects in the current front. The I/O process divides this set into two lists:

- $\mathcal{L}_{\mathcal{I}}$: It is the list of all objects that are currently in main memory.
- $\mathcal{L}_{\mathcal{O}}$: It is the list of all objects that are not in the main memory and need to be loaded from the disk.

The I/O process starts loading the objects belonging to $\mathcal{L}_{\mathcal{O}}$ (Stage II in Fig 7). Different objects that constitute the front can be rendered in any order. As a result, the rendering process starts rendering the objects that belong to $\mathcal{L}_{\mathcal{I}}$ and does not wait till all the objects in $\mathcal{L}_{\mathcal{O}}$ are loaded in the main memory. The rendering and the loading of out-of-core objects proceeds in parallel. If the rendering process has rendered all the objects belonging to $\mathcal{L}_{\mathcal{I}}$, it has to wait till new objects are loaded. Whenever the I/O process loads an object, it removes it from $\mathcal{L}_{\mathcal{O}}$ and appends it to $\mathcal{L}_{\mathcal{I}}$. Once the I/O process has fetched all the objects belonging to $\mathcal{L}_{\mathcal{O}}$, it spends the remainder of the frame time prefetching other objects that may be needed for subsequent frames (Stage III). The prefetching algorithm estimates the LOD switching and visibility events along with frame-to-frame coherence [Varadhan and Manocha 2002]. This scheme is mainly limited to using static LODs.

6 Implementation and Performance

We have implemented our simplification and occlusion culling algorithms 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 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 measured the performance of our system on two complex environments: a coal fired Power Plant (shown in Fig. 9) composed of 13 million polygons and 1200 objects, a Double Eagle Tanker

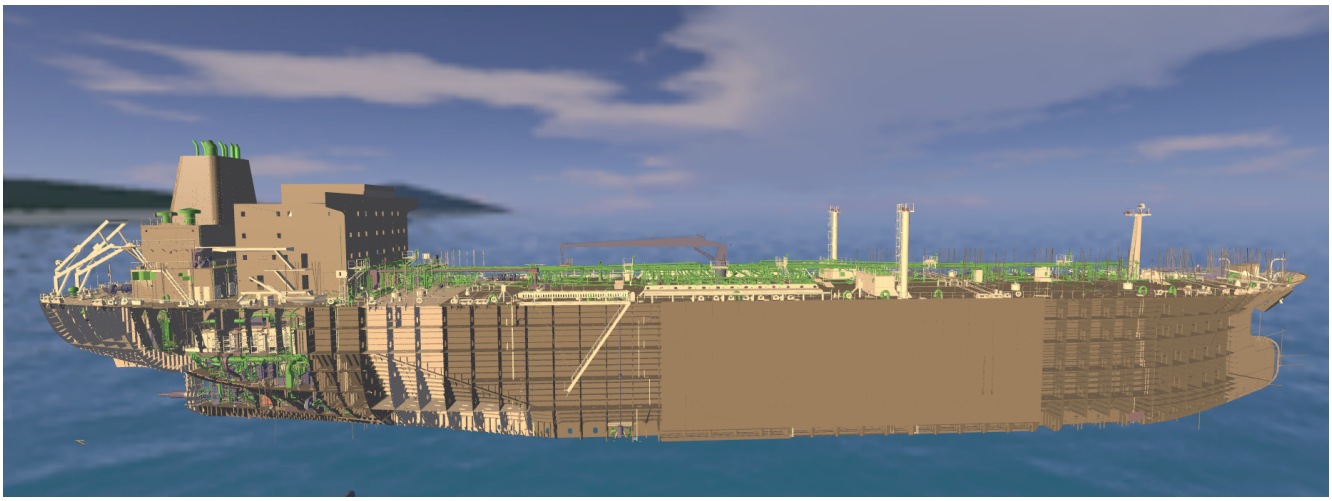


Figure 8: *DoubleEagle Tanker: This 6.5GB environment consists of over* 82.4 *million triangles and* $127K$ *objects. Our static-LOD based rendering algorithm can display it at* $11 - 25$ *frames per second on PC Workstations with NVIDIA GeForce 4 Card*

.

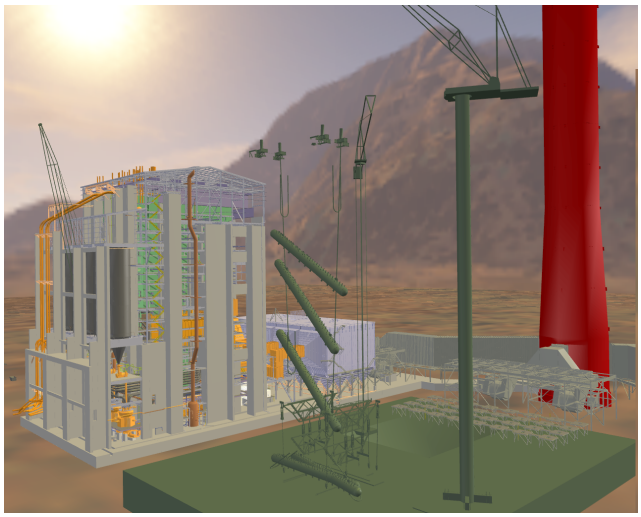


Figure 9: *Coal-Fired Power plant: This* 1.7 *gigabyte environment consist s of over* 13 *million triangles and* 1200 *objects. Our static-LOD based rendering can display it* $13 - 29$ *frames per second on PC Workstations with NVIDIA GeForce 4 Card*

.

(shown in Fig. 8) composed of 82 million polygons and $127K$ objects. 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.

6.0.1 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.

Env	Poly $\times 10^6$	Object Count 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. PP is the powerplant model and DE is the Double Eagle Tanker.*

6.1 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. 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.

6.2 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 id's 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 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 the algorithm is governed by two factors: culling efficiency for occlusion culling and the overall framerates achieved by the rendering algorithm.

- **Culling Efficiency**: Culling efficiency is measured in terms of the ratio of the 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 a 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.

7 Summary and Future Work

We have presented a number of algorithms that are used to accelerate the rendering of large environments on commodity hardware. These include new algorithms to compute a scene graph representation, computation of HLODs for static-LOD based algorithms or using view-dependent simplification algorithms, occlusion culling using multiple GPUs and out-of-core renderings. We have also highlighted the performance of these algorithms on two complex environments.

There are many avenues for future work. 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 Myrinet. At the same time, we expect to perform occlusion culling and render the visible primitives on the same GPU, as opposed to using three different GPUs. Our current implementation of the out-of-core algorithm only takes into account LOD-switching events and we will like to extend it to handle occlusion events as well. Finally, we would like to apply our algorithm to other complex environments and lower the frame rate and end-to-end latency so that we can use it with immersive hardware.

Acknowledgements

The results presented in this paper describe joint work with different students including Bill Baxter, Naga Govindraju, Brian Salomon, Avneesh Sud, Gokul Varadhan and Sungeui Yoon. This work was supported in part by ARO Contract DAAD19-02-1-0390, NSF awards ACI-9876914 and ACR-0118743, ONR Contract N00014-01-1-0067 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.

References

ALIAGA, D., AND LASTRA, A. 1999. Automatic image placement to provide a guaranteed frame rate. In *Proc. of ACM SIGGRAPH*.

ALIAGA, D., COHEN, J., WILSON, A., ZHANG, H., ERIKSON, C., HOFF, K., HUDSON, T., STUERZLINGER, W., BAKER, E., BASTOS, R., WHITTON, M., BROOKS, F., AND MANOCHA, D. 1999. Mmr: An integrated massive model rendering system using geometric and image-based acceleration. In *Proc. of ACM Symposium on Interactive 3D Graphics*.

BAXTER, B., SUD, A., GOVINDARAJU, N., AND MANOCHA, D. 2002. Gigawalk: Interactive walkthrough of complex 3d environments. *Proc. of Eurographics Workshop on Rendering*.

BERNADINI, F., MITTLEMAN, J., AND RUSHMEIER, H. 1999. Case study: Scanning michelangelo' florentine pieta. In *ACM SIGGRAPH 99 Course Notes Course 8*.

CIGNONI, P., MONTANI, C., ROCCHINI, C., AND SCOPIGNO, R. 2000. External memory simplification of huge meshes. In *Technical Report IEI:B4-02-00, IEI, CNR, Pisa, Italy*.

COHEN, J., VARSHNEY, A., MANOCHA, D., ET AL. 1996. Simplification envelopes. In *Proc. of ACM Siggraph'96*, 119–128.

COHEN-OR, D., CHRYSANTHOU, Y., DURAND, F., GREENE, N., KOLTUN, V., AND SILVA, C. 2001. Visibility, problems, techniques and applications. *SIGGRAPH Course Notes # 30*.

DARSA, L., COSTA, B., AND VARSHNEY, A. 1998. Walkthroughs of complex environments using image-based simplification. *Computer and Graphics 22*, 1, 55–69.

DECORET, X., SCHAUFLER, G., SILLION, F., AND DORSEY, J. 1999. Multi-layered impostors for accelerated rendering. *Computer Graphics Forum 18*, 3.

EL-SANA, J., AND CHIANG, Y.-J. 2000. External memory view-dependent simplification. *Computer Graphics Forum 19*, 3, 139–150.

EL-SANA, J., AND VARSHNEY, A. 1999. Generalized view-dependent simplification. *Computer Graphics Forum*, C83–C94.

EL-SANA, J., SOKOLOVSKY, N., AND SILVA, C. 2001. Integrating occlusion culling with view-dependent rendering. *Proc. of IEEE Visualization*.

ERIKSON, C., AND MANOCHA, D. 1999. Gaps: General and automatic polygon simplification. In *Proc. of ACM Symposium on Interactive 3D Graphics*.

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*.

FELZENSZWALB, P., AND HUTTENLOCHER, D. 1998. Efficiently computing a good segmentation. In *Proceedings of IEEE CVPR*, 98–104.

FUNKHOUSER, T., KHORRAMABADI, D., SEQUIN, C., AND TELLER, S. 1996. The ucb system for interactive visualization of large architectural models. *Presence 5*, 1, 13–44.

FUNKHOUSER, T. 1996. Database management for interactive display of large architectural models. In *Graphics Interface*.

GARLAND, M., AND HECKBERT, P. 1997. Surface simplification using quadric error bounds. *Proc. of ACM SIGGRAPH*, 209–216.

GOVINDARAJU, N., LLOYD, B., YOON, S., SUD, A., AND MANOCHA, D. 2003. Interactive shadow generation in complex environments. Tech. rep., University of North Carolina. To Appear in Proc. of ACM SIGGRAPH 2003.

GREENE, N., KASS, M., AND MILLER, G. 1993. Hierarchical z-buffer visibility. In *Proc. of ACM SIGGRAPH*, 231–238.

HOPPE, H. 1997. View dependent refinement of progressive meshes. In *ACM SIGGRAPH Conference Proceedings*, 189–198.

HOPPE, H. 1998. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *IEEE Visualization Conference Proceedings*, 35–42.

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

JESCHKE, S., AND WIMMER, M. 2002. Textured depth mesh for real-time rendering of arbitrary scenes. In *Proc. Eurographics Workshop on Rendering*.

KOFLER, M., GERVAUTZ, M., AND GRUBER, M. 2000. {R}-trees for organizing and visualizing {3D GIS} databases. *The Journal of Visualization and Computer Animation 11*, 3, 129–143.

KRUSKAL, J. B. 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. of American Mathematical Society 7*, 48–50.

LINDSTROM, P., AND SILVA, C. 2001. A memory insensitive technique for large model simplification. In *Proc. of IEEE Visualization*.

LINDSTROM, P. 2000. Out-of-core simplification of large polygonal models. In *Proc. of ACM SIGGRAPH*.

LUEBKE, D., REDDY, M., COHEN, J., VARSHNEY, A., WATSON, B., AND HUEBNER, R. 2002. *Level of Detail for 3D Graphics*. Morgan-Kaufmann.

MACIEL, P., AND SHIRLEY, P. 1995. Visual navigation of large environments using textured clusters. In *ACM Symposium on Interactive 3D Graphics*, 95–102.

RUSINKIEWICZ, S., AND LEVOY, M. 2000. Qsplat: A multiresolution point rendering system for large meshes. *Proc. of ACM SIGGRAPH*.

SCHAUFLER, G., AND STURZLINGER, W. 1996. A three dimensional image cache for virtual reality. *Computer Graphics Forum 15*, 3, C227–C235.

SHADE, J., LISCHINSKI, D., SALESIN, D., DEROSE, T., AND SNYDER, J. 1996. Hierarchical image caching for accelerated walkthroughs of complex environments. In *Proc. of ACM SIGGRAPH*, 75–82.

SHAFFER, E., AND GARLAND, M. 2001. Effient adaptive simplification of massive meshes. In *Proc. of IEEE Visualization*.

SILLION, F., DRETTAKIS, G., AND BODELET, B. 1997. Efficient impostor manipulation for real-time visualization of urban scenery. In *Computer Graphics Forum*, vol. 16.

VARADHAN, G., AND MANOCHA, D. 2002. Out-of-core rendering of massive geometric environments. Tech. Rep. TR02-028, Department of Computer Science, University of North Carolina. Appeared in Proc. of IEEE Visualization, 2002.

WILSON, A., AND MANOCHA, D. 2003. Simplifying complex environments using incremental textured depth meshes. *Proceedings of ACM SIGGRAPH*.

WILSON, A., MAYER-PATEL, K., AND MANOCHA, D. 2001. Spatially-encoded far-field representations for interactive walkthroughs. *Proc. of ACM Multimedia*.

WONKA, P., WIMMER, M., AND SILLION, F. 2001. Instant visibility. In *Proc. of Eurographics*.

WOO, M., NEIDER, J., AND DAVIS, T. 1997. *OpenGL Programming Guide, Second Edition*. Addison Wesley.

XIA, J., EL-SANA, J., AND VARSHNEY, A. 1997. Adaptive real-time level-of-detail-based rendering for polygonal models. *IEEE Transactions on Visualization and Computer Graphics 3*, 2 (June), 171–183.

YOON, S., SALOMON, B., AND MANOCHA, D. 2003. Interactive view-dependent rendering with conservative occlusion culling in complex environments. *Proc. of IEEE Visualization*.

ZHANG, H., MANOCHA, D., HUDSON, T., AND HOFF, K. 1997. Visibility culling using hierarchical occlusion maps. *Proc. of ACM SIGGRAPH*.