Quick-VDR: Interactive View-Dependent Rendering of Massive Models

Sung-Eui Yoon Brian Salomon Russell Gayle Dinesh Manocha

University of North Carolina at Chapel Hill
{sungeui,salomon,rgayle,dm}@cs.unc.edu
http://gamma.cs.unc.edu/QVDR

ABSTRACT

We present a novel approach for interactive view-dependent rendering of massive models. Our algorithm combines view-dependent simplification, occlusion culling, and out-of-core rendering. We represent the model as a clustered hierarchy of progressive meshes (CHPM). We use the cluster hierarchy for coarse-grained selective refinement and progressive meshes for fine-grained local refinement. We present an out-of-core algorithm for computation of a CHPM that includes cluster decomposition, hierarchy generation, and simplification. We make use of novel cluster dependencies in the preprocess to generate crack-free, drastic simplifications at runtime. The clusters are used for occlusion culling and out-of-core rendering. We add a frame of latency to the rendering pipeline to fetch newly visible clusters from the disk and to avoid stalls. The CHPM reduces the refinement cost for view-dependent rendering by more than an order of magnitude as compared to a vertex hierarchy. We have implemented our algorithm on a desktop PC. We can render massive CAD, isosurface, and scanned models, consisting of tens or a few hundreds of millions of triangles at $10-35$ *frames per second with little loss in image quality.*

Keywords: Interactive display, view-dependent rendering, occlusion culling, external-memory algorithm, levels-of-detail

1 INTRODUCTION

Recent advances in acquisition, modeling, and simulation technologies have resulted in large databases of complex geometric models. These gigabyte-sized datasets consist of tens or hundreds of millions of polygons. The enormous size of these datasets poses a number of challenges in terms of interactive display and manipulation on current graphics systems.

View-dependent simplification and rendering have been actively researched for interactive display of large datasets [20, 28, 35]. These algorithms have many appealing properties because they compute different levels-of-detail (LODs) over different regions of the model. The selection of appropriate LODs is based on viewposition, local illumination and other properties such as visibility and silhouettes. Most view-dependent algorithms precompute a vertex hierarchy of the model and perform incremental computations between successive frames. This reduces the "popping" artifacts that can occur while switching between different LODs. The algorithms generally maintain a cut, or *active vertex front*, across the hierarchy and use it for mesh refinement. The front is traversed each frame and is updated based on the change in view parameters. In order to preserve the local topology, dependencies are introduced between simplification operations.

Current representations and refinement algorithms for view-dependent rendering do not scale well to large models composed of tens or hundreds of millions of triangles. The refinement cost is a function of the front size and may be prohibitively expensive for massive models. Furthermore, resolving dependencies in the vertex hierarchy can be expensive (e.g. hundreds of milliseconds or more per frame).

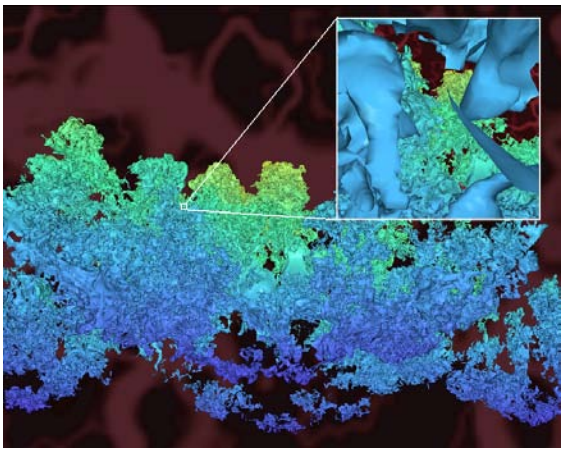

Figure 1: This image shows the application of Quick-VDR to a complex isosurface (100M triangles) generated from a very high resolution 3D simulation of Richtmyer-Meshkov instability and turbulence mixing. The right inset image shows a zoomed view. The isosurface has high depth complexity, holes, and a very high genus. Quick-VDR can render it at $11-21$ frames per second on a PC with NVIDIA GeForce FX5950 card and uses a memory footprint of 600MB.

In addition to reducing the refinement cost, it is necessary to integrate view-dependent simplification algorithms with occlusion culling and out-of-core rendering. Occlusion culling computes a set of potentially visible primitives during each frame and is needed to handle high depth complexity models. Out-of-core rendering techniques operate with a bounded memory footprint and are required to render massive models on commodity graphics systems with limited memory. Algorithms for occlusion culling and out-of-core techniques also perform computations based on the view parameters. However, no known algorithms integrate conservative occlusion culling and out-of-core rendering with vertex hierarchies.

Main Contributions: We present a new view-dependent rendering algorithm (Quick-VDR) for interactive display of massive models. We use a novel scene representation, a *clustered hierarchy of progressive meshes* (CHPM). The cluster hierarchy is used for coarse-grained view-dependent refinement. The progressive meshes provide fine-grained local refinement to reduce the popping between successive frames without high refinement cost.

Our rendering algorithm uses temporal coherence and occlusion queries for visibility computations at the cluster level. We account for visibility events between successive frames by combining fetching and prefetching techniques for out-of-core rendering. Our rendering algorithm introduces one frame of latency to fetch newly visible clusters from the disk and to avoid stalling the pipeline.

Quick-VDR relies on an out-of-core algorithm to compute a CHPM that performs a hierarchical cluster decomposition and simplification. We introduce the concept of *cluster dependencies* between adjacent clusters to generate crack-free and drastic simplifications of the original model.

We have implemented and tested Quick-VDR on a commodity

IEEE Visualization 2004
October 10-15, Austin, Texas, USA
0-7803-8788-0/04/$20.00 ©2004 IEEE

PC with NVIDIA 5950FX Ultra card. To illustrate the generality of our approach we have highlighted its performance on several models: a complex CAD environment (12M triangles), scanned models (372M triangles), and an isosurface (100M triangles). We can render these models at $10 - 35$ frames per second using a limited memory footprint of $400 - 600$MB.

Advantages: Our approach integrates view-dependent simplification, conservative occlusion culling, and out-of-core rendering for high quality interactive display of massive models on current graphics systems. As compared to prior approaches, Quick-VDR offers the following benefits:

1. **Lower refinement cost:** The overhead of view-dependent refinement in the CHPM is one to two orders of magnitude lower than vertex hierarchies for large models.
2. **Massive models:** We are able to compute drastic simplifications of massive models, using hierarchical simplification with cluster dependencies, necessary for interactive rendering.
3. **Runtime performance:** Quick-VDR renders CHPMs using a bounded memory footprint and exploits the features of current graphics processors to obtain a high frame rate.
4. **Image quality:** We significantly improve the frame rate with little loss in image quality and alleviate popping artifacts between successive frames.
5. **Generality:** Quick-VDR is a general algorithm and applicable to all types of polygonal models, including CAD, scanned, and isosurface.

Organization: The rest of the paper is organized in the following manner. We give a brief overview of related work in Section 2 and describe our scene representation and refinement algorithm in Section 3. Section 4 describes our out-of-core algorithm to generate a CHPM for a large environment. We present the rendering algorithm in Section 5 and highlight its performance in Section 6. We compare our algorithm with other approaches in Section 7 and discuss some of its limitations.

2 Related Work

We give a brief overview of the previous work in view-dependent rendering, out-of-core rendering, occlusion culling, and hybrid approaches to massive model rendering.

2.1 View-Dependent Simplification

View-dependent simplification of complex models has been an active area of research over the last decade. View-dependent rendering originated as an extension of the progressive mesh (PM) [19]. A PM is a linear sequence of increasingly coarse meshes built from an input mesh by repeatedly applying edge collapse operations. It provides a continuous resolution representation of an input mesh and is useful for efficient storage, rendering, and transmission.

Xia and Varshney [35] and Hoppe [20] organized the PM as a vertex hierarchy (or view-dependent progressive mesh (VDPM)) instead of a linear sequence. Luebke and Erikson [28] developed a similar approach employing octree-based vertex clustering operations and used it for dynamic simplification. El-Sana and Varshney [12] extended these ideas using a uniform error metric based on cubic interpolants and reduced the cost of runtime tests.

Pajarola [29] improved the update rate of runtime mesh selection by exploiting properties of the half-edge mesh representation and applied it to manifold objects. El-Sana and Bachmat [9] presented a mesh refinement prioritization scheme to improve the runtime performance.

2.2 Out-of-core Computation and Rendering

Many algorithms have been proposed for out-of-core simplification. These include [27, 33, 4] for generating static LODs. Hoppe [21] extended the VDPM framework for terrain rendering by decomposing the terrain data into blocks, generating a block hierarchy and simplifying each block independently. Prince [30] extended this out-of-core terrain simplification to handle arbitrary polygonal models.

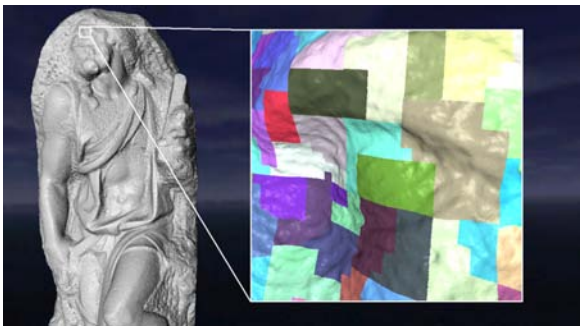

Figure 2: Scan of Michelangelo's St. Matthew. *This 9.6GB scanned model consists of 372M triangles. The right inset image shows clusters in color from a 64K cluster decomposition of the model. Quick-VDR is able to render this model at $13 - 23$ frames per second using a memory footprint of 600MB.*

El-Sana and Chiang [10] segment a mesh into sub-meshes such that the boundary faces are preserved while performing edge-collapse operations. DeCoro and Pajarola [8] present an external data structure for the half-edge hierarchy and an explicit paging system for out-of-core management of view-dependent rendering. Lindstrom [25] presents an approach for out-of-core simplification and view-dependent visualization. Cignoni et al. [3] present an efficient method for out-of-core rendering of terrain data.

2.3 Occlusion Culling

The problem of computing portions of the scene visible from a given viewpoint has been well-studied [5]. Many specialized object-space algorithms have been developed for architectural models or urban environments. For general environments, image-based occlusion representations are widely used and the resulting algorithms use graphics hardware to perform visibility computations [17, 37, 24]. In some cases, additional graphics processors have been used for visibility computations [34, 16]. All of these algorithms load the entire scene graph into main memory.

2.4 Hybrid Algorithms for Rendering Acceleration

Many hybrid algorithms have been proposed that combine model simplification with visibility culling or out-of-core data management. The Berkeley Walkthrough system [14] combines cells and portals based on visibility computation algorithms with static LODs for architectural models. The MMR system [1] combines static LODs with occlusion culling and out-of-core computation and is applicable to models that can be partitioned into rectangular cells. The QSplat system [31] uses a compact bounding volume hierarchy of spheres for view-frustum and backface culling, level-of-detail control and point-based rendering. Erikson et al. [13] and Samanta et al. [32] use hierarchies of static LODs (HLODs) and view-frustum culling. Govindaraju et al. [16] integrate HLODs and conservative occlusion culling for interactive display of large CAD environments. Yoon et al. [36] presented an in-core algorithm to combine view-dependent simplification with conservative occlusion culling. El-Sana et al. [11] combined view-dependent rendering with approximate occlusion culling. The *iWalk* system [6, 7] partitions the space into cells and performs out-of-core rendering of large architectural and CAD models on commodity hardware using non-conservative occlusion culling.

3 Overview

In this section we introduce some of the terminology and representations used by Quick-VDR. We also give a brief overview of our approach for out-of-core hierarchical simplification and rendering.

3.1 View-Dependent Rendering of Massive Datasets

Most of the prior work on view-dependent simplification and rendering of large datasets uses vertex hierarchies such as VDPM.

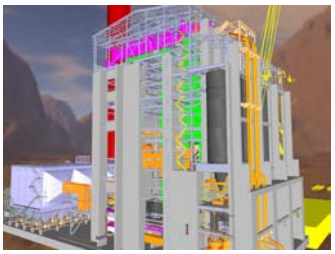

Figure 3: Power Plant. *A rendering of the power plant model using our runtime algorithm. This model consists of over 12M triangles and has high depth complexity. It is rendered at an average of* 33 *FPS using* 400*MB of main memory by our system.*

These approaches augment each edge collapse with *dependency* information related to the local neighborhood at the time of the edge collapse during construction. This information is used to prevent "fold-overs" whereby a face normal is reversed at runtime. However, many issues arise in applying these approaches to massive datasets composed of tens or hundreds of millions of triangles. Traversing and refining an *active vertex front* across a vertex hierarchy composed of tens of millions of polygons can take hundreds of milliseconds per frame. Also, resolving the dependencies can lead to non-localized memory accesses which can be problematic for out-of-core rendering. Moreover, performing occlusion culling and out-of-core rendering using vertex hierarchies can become expensive.

3.2 Scene Representation

We propose a novel representation, a clustered hierarchy of progressive meshes (CHPM), for view-dependent rendering of massive datasets. The CHPM consists of two parts:

Cluster Hierarchy: We represent the entire dataset as a hierarchy of clusters, which are spatially localized mesh regions. Each cluster consists of a few thousand triangles. The clusters provide the capability to perform coarse-grained view-dependent (or selective) refinement of the model. They are also used for visibility computations and out-of-core rendering.

Progressive Mesh: We precompute a simplification of each cluster and represent a linear sequence of edge collapses as a progressive mesh (PM). The PMs are used for fine-grained local refinement and to compute an error-bounded simplification of each cluster at runtime.

We refine the CHPM at two levels. First we perform a coarse-grained refinement at the cluster level. Next we refine the PMs of the selected clusters. The PM refinement provides smooth LOD transitions.

3.2.1 Cluster Hierarchy

Conceptually, a cluster hierarchy is similar to a vertex hierarchy. However, every node of a cluster hierarchy represents a set of vertices and faces rather than a single vertex. At runtime, we maintain an *active cluster list* (ACL), which is similar to an active front in a vertex hierarchy and perform selective refinement on this list via the following operations:

- **Cluster-split:** A cluster in the active cluster list is replaced by its children.
- **Cluster-collapse:** Sibling clusters are replaced by their parent.

These operations are analogous to the vertex split and collapse in a vertex hierarchy but provide a more coarse-grained approach to selective refinement.

3.2.2 Progressive Meshes and Refinement

A PM is a mesh sequence built from an input mesh by a sequence of edge collapse operations. The inverse operation, a vertex split, restores the original vertices and replaces the removed triangles. We use the notation M_A^0 to represent the most simplified or *base mesh* of cluster A. Moreover, M_A^i is computed by applying a vertex split operation to M_A^{i-1} (as shown in Fig. 4). Each PM is stored as a base mesh and a series of vertex split operations. For each cluster in the ACL we select the position in the edge collapse sequence that meets the allowed error for the cluster with the least number of faces. We take advantage of temporal coherence by starting with the position from the previous frame. In practice, refining a PM is a very fast operation and requires no dependency checks.

The PMs allow us to perform smooth LOD transitions at the level of a single cluster. In order to perform globally smooth LOD transitions we require that the changes to the ACL between successive frames are also smooth. If cluster C is the parent of clusters A and B, we set the highest resolution mesh approximation of cluster C's PM to be the union of the base meshes of cluster A and B's PMs. That is, $M_C^k = M_A^0 \bigcup M_B^0$ (see Fig. 4). Therefore, the cluster-collapse and cluster-split operations introduce no popping artifacts.

3.3 Simplification Error Bounds

A key issue is computation of errors associated with the LODs generated at runtime. Each cluster contains progressive mesh that can be refined within a range of object space error values. We refer to this range as the *error-range* of a cluster and is expressed as a pair: (*min-error*, *max-error*). The *max-error* is the error value associated with the base mesh (M^0) and the *min-error* is the error value associated with the highest resolution mesh (e.g. M_A^i, M_B^j, and M_C^k in Fig. 4).

The allowed runtime error is expressed in screen-space as a *pixels-of-error* (POE) value. Using the POE value and the minimum distance between a cluster and the viewpoint, we compute the maximum object-space error that is allowed for the cluster, called the *error-bound*. View-dependent refinement of each cluster based on the *error-bound* will be explained in Sec. 5.1.

3.4 Preprocess

Given a large dataset, we compute a CHPM representation. Our out-of-core algorithm begins by decomposing the input mesh into a set of clusters. The clusters are input to a cluster hierarchy generation algorithm which builds a balanced hierarchy in a top-down manner. We perform out-of-core hierarchical simplification using the cluster hierarchy.

Each cluster should be independently refinable at runtime for efficiency and out-of-core rendering. For this reason, boundary constraints on simplification are introduced during hierarchical simplification. While guaranteeing crack-free simplifications at runtime, these constraints can prevent drastic simplification and may dramatically increase the number of faces rendered at runtime. To alleviate these problems we introduce *cluster dependencies* that allow boundary simplification while maintaining crack-free rendering at runtime.

3.5 Rendering Algorithm

Quick-VDR uses the CHPM as a scene representation for out-of-core view-dependent rendering and occlusion culling. Coarse-grained selective refinement is accomplished by applying cluster-split and cluster-collapse operations to the ACL. Cluster dependencies assure that consistent cluster boundaries are rendered and that we are able to compute drastic simplifications. We use temporal coherence to accelerate refinement and to perform occlusion culling.

Quick-VDR uses the operating system's virtual memory manager through a memory mapped file for out-of-core rendering. In order to overcome the problem of accurately predicting the occlusion events, we introduce one frame of latency in the runtime pipeline. This allows us to load newly visible clusters to avoid stalling the rendering pipeline.

4 Building a CHPM

In this section we present an out-of-core algorithm to compute CHPMs for massive models. Our algorithm proceeds in three steps. First, we decompose the input mesh into a set of clusters. The

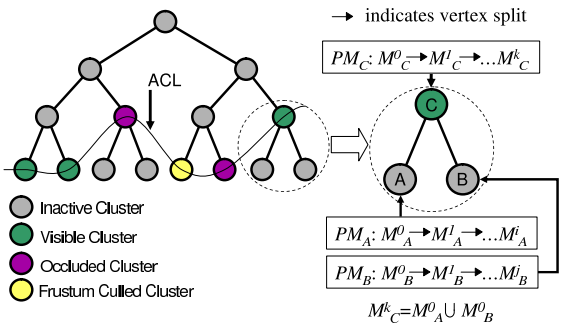

Figure 4: CHPM: Clustered Hierarchy of Progressive Meshes. *At runtime the active cluster list (ACL) represents a front in the cluster hierarchy containing the clusters of the current mesh (left). Clusters on the ACL are classified as visible, frustum culled, or occlusion culled. The PMs (right) of visible clusters are refined to meet the screen space error bound by selecting a mesh from the PM mesh sequence. When the ACL changes, smooth LOD transitions occur because the most refined mesh of each PM is equal to the union of the base meshes of its children.*

decomposition occurs in several passes to avoid loading the entire input mesh at once. Next, we construct the cluster hierarchy by repeatedly subdividing the mesh in a top-down manner. Finally, we compute progressive meshes for each cluster by performing a bottom-up traversal of the hierarchy.

4.1 Cluster Decomposition

The clusters form the underlying representation for both the preprocessing step as well as out-of-core view-dependent rendering with occlusion culling. We decompose the model into clusters, which are spatially localized portions of the input mesh. The generated clusters should be nearly equally sized in terms of number of triangles for several reasons. This property is desirable for out-of-core mesh processing to minimize the memory requirements. If the cluster size as well as the number of clusters required in memory at one time are bounded, then simplification and hierarchy construction can be performed with a constant memory footprint. Moreover, enforcing spatial locality and uniform size provides higher performance for occlusion culling and selective refinement.

The out-of-core cluster decomposition algorithm proceeds in four passes. The first three passes only consider the vertices of the original model and create the clusters while the fourth assigns the faces to the clusters. We use a variation of the cluster decomposition algorithm for out-of-core compression of large datasets presented in [22]. However, our goal is to decompose the mesh for out-of-core processing and view-dependent rendering. As a result, we only compute and store the connectivity information used by the simplification algorithm. The four passes of the algorithm are:

First vertex pass: We compute the bounding box of the mesh.

Second vertex pass: We compute balanced-size clusters of vertices (e.g. 2K vertices). Vertices are assigned to cells of a uniform 3D grid. A graph is built with nodes representing the non-empty cells weighted by vertex count. Edges are inserted between each cell and its k nearest neighbors using an approximate nearest neighbor algorithm [2] (e.g. k=6). We use a graph partitioning algorithm [18] to partition the graph and compute balanced-size clusters.

Third vertex pass: Based on the output of the partitioning, we assign vertices to clusters and reindex the vertices. A mapping is created that maps the original vertex indices to the new indices. This mapping can be quite large so it is stored in a file that can be accessed in blocks with LRU paging.

Face pass: In the final pass, we assign each face to a single cluster. The mapping file created in the previous pass is used to locate the vertices. The vertices of faces spanning multiple clusters are marked as constrained for simplification. These vertices make up the boundaries between clusters and are referred to as *shared vertices* while the remaining vertices are referred to as *interior vertices*.

The resulting cluster decomposition consists of manageable mesh pieces that can be transparently accessed in an out-of-core manner for hierarchy generation and simplification, while preserving all the original connectivity information. Different clusters computed for the St. Matthew model are shown in Fig. 2.

4.2 Cluster Hierarchy Generation

In this section, we present an algorithm to compute the cluster hierarchy. The clusters computed by the decomposition algorithm described in the previous section are used as the input to hierarchy generation. Our goal is to compute a hierarchy of clusters with the following properties:

Nearly equal cluster size As previously discussed, consistent cluster size is important for memory management, occlusion culling, and selective refinement. Clusters at all levels of the hierarchy must possess this property.

Balanced cluster hierarchy During hierarchical simplification, cluster geometry is repeatedly simplified and merged in a bottom up traversal. The hierarchy must be well balanced so that merged clusters have nearly identical *error-range*s.

Minimize shared vertices The number of shared vertices at the cluster boundary should be minimized for simplification. Otherwise, in order to maintain consistent cluster boundaries, the simplification will be over-constrained and may result in lower fidelity approximations of the original model.

High spatial locality The cluster hierarchy should have high spatial locality for occlusion culling and selective refinement.

We achieve these goals by transforming the problem of computing a cluster hierarchy into a graph partitioning problem and compute the hierarchy in a top down manner. Each cluster is represented as a node in a graph, weighted by the number of vertices. Clusters are connected by an edge in the graph if they share vertices or are within a threshold distance of each other. The edges are weighted by the number of shared vertices and the inverse of the distance between the clusters, with greater priority placed on the number of shared vertices. The cluster hierarchy is then constructed in a top-down manner by recursively partitioning the graph into halves considering the weights, thus producing a binary tree. The weights guide the partitioning algorithm [23] to produce clusters with spatial locality while tending towards fewer shared vertices. The top down partitioning creates an almost balanced hierarchy.

4.3 Out-of-Core Hierarchical Simplification

We simplify the mesh by traversing the cluster hierarchy in a bottom-up manner. Each level of the cluster hierarchy is simplified in a single pass so the simplification requires $\lceil log_2(n) + 1 \rceil$ passes where n is the number of leaf clusters. During each pass only the cluster being simplified and clusters with which it shares vertices must be resident in memory.

Simplification operations are ordered by a priority queue based upon quadric errors [15]. We build the progressive meshes (PMs) for each cluster by applying "half-edge collapses". The half-edge collapse, in which an edge is contracted to one of the original vertices, is used to avoid creation of new vertices during simplification. Edges adjacent to shared vertices are not collapsed during simplification. The edge collapses and associated error values are stored along with the most refined mesh of a PM. After creating the PM, the *error-range* of the cluster is computed based on the errors of the PM's original and base mesh.

When proceeding to the next level up the hierarchy, the mesh within each cluster's PM is initialized by merging the base meshes of the children. Constraints on vertices that are shared by two clusters being merged are removed thereby allowing simplification of the merged boundary. Since the intermediate clusters should be nearly the same size as the leaf level clusters, each cluster is simplified to half its original face count at each level of the hierarchy.

As simplification proceeds, a file is created for the progressive mesh of each cluster. However, handling many small files at run-

time is inefficient. The PM files are merged into one file which can be memory mapped to allow the OS to perform memory management of the PMs. The file is stored in a breadth first manner in an attempt to match the probable access pattern during runtime refinement.

4.4 Boundary Constraints and Cluster Dependencies

In order to support out-of-core rendering and to allow efficient refinement at runtime, the PMs of each cluster should be independently refinable while still maintaining a crack-free consistent mesh. To achieve this, our algorithm detects the shared vertices and restricts collapsing the edges adjacent to them during hierarchical simplification. As simplification proceeds up the hierarchy, these constraints are removed because the clusters sharing the vertices have been merged.

While these constraints assure crack-free boundaries between clusters at runtime, they can be overly restrictive. After simplifying several levels of the hierarchy most of the vertices in the base mesh of the PM are shared vertices. As illustrated in Fig. 5 this problem arises along boundaries between clusters that are merged at higher levels in the hierarchy. This can degrade the quality of simplification, and impedes drastic simplification. In Fig. 5 notice that the boundary between B and C is not merged until higher up the hierarchy.

This constraint problem is common to many hierarchical simplification algorithms that decompose a large mesh for view-dependent rendering [21, 30] or compute hierarchies of static LODs (HLODs) [13, 16, 32].

We introduce *cluster-level dependencies* to address this constraint problem. The intuition behind dependencies is that precomputed simplification constraints on shared vertices can be replaced by runtime dependencies. During hierarchical simplification, we may collapse an edge adjacent to a shared vertex. The clusters sharing that vertex are marked as dependent upon each other. This boundary simplification occurs on the merged mesh prior to PM generation for the cluster. In Fig. 5 clusters E and F are marked dependent and thereby allow the boundary to be simplified.

At runtime, splitting a cluster forces all its dependent clusters to split so that the boundaries are rendered without cracks. Likewise, a parent cluster cannot be collapsed unless all of its dependent clusters have also been collapsed. In Fig. 5, clusters E and F must be split together and clusters A, B, C, and D must be collapsed together (assuming E and F are dependent). For example, if clusters B and F are rendered during the same frame, their boundary will be rendered inconsistently and may have cracks.

Although cluster dependencies allow boundary simplification, we need to use them carefully. Since splitting a cluster forces its dependent clusters to split, dependencies will cause some clusters to be rendered that are overly conservative in terms of their *error-bound*. Furthermore, the boundaries change in one frame which may cause popping artifacts. This can be exacerbated by "chained" dependencies in which one cluster is dependent upon another cluster which is in turn dependent upon a third cluster, and so on.

To avoid these potential runtime problems, we prioritize clusters for boundary simplification. At each level of hierarchical simplification the clusters are entered into a priority queue. Priorities are assigned as the ratio of average error of shared vertices to the average error of interior vertices. A cluster, A, is removed from the head of the priority queue. For each cluster, B, that shares at least j (e.g. 5) vertices with A we apply boundary simplification between A and B if the following conditions are met:

1. A and B will not be merged within a small number of levels up the cluster hierarchy (e.g., 2).
2. A and B have similar *error-range*.
3. A dependency between A and B will not introduce a chain (unless all the clusters in the chain share vertices).

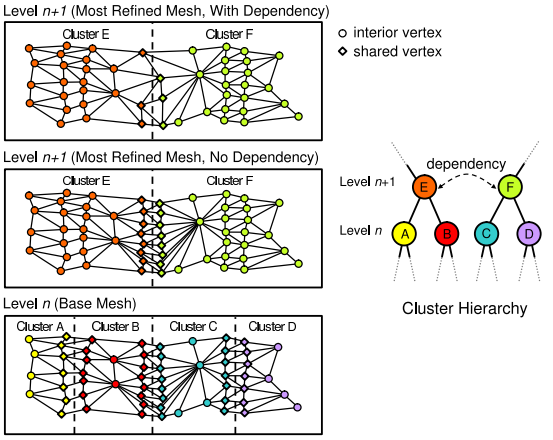

Figure 5: Dependencies. *After simplifying level n of the hierarchy the boundaries AB, BC, and CD are all under-simplified because they are constrained. When initializing the base meshes of E and F prior to simplifying level $n+1$, two of these boundaries, AB and CD, are no longer constrained because they have been merged. The boundary BC was not merged and will remain under-simplified. We can, however, simplify the faces along this boundary if we mark E and F as dependent.*

This is repeated for each cluster in the priority queue. The first condition avoids creating dependencies between clusters that are resolved within only a few additional hierarchy levels. The second condition discourages dependencies between those clusters that are unlikely to be simultaneously present in the ACL at runtime. The third condition prevents long dependency chains and preserves selective refinement at the cluster level. The cluster dependencies ensure that a sufficient number of shared vertices are collapsed at each level of the hierarchy while still generating and rendering crack-free simplifications at runtime.

5 Interactive Out-of-Core Display

In the previous section, we described an algorithm to compute the CHPM. In this section, we present a novel rendering algorithm that uses the CHPM for occlusion culling, view-dependent refinement and out-of-core rendering. The entire representation including the PMs is stored on the disk. We load the coarse-grained cluster hierarchy into main memory and keep a working set of PMs in main memory. The cluster hierarchy without the PMs is typically a few megabytes for our benchmark models (e.g. 5MB for St. Matthew model). We perform coarse-grained refinement at the cluster level and fine-grained refinement at the level of PMs. We introduce a frame of latency in the rendering pipeline in order to fetch the PMs of newly visible clusters from the disk and avoid stalls in the rendering pipeline.

5.1 View-Dependent Refinement

Our algorithm maintains an active cluster list (ACL), which is a cut in the tree representing the cluster hierarchy. During each frame, we refine the ACL based on the current viewing parameters. Specifically, we traverse the ACL and compute the *error-bound* for each cluster. Each cluster on the active front whose *error-bound* is less than the *min-error* of its *error-range* is split because the PM cannot meet the *error-bound*. Similarly, sibling clusters that have a greater *error-bound* than *max-error* are collapsed. Each PM in the ACL is refined prior to being rendered by choosing the mesh in the PM mesh sequence with the lowest face count that meets the *error-bound*.

5.2 Handling Cluster Dependencies

Our simplification algorithm introduces dependencies between the clusters so that we can simplify their boundaries during the preprocess. We use these dependencies to generate a crack-free simplification at runtime. Cluster-collapses occur to reduce the polygon

count in the current refinement. However, prior to collapsing a pair of sibling clusters we must check the parent's dependencies. If the children of any dependent clusters cannot also be collapsed, then the initial cluster collapse cannot occur. These checks occur at the cluster level and are relatively inexpensive.

5.3 Rendering Algorithm

Our rendering algorithm combines occlusion culling, view-dependent refinement, and out-of-core rendering. We first explain the sequence of operations performed during each frame for occlusion culling and view-dependent refinement. After that we present the algorithm for out-of-core rendering by introducing one frame of latency. This extra frame time is used to load the PMs of newly visible clusters from the disk.

5.4 Conservative Occlusion Culling

We exploit temporal coherence in occlusion culling. Each frame our algorithm computes a potentially visible set of clusters (PVS) and a newly visible set (NVS), which is a subset of the PVS. The PVS for frame i is denoted as PVS_i and the NVS as NVS_i. An occlusion representation (OR_i), represented as a depth buffer, is computed by rendering PVS_{i-1} as an occluder set. Using OR_i we determine PVS_i. The overall rendering algorithm is:

Step 1: Refine ACL. The ACL is refined as described in Sec. 5.1 based on the camera parameters for frame i.

Step 2: Render PVS_{i-1} to compute OR_i: We refine clusters in PVS_{i-1} based on the viewpoint, compute a simplification for each cluster and render them to compute OR_i. OR_i is represented as a depth map that is used for occlusion culling. These clusters are rendered to both the depth and color buffers.

Step 3: Compute NVS_i and PVS_i: The bounding boxes of all the clusters in the ACL are tested for occlusion against OR_i. This test is performed with hardware occlusion queries at the resolution of image precision. PVS_i contains all the clusters with visible bounding boxes, while NVS_i contains the clusters with visible bounding boxes that were not in PVS_{i-1}.

Step 4: Render NVS_i: The PMs of clusters in NVS_i are refined and rendered, generating the final image for frame i.

5.5 Out-of-Core Rendering

Our algorithm works with a fixed memory footprint of main memory and graphics card memory. The entire cluster hierarchy is in main memory and we fetch the PMs of the clusters needed for the current frame as well as prefetch some PMs of clusters for subsequent frames. Additionally, we store the vertices and faces of active clusters in GPU memory. By rendering the primitives directly from GPU memory, AGP bus bandwidth requirement is reduced and we obtain an increased triangle throughput.

Our out-of-core rendering algorithm uses the paging mechanism of the operating system by mapping a file into read-only logical address space [26]. To fully take advantage of this mechanism, we store our view-dependent representation in a memory coherent manner, as described in Sec 4.3. We use two separate threads: a fetch thread and a main thread. The fetch thread is used to prepare data for PMs that are likely to be used in the future. This thread provides hints to OS and converts the PM data to the runtime format while a main thread handles refinement, occlusion culling, and rendering.

5.5.1 LOD Prefetching

When we update clusters in the ACL by performing cluster-collapse and cluster-split operations, the children and parent clusters are activated. The PMs of these clusters may not be loaded in the main memory and GPU memory. This can stall the rendering pipeline. To prevent these stalls whenever a cluster is added to the ACL we prefetch its parent and children clusters. Thus, we attempt to keep one level of the hierarchy above and below the current ACL in main memory.

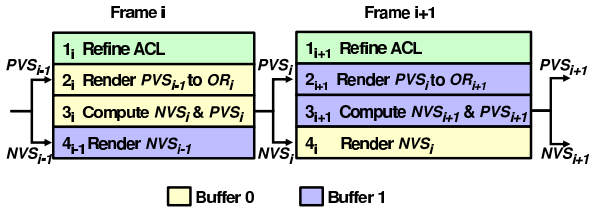

Figure 6: Our Rendering Pipeline. *In frame i occlusion culling is performed for frame i but the final image for frame $i-1$ is displayed. This allows extra time for loading the PMs of newly visible clusters. Two off-screen buffers facilitate this interleaving of successive frames. The partial rendering of frame i is stored in one buffer while occlusion culling for frame $i+1$ occurs in the other buffer.*

5.5.2 Visibility Fetching

Predicting visibility or occlusion events is difficult, especially in complex models with high depth complexity and small holes. As a result, our algorithm introduces a frame of latency in the rendering pipeline and fetches the PMs of the newly visible cluster in the ACL from the disk.

In our rendering algorithm visibility events are detected in Step 3, and the newly visible clusters are added to NVS_i (Sec. 5.4). These clusters are then rendered in Step 4, which will likely not allow enough time to load these clusters without stalling. Step 2, rendering OR_i, is the most time consuming step of the rendering algorithm. Therefore, we delay the rendering of NVS_i until the end of Step 2 of the next frame and perform rendering of PVS_{i-1} and fetching PMs from the disk in parallel using a fetch thread. Our rendering pipeline is reordered to include a frame of latency thereby increasing the time allowed to load a cluster to avoid stall.

During frame i we perform Steps 1 through 3 of the rendering algorithm with the camera parameters for frame i. However, we perform Step 4 for frame $i-1$ and generate the final image for frame $i-1$. The overall pipeline of the algorithm proceeds as: 1_i, 2_i, 3_i, 4_{i-1}, 1_{i+1}, 2_{i+1}, 3_{i+1}, 4_i, ..., where n_j refers to Step n of frame j (as shown in Fig. 6). In this reordered pipeline, the PM of a cluster in NVS_i can be loaded anytime from step 3_i until step 2_{i+1} when it must be rendered for OR_{i+1}. However, rendering the already loaded PMs in OR consumes most of the frame time, so there is almost a full frame time to load NVS.

To implement this pipeline, we use a pair of off-screen buffers. One buffer holds the partial rendering of a frame from Step 2 so that it may be composited with the newly visible clusters in Step 4 the following frame. Fig. 6 illustrates how the buffers are used for two consecutive frames.

6 Implementation and Performance

In this section we describe our implementation and highlight its performance on massive models.

6.1 Implementation

We have implemented our out-of-core simplification and runtime system on a dual 2.4GHz Pentium-IV PC, with 1GB of RAM and a GeForce Ultra FX 5950 GPU with 128MB of video memory. Our system runs on Windows XP and uses the operating system's virtual memory through memory mapped files. Windows XP imposes a 2GB limitation for mapping a file to user-addressable address space. We overcome this limitation by mapping a 32MB portion of the file at a time and remapping when a PM is required from outside this range.

We use the METIS graph partitioning library [23] for cluster computation. We use NVIDIA OpenGL extension GL_NV_occlusion_query to perform occlusion queries.

We achieve high throughput from graphics cards by storing the mesh data on the GPU, thereby reducing the data transferred to the GPU each frame. We use the GL_ARB_vertex_buffer_object OpenGL extension that performs GPU memory management for

Model	PP	Isosurface	St. Matthew
Triangles (M)	12.2	100	372
Original Size (MB)	485	2,543	9,611
Num Clusters (K)	5.8	32	65
Memory footprint used (MB)	32	256	512
Size of CHPM (MB)	625	3,726	13,992
Processing time (min)	64	350	2,369

Table 1: Preprocess *Preprocess timings and storage requirements for test models. We are able to compute a CHPM for each environment using our out-of-core algorithm and a memory footprint of* $32 - 512$*MB.*

both the vertex and the face arrays. However, we generate some new faces during each frame by performing vertex splits or edge collapse operations during local refinement of each PM. In practice, only a small number of PMs require refinement during each frame. As a result, we only transmit the faces of these PMs to the GPU and the other faces are cached in the GPU memory.

6.2 Massive Models

Our algorithm has been applied to three complex models, a coal-fired power plant composed of more than 12 million polygons and 1200 objects (Fig. 3), the St. Matthew model consisting of a single 372 million polygon object (Fig. 2), and an isosurface model consisting of 100 million polygons also originally represented as a single object (Fig. 1). The details of these models are shown in Table 1. We generated paths in each of our test models and used them to test the performance of our algorithm. These paths are shown in the accompanying video.

6.3 Performance

We have applied our unoptimized out-of-core CHPM generation preprocess to each of the models. Table 1 presents preprocessing time for each model on the PC. Hierarchical simplification takes approximately 85% of the preprocess time. The remainder of the time is dominated by the face pass of the cluster decomposition.

We are able to render all these models at interactive rates (10-30 frames per second) on a single PC. Fig. 7 illustrates the performance of the system on a complex path in isosurface model. Table 2 shows the average frame rate, front size, and number of edge collapse and vertex split operations performed for paths in each of our test models. Table 3 shows the average breakdown of the frame time for each model. Rendering costs dominate the frame time.

6.3.1 Out-of-core

Our system relies on the underlying operating systems virtual memory management for paging of PMs and, as discussed in Sec. 5.5.2, uses a frame of latency to hide load times of newly visible clusters. The frame rates of a sample path of the isosurface model are shown in Fig. 7.

6.3.2 Occlusion culling

Occlusion culling is very important for rendering models with high depth complexity such as the power plant and isosurface models. Fig. 7 highlights the benefit of occlusion culling by comparing the frame rate of our system over a path with occlusion culling enabled and disabled. On average the frame rate is $25 - 55\%$ higher when occlusion culling enabled.

7 Comparisons and Limitations

In this section, we analyze the performance of Quick-VDR. We also highlight the benefits over prior approaches and describe some of its limitations.

Refinement Cost of CHPMs vs. Vertex Hierarchies: Most of the earlier algorithms for view-dependent simplification use a vertex hierarchy. These algorithm compute an active vertex front in the hierarchy and handle dependencies at the vertex or edge level.

We compared the refinement cost of CHPM with an implementation of a vertex hierarchy (VDPM) for an isosurface with about 1M triangles (see Table 4). We have observed that CHPM refinement

Model	POE	Avg FPS	Avg Front Size	Avg # Ecol/Vsplit	Avg # Tri(K)
Power plant	1	33	1219	410	403
Isosurface	40	16	937	164	765
St. Matthew	1	17	366	802	771

Table 2: Runtime Performance: *We highlight the performance on the three benchmarks. The average frame rate, average front size, and average number of edge collapse and vertex splits are presented for a sample path in each model. All the data is acquired at* 512×512 *resolution. We use a* 400*MB memory footprint for the power plant model and* 600*MB for other models.*

Model	Refining	Occlusion Culling	Rendering
Power plant	3.6%	10.7%	85.7%
Isosurface	1.9%	1.9%	96.2%
St. Matthew	4.2%	1.4%	94.4%

Table 3: Runtime Timing Breakdown. *This table shows the percentage of frame time spent on the three major computations of the runtime algorithm. More than* 85% *of the time is spent in rendering the potential occluders and visible primitives.*

Method	Vertex Hierarchy	CHPM
Num Dependency Checks	4.2M	223
Refinement Time(ms)	1,221	32

Table 4: Refinement Performance. *A comparison of refinement cost between a CHPM and vertex hierarchy in a* 1*M triangle isosurface. This table measures the time to fully refine the mesh from the base mesh.*

cost is one-two orders of magnitude lower, even without occlusion culling. This lowered cost is mainly due to the coarse-grained refinement operations and handling dependencies at the cluster level.

Storage Overhead: Our CHPM implementation requires on average 88MB per million vertices. This is low compared to Hoppe's [20] VDPM representation (224MB) and XFastMesh (108MB) [8]. Moreover, CHPM can easily represent models with non-manifold topologies.

Out-of-Core Computation: Our out-of-core preprocess is able to construct a CHPM from large datasets using a constant-sized memory footprint. Moreover, our hierarchical simplification algorithm produces nearly in-core quality progressive meshes and preserves the mesh connectivity.

Quick-VDR introduces a frame of latency to fetch PMs of the newly visible cluster from the disk. This is needed to take into account the visibility events that can occur between successive frames. Earlier algorithms that combine visibility computations with out-of-core rendering decompose large CAD environments into rectangular cells and do not introduce additional latency [1, 6]. However, it may not be easy to decompose large isosurfaces for visibility-based prefetching. The iWalk system [6, 7] performs approximate occlusion culling.

Limitations The main limitation of our approach is one frame of latency in the rendering pipeline. Our visibility fetching scheme ameliorates the stalling problem but does not eliminate the problem. The set of dynamic LODs or simplifications computed by the CHPM could be smaller than the ones computed using a full vertex hierarchy. This is because of our decomposition of the model into a cluster and representation of each cluster as a linear sequence of edge collapses. Moreover, cluster dependencies that force us to perform additional cluster-split operations might cause popping artifacts.

Our occlusion culling algorithm assumes high temporal coherence between successive frames. Its effectiveness varies as a function of coherence between successive frames.

8 Conclusion and Future Work

We have presented Quick-VDR, a novel algorithm for view-dependent rendering of massive models. Quick-VDR represents the scene using a CHPM and provides us with the ability to perform coarse-grained as well as fine-grained refinement. It sig-

nificantly reduces the refinement cost as compared to earlier approaches based on vertex hierarchies. The cluster hierarchy enables occlusion culling and out-of-core rendering. Quick-VDR relies on an out-of-core algorithm to compute a CHPM and combines view-dependent simplification, occlusion culling and out-of-core rendering. Quick-VDR has been applied to massive models with a few hundred million triangles and can render them at interactive rates using commodity graphics systems.

Many avenues for future work lie ahead. In addition to overcoming the limitations of the current approach, we would like to use geomorphing [19] to smooth the popping artifacts. Moreover, the two level refinement could be extended to consider view-dependent effects such as specular highlights, silhouettes and shadows. We would also like to explore methods for predicting occlusion events so that we can further improve our out-of-core computation and eliminate the frame of latency in the rendering pipeline. Finally, we would like to use the CHPM representation for multiresolution compression and collision detection between massive models.

ACKNOWLEDGMENTS

Our work was supported in part by ARO Contracts DAAD19-02-1-0390 and W911NF-04-1-0088, NSF awards 0400134 and 0118743, ONR Contracts N00014-01-1-0067 and N00014-01-1-0496, DARPA Contract N61339-04-C-0043 and Intel.

The St. Matthew statue are courtesy of the Digital Michelangelo Project at Stanford University. The isosurface model is courtesy of the LLNL ASCI VIEWS Visualization project. The power plant environment is courtesy of an anonymous donor. We would like to thank Naga Govindaraju, Martin Isenburg, Stefan Gumhold, Mark Duchaineau, Peter Lindstrom, and the members of UNC Walkthrough group for their useful discussions and support.

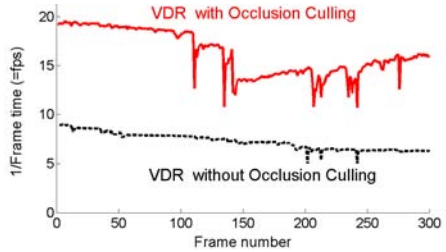

Figure 7: Frame Rate in Isosurface Model. *Frame rates are shown for a sample path using our system. For comparison we show our system without occlusion culling.*

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*, pages 199–206, 1999.

[2] S. Arya and D. M. Mount. Approximate nearest neighbor queries in fixed dimensions. In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, pages 271–280, 1993.

[3] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. Planet-sized batched dynamic adaptive meshes (p-bdam). In *IEEE Visualization*, 2003.

[4] P. Cignoni, C. Montani, C. Rocchini, and R. Scopigno. External memory management and simplification of huge meshes. In *IEEE Transaction on Visualization and Computer Graphics*, 2003.

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

[6] W. Correa, J. Klosowski, and C. Silva. iwalk: Interactive out-of-core rendering of large models. In *Technical Report TR-653-02, Princeton University*, 2002.

[7] Wagner T. Corrêa, James T. Klosowski, and Cláudio T. Silva. Visibility-based prefetching for interactive out-of-core rendering. In *Proceedings of PVG 2003 (6th IEEE Symposium on Parallel and Large-Data Visualization and Graphics)*, pages 1–8, 2003.

[8] C. DeCoro and R. Pajarola. Xfastmesh: View-dependent meshing from external memory. In *IEEE Visualization*, 2002.

[9] J. El-Sana and E. Bachmat. Optimized view-dependent rendering for large polygonal dataset. *IEEE Visualization*, pages 77–84, 2002.

[10] J. El-Sana and Y.-J. Chiang. External memory view-dependent simplification. *Computer Graphics Forum*, 19(3):139–150, 2000.

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

[12] J. El-Sana and A. Varshney. Generalized view-dependent simplification. *Computer Graphics Forum*, pages C83–C94, 1999.

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

[14] 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.

[15] M. Garland and P. Heckbert. Surface simplification using quadric error bounds. *Proc. of ACM SIGGRAPH*, pages 209–216, 1997.

[16] N. Govindaraju, A. Sud, S. Yoon, and D. Manocha. Interactive visibility culling in complex environments with occlusion-switches. *Proc. of ACM Symposium on Interactive 3D Graphics*, pages 103–112, 2003.

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

[18] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Super Computing*, 1995.

[19] H. Hoppe. Progressive meshes. In *Proc. of ACM SIGGRAPH*, pages 99–108, 1996.

[20] H. Hoppe. View dependent refinement of progressive meshes. In *ACM SIGGRAPH Conference Proceedings*, pages 189–198, 1997.

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

[22] M. Isenburg and S. Gumhold. Out-of-core compression for gigantic polygon meshes. In *ACM Trans. on Graphics (Proc. of ACM SIGGRAPH)*, volume 22, 2003.

[23] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 1998.

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

[25] P. Lindstrom. Out-of-core construction and visualization of multiresolution surfaces. In *ACM Symposium on Interactive 3D Graphics*, 2003.

[26] P. Lindstrom and V. Pascucci. Terrain simplification simplified: A general framework for view-dependent out-of-core visualization. In *IEEE Transaction on Visualization and Computer Graphics*, pages 239–254, 2002.

[27] P. Lindstrom and C.T. Silva. A memory insensitive technique for large model simplification. In *Proc. of IEEE Visualization*, pages 121–126, 2001.

[28] D. Luebke and C. Erikson. View-dependent simplification of arbitrary polygon environments. In *Proc. of ACM SIGGRAPH*, 1997.

[29] R. Pajarola. Fastmesh: Efficient view-dependent mesh. In *Proc. of Pacific Graphics*, pages 22–30, 2001.

[30] Chris Prince. Progressive meshes for large models of arbitrary topology. Master's thesis, University of Washington, 2000.

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

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

[33] E. Shaffer and Michael Garland. Effient adaptive simplification of massive meshes. In *Proc. of IEEE Visualization*, 2001.

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

[35] 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.

[36] S. Yoon, B. Salomon, and D. Manocha. Interactive view-dependent rendering with conservative occlusion culling in complex environments. *Proc. of IEEE Visualization*, 2003.

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