Out-of-Core Rendering of Massive Geometric Environments

Gokul Varadhan Dinesh Manocha
Department of Computer Science
University of North Carolina at Chapel Hill

Abstract

We present an external memory algorithm for fast display of very large and complex geometric environments. We represent the model using a scene graph and employ different culling techniques for rendering acceleration. Our algorithm uses a parallel approach to render the scene as well as fetch objects from the disk in a synchronous manner. We present a novel prioritized prefetching technique that takes into account LOD-switching and visibility-based events between successive frames. We have applied our algorithm to large gigabyte sized environments that are composed of thousands of objects and tens of millions of polygons. The memory overhead of our algorithm is output sensitive and is typically tens of megabytes. In practice, our approach scales with the model sizes, and its rendering performance is comparable to that of an in-core algorithm.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling - Curve, surface, solid, and object representations

Keywords: External memory, large datasets, walkthroughs, visibility, LODs, prefetching

1 INTRODUCTION

Recent advances in acquisition and computed-aided design technologies have resulted in large databases of complex geometric models. These datasets are represented using polygons or higher order surfaces. Large gigabyte sized models composed of millions of primitives are commonly used to represent architectural buildings, urban datasets, complex CAD structures or real-world environments. The enormous size of these environments poses a number of challenges in terms of storage overhead, interactive display and manipulation on current graphics systems.

Given the complexity of these massive models, a number of acceleration techniques that limit the number of primitives rendered at runtime have been proposed. These include visibility culling, model simplification and use of sample-based representations. Many of the resulting algorithms compute additional data structures for faster rendering like levels-of-detail or multi-resolution representations, image or sample-based approximations or a separate occluder database. All these 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 environments that can be partitioned into cells or utilize view-dependent simplification algorithms. However, these approaches are not directly applicable to very large, general and complex environments that are composed of tens of millions of primitives.

Main Results: We present an external memory algorithm for fast display of massive geometric environments. We represent the model using a scene graph and precompute bounding boxes, levels-of-detail (LODs) along with error metrics for each node in the scene graph. At runtime the algorithm traverses the scene graph and performs different culling techniques, including visibility culling and simplification culling, to compute the *front* in the scene graph. Our algorithm uses a combination of parallel fetching and prefetching techniques to load the visible objects or their LODs from secondary storage. The prefetching algorithm takes into account LOD-switching and visibility events that change the front between successive frames. Moreover, it employs a prioritized scheme to handle very large datasets and front sizes at interactive rates. The runtime memory overhead of our algorithm is output sensitive and varies as a function of the front size. Some of the key features of our approach include:

- An out-of-core algorithm for rendering massive environments, whose performance is comparable to that of an in-core algorithm.
- A data representation that decouples the representation of the scene graph from the actual primitives corresponding to each node's LOD.
- A prioritized prefetching algorithm that takes into account LOD-switching and visibility events and scales with the model size.
- A replacement policy that reduces the number of misses.

We have applied our algorithm to large environments composed of thousands of objects and tens of millions of polygons. The resulting scene graph sizes vary from hundreds of megabytes to a few gigabytes. Our out-of-core rendering algorithm typically uses a memory footprint of tens of megabytes and can render the models with very little or no loss in the frame rate, as compared to an in-core rendering algorithm.

Organization: The rest of the paper is organized in the following manner. We give a brief survey of previous work on interactive display of large datasets and out-of-core rendering algorithms in Section 2. We present our scene graph representation in Section 3 along with different culling algorithms used for faster display. Section 4 describes the out-of-core rendering algorithm, including a prioritized prefetching scheme. We highlight its performance on complex datasets in Section 5. In Section 6, we conclude and outline areas for future work.

2 RELATED WORK

In this section, we give a brief overview of previous work on interactive display of large models and out-of-core algorithms in computer graphics, GIS and computational geometry.

2.1 Interactive display of large models

Different approaches have been proposed to accelerate the rendering of large datasets. These are based on model simplification, visibility culling, and using image-based representations. Image-based impostors are typically used to replace geometry distant from the viewpoint and thereby speed up the frame rate [2, 1, 26, 30, 32]. Impostors can be combined with levels-of-detail (LODs) and occlusion culling using a cell based decomposition of the model [1].

The UC Berkeley Architecture Walkthrough system [19] combined hierarchical algorithms with visibility computations and LODs for architectural models. The BRUSH system [31] used LODs with hierarchical representation for large mechanical and architectural models. The QSplat system [29] uses a single data structure that combines view frustum culling, backface culling and LOD selection with point rendering for progressive display of large meshes at interactive rates. The IRIS Performer [28], a high performance library, used a hierarchical representation to organize the model into smaller parts, each of which had an associated bounding volume. Erikson et al. [16] used a hierarchy of levels-of-detail to accelerate the rendering of large geometric datasets. The GigaWalk system used a parallel rendering algorithm that combines occlusion culling with hierarchical levels-of-detail [5]. Govindraju

{varadhan,dm}@cs.unc.edu
http://gamma.cs.unc.edu/ooc/

IEEE Visualization 2002 Oct. 27 - Nov. 1, 2002, Boston, MA, USA
0-7803-7498-3/02/$17.00 © 2002 IEEE

et al. [21] have presented an improved occlusion algorithm that uses three graphics pipelines.

2.2 Out-of-Core Algorithms

There has been lot of work on out-of-core or external-memory algorithms in computational geometry and related areas [7, 36]. Nodine et al. [27] presented an algorithm for efficient use of disk blocks to perform graph searching on graphs that are too large to fit in internal memory.

Funkhouser et al. [20, 18] described techniques for managing large amounts of data in the context of an adaptive display algorithm used to maintain interactive frame rates. They employed a real-time memory management algorithm for swapping objects in and out of memory, based on a spatial subdivision of architectural models, as the observer moves through the model. Aliaga et al. [1] presented a system for interactive rendering of complex models that can be easily partitioned into virtual cells. It included prefetching and data management schemes for models that are larger than available memory. However, no good algorithms are known for automatic decomposition of a large model into cells. Furthermore, the use of image-based representations can lead to popping and disocclusion artifacts.

[23, 24, 33, 6, 10] have presented out-of-core algorithms for simplification of large models. Their focus is on offline simplification of very large scanned or related datasets. We could use these simplification algorithms to precompute levels-of-detail (LODs) & hierarchical levels-of-detail (HLODs) of objects in our model.

El-Sana and Chiang [13] presented an external-memory algorithm to support view-dependent simplification of datasets that do not fit in main memory. They have demonstrated its performance on models consisting of a few hundreds of thousand triangles that take tens of megabytes of storage. Although view-dependent simplification algorithms [22, 25, 37, 14] are elegant and work well for spatially large objects, they may impose significant overhead during visualization especially for scenes composed of tens of thousands of objects. Instead of choosing an LOD per visible object, view-dependent algorithms may query every active vertex or edge of every visible object. In contrast with these approaches, our algorithm uses static LODs because of simplicity and low runtime overhead, and accepts their limitations in terms of potential “popping” artifacts that can occur as we switch between different LODs at runtime.

A number of methods for out-of-core visualization including isosurface extraction, and rendering of large unstructured grids have been proposed [4, 8, 17]. [35, 12] presented application-controlled segmentation & paging methods for out-of-core visualization of computational fluid dynamics (CFD) data.

[34] presented a number of techniques such as indexing, caching, and prefetching to improve the performance of walkthrough of a very large virtual environment. Their system supports view frustum culling, but does not support levels-of-detail. [3] presented a system for interactive visualization of aircraft and power generation engines. Their system supports LOD and dynamic loading. [11] present a system for interactive out-of-core rendering of large models.

[9] proposed multi-resolution caching and prefetching mechanisms to support virtual walkthrough applications in the distributed environment. Our rendering algorithm in addition to supporting multi-resolution object models also employs view frustum culling and simplification culling [16].

3 SCENE REPRESENTATION

In this section, we describe our scene representation. Our rendering algorithm uses a scene graph based representation along with static, precomputed LODs. We also discuss different culling techniques used to limit the number of primitives that are rendered at runtime.

3.1 Scene Representation

Given a large environment composed of a number of objects, we represent it using a scene graph. We assume that the scene graph has been constructed using space partitioning or clustering-based approaches. The leaf nodes in the scene graph represent original objects in the scene. The algorithm also precomputes static levels-of-detail (LODs) for each object and associates them with the leaf nodes. In addition to the LODs, the algorithm precomputes drastic approximations of portions of the environments and associates them with the intermediate nodes in the scene graph. In this paper, we restrict ourselves to hierarchical levels-of-detail (HLODs) [16], though our out-of-core algorithm is also directly applicable to image-based impostors [26, 1]. An HLOD of a node in the scene graph is an approximation of that node as well as its descendants. In the rest of the paper, we use the term *object-rep*, to refer to the set of approximations associated with each node in the scene graph, including different LODs, HLODs or impostors. Different object-reps associated with a node are represented as a linear array from the coarsest object-rep to the finest object-rep, including the original object-rep. They are primarily used for rendering acceleration. In addition to the object-reps, the algorithm also stores additional data structures like bounding boxes that are used by visibility culling algorithms. Moreover, it associates error metrics with each object-rep that are precomputed by the simplification algorithm.

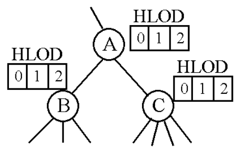


Figure 1: *Scene Graph Representation: A, B, C are nodes of the scene graph. Each node is associated with an HLOD which is a set of approximations for the node as well as its descendants. We use the term,* object-rep *to refer to each individual approximation (e.g* $A[0]$, $B[2]$, $C[1]$*).*

Notation: We use the following notation in the rest of the paper. Let $Num(N)$ be the number of object-reps associated with a node N. Let $\mathcal{PAR}(N)$ be its parent node and $\mathcal{CHD}(N)$ represent its children nodes. Moreover, we use $N[i]$ to represent the ith object-rep of node N. For an object-rep $O = N[i]$ and an integer $k > 0$, we can define its parent $\mathcal{PAR}(O)$, children $\mathcal{CHD}(O)$, k-th level ascendant $\mathcal{ASC}(O,k)$ and descendants $\mathcal{DSC}(O,k)$ in the scene graph as follows:

$$\mathcal{PAR}(N[i]) = \begin{cases} N[i-1] & i > 0 \\ P[Num(P)-1] & i = 0 \end{cases}$$

$$where\ P = \mathcal{PAR}(N)$$

$$\mathcal{CHD}(N[i]) = \begin{cases} \{N[i+1]\} & i < (Num(N)-1) \\ \cup\{C[0] \mid C \in \mathcal{CHD}(N)\} & i = (Num(N)-1) \end{cases}$$

$$\mathcal{ASC}(N[i],k) = \begin{cases} \mathcal{PAR}(N[i]) & k = 1 \\ \mathcal{ASC}(\mathcal{PAR}(N[i]),k-1) & k > 1 \end{cases}$$

$$\mathcal{DSC}(N[i],k) = \begin{cases} \mathcal{CHD}(N[i]) & k = 1 \\ \cup\{\mathcal{DSC}(C,k-1) \mid C \in \mathcal{CHD}(N)\} & k > 1 \end{cases}$$

This is illustrated in Fig 1. A is a node of the scene graph. We have $\mathcal{PAR}(A[1]) = A[0], \mathcal{CHD}(A[0]) = \{A[1]\}, \mathcal{PAR}(B[0]) = A[2]$, $\mathcal{CHD}(A[2]) = \{B[0], C[0]\}$, $\mathcal{ASC}(B[1],2) = A[2]$, $\mathcal{DSC}(A[2],2) = \{B[1], C[1]\}$.

Each object-rep $A[i]$ is associated with an object-space error metric, $E_{A[i]}$, computed by the simplification algorithm. For example, $A[0]$, $A[2]$ are the coarsest and finest objectreps of node A, respectively. Moreover, $E_{A[0]} > E_{A[1]} > E_{A[2]}$. To obtain a finer representation, as compared to $A[2]$, we descend to A's children, B and C. Based on our representation, $\{B[j], C[k]\}\ j,k \in \{0,1,2\}$ is a finer representation than $A[i]\ i \in \{0,1,2\}$. At runtime, the object-space error metric $E_{A[i]}$ of node $A[i]$ is projected to the view plane and the screen-space error bound $\epsilon_{A[i]}$ is computed.

3.2 Scene Graph Traversal

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

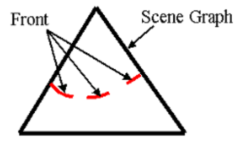


Figure 2: *Front: The exterior black triangle represents the scene graph. Front (in red) is a subset of the cut of the scene graph.*

- **View-Frustum culling**: Check whether the node's bounding box lies in the viewing frustum.
- **Simplification Culling**: Check whether any object-rep associated with that node satisfies the user-specified error bound. Among all object-reps associated with the node, the algorithm chooses the coarsest object-rep that meets the screen-space error criteria. Of all object-reps $N[i]$ that satisfy
$$\epsilon_{N[i]} < \epsilon$$
it chooses the one with smallest i. Here ϵ is the user-specified error tolerance. If such an object-rep $N[i]$ exists, the algorithm renders it and terminates the traversal [16].
- **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 object-reps that need to be rendered during the current frame. We refer to the resulting set of object-reps as the *front*. The front represents the working set of object-reps for the current frame and corresponds to a subset of a cut of the scene graph (as shown in Fig 2). 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 object-reps associated with each node. The index of the object-rep selected for each node represents an additional LOD dimension. As the viewpoint moves, the front changes in many ways. These include different events:

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

    Formally, an object-rep O can get replaced either by $\mathcal{ASC}(O,k)$ or $\mathcal{DSC}(O,k)$ for some k. These events occur when the user zooms in or out of the scene.
2. **Visibility Events**:
    - An object-rep that was in the front may disappear because the corresponding node is no longer visible.
    - An object-rep 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 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.

3.3 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 object-reps used by different culling algorithms. It stores all the object-reps on the disk and loads them on the fly based on front computation and the fetching algorithm, as described in Section 4. The resulting skeleton typically takes only a small fraction of the overall model representation.

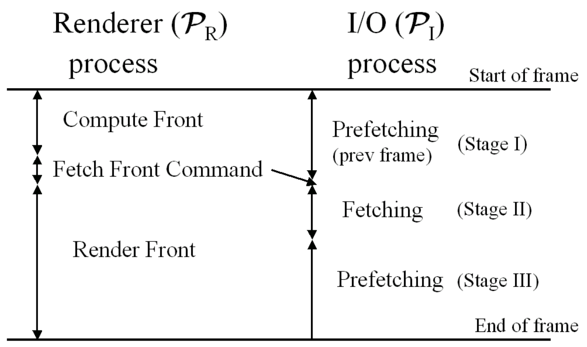


Figure 3: *Parallel Processes: Our algorithm uses two processes, one for rendering and one for I/O. The figure shows the tasks performed by each of the two processes in a given frame time*

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.

4 OUT-OF-CORE RENDERING ALGORITHM

In this section, we present our out-of-core rendering algorithm. It uses two synchronous processes, one for rendering and the second one manages the disk I/O. Each of them use the scene representation described in Section 3. We also present 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. We use terms like "hit" or "miss" to refer to the fact that a particular object-rep is present in the main memory or not, respectively.

4.1 Parallel Rendering & I/O Management

Our algorithm uses two main processes: one for scene graph traversal and rendering ($\mathcal{P}_\mathcal{R}$), and the second one for I/O management and prefetching ($\mathcal{P}_\mathcal{I}$). Both of them run in parallel and operate synchronously (see Fig 3).

$\mathcal{P}_\mathcal{R}$ traverses the scene graph and computes the front based on the current viewpoint and scene graph skeleton. During this time, $\mathcal{P}_\mathcal{I}$ continues to perform prefetching for the the previous frame (Stage I). Once $\mathcal{P}_\mathcal{R}$ finishes the front computation, it sends a fetch command to $\mathcal{P}_\mathcal{I}$. On receiving a fetch command, $\mathcal{P}_\mathcal{I}$ gets synchronized with $\mathcal{P}_\mathcal{R}$ and fetches object-reps for the current frame. The fetch command has information about the set of object-reps in the current front. $\mathcal{P}_\mathcal{I}$ divides this set into two lists:

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

$\mathcal{P}_\mathcal{I}$ starts loading the object-reps belonging to $\mathcal{L}_\mathcal{O}$ (Stage II in Fig 3). Different object-reps that constitute the front can be rendered in any order. As a result, $\mathcal{P}_\mathcal{R}$ starts rendering the object-reps that belong to $\mathcal{L}_\mathcal{I}$ and not wait till all the object-reps in $\mathcal{L}_\mathcal{O}$ are loaded in the main memory. The rendering and the loading of out-of-core object-reps proceeds in parallel. If $\mathcal{P}_\mathcal{R}$ has rendered all the object-reps belonging to $\mathcal{L}_\mathcal{I}$, it has to wait till new object-reps are loaded. Whenever $\mathcal{P}_\mathcal{I}$ loads an object-rep, it removes it from $\mathcal{L}_\mathcal{O}$ and appends it to $\mathcal{L}_\mathcal{I}$. Once $\mathcal{P}_\mathcal{I}$ has fetched all the object-reps belonging to $\mathcal{L}_\mathcal{O}$, it spends the remainder of the frame time prefetching other object-reps that may be needed for subsequent frames (Stage III).

4.2 Prefetching

In an interactive application, there is considerable coherence between successive frames. Our algorithm uses prefetching techniques to improve the hit rate. During each frame, $\mathcal{P}_\mathcal{I}$ performs prefetching during two stages of that frame, Stage I & Stage III as

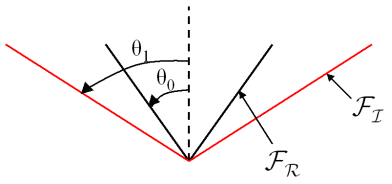


Figure 4: *Visibility Prefetching: Each of the two processes, $\mathcal{P}_{\mathcal{R}}$ & $\mathcal{P}_{\mathcal{I}}$ use two separate view frustums, $\mathcal{F}_{\mathcal{R}}$ & $\mathcal{F}_{\mathcal{I}}$ respectively. $\mathcal{F}_{\mathcal{R}}$ is in black while $\mathcal{F}_{\mathcal{I}}$ is the larger view frustum in red*

shown in Fig. 3. The goal of prefetching is to load most of the object-reps that the algorithm needs to render during subsequent frames. The prefetching is performed in parallel by $\mathcal{P}_{\mathcal{I}}$.

4.2.1 LOD Switching Events

As the user traverses through the scene, it typically moves closer to some portions of the scene and away from the other. Objects that are closer to the user need to be rendered at a finer resolution. The object-reps (O) representing these objects are replaced by their descendants ($\mathcal{DSC}(O,k)$ for some k) in the scene graph. Similarly, objects that move farther from the viewer get replaced by their ascendants ($\mathcal{ASC}(O,k)$ for some k). Moreover, the object-rep may switch between different LODs or HLODs of the same node. We avoid misses due to LOD switching events by prefetching multiple levels of ascendants and descendants of object-reps in the front.

4.2.2 Visibility Events

When the user pans across the scene, objects that were not visible earlier may become visible. In case of view frustum culling, typically these objects are outside the view frustum but close to the edge of the view frustum. We avoid such misses by letting $\mathcal{P}_{\mathcal{I}}$ use an expanded view frustum for prefetching ($\mathcal{F}_{\mathcal{I}}$) that bounds the view frustum used for rendering ($\mathcal{F}_{\mathcal{R}}$) (see Fig 4). The size of $\mathcal{F}_{\mathcal{I}}$ can be adapted based on the rate at which the user pans across the scene.

We also prioritize object-reps outside of $\mathcal{F}_{\mathcal{R}}$ that lie within $\mathcal{F}_{\mathcal{I}}$ depending on the angle they make with the line-of-sight. In particular, we assign a higher priority to object-reps that make a smaller angle. Let θ_0, θ_1 be the angles associated with the two view frustums. For an object-rep O that makes an angle θ with line of sight, we associate an angle-priority function $AngPr(O)$ as follows:

$$AngPr(O) = \frac{1}{1 + \frac{\theta \sim \theta_0}{\theta_1}}$$

$$\text{where } \theta \sim \theta_0 = \begin{cases} \theta - \theta_0 & \theta > \theta_0 \\ 0 & otherwise \end{cases}$$

We prioritize prefetching of object-reps based on the $AngPr()$ function.

4.2.3 Priority-based prefetching

One of our goals is to handle very large datasets at interactive rates. For large models, the number of object-reps in a front can be very large. For example, there are more than $6,500$ object-reps in the front corresponding to the Double Eagle model in one of the sample paths. As a result, we want to use a prefetching strategy that can handle large front sizes at interactive rates. Typically, the user's motion governs the rate at which an object-rep switches between different LODs between successive frames. To improve the hit rate, we prefetch multiple levels of ascendants and descendants of the node. Given a very large environment, whose front may consist of thousands of nodes and object-reps, it may not be possible to prefetch multiple levels of ascendants and descendants of all the object-reps in the front in the given frame time. As a result, we prioritize different object-reps in the front and the prefetching algorithm selects them based on their priority. An object-rep that is more likely to switch between the LODs is assigned a higher priority. The key issue is to predict which object-reps in the front are going to switch between different LODs. The switching takes place when the projected screen-space error for an object-rep exceeds the user-specified error tolerance (ϵ).

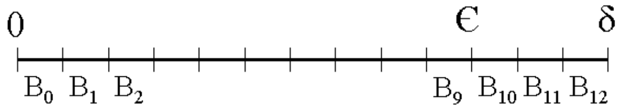


Figure 5: *Binning: The range $[0,\delta)$ is divided into B intervals, each interval corresponding to a bin as labeled in the figure. ϵ is the user-specified error tolerance. In this fig, $\epsilon = 10, \delta = 12, B = 13$*

Initially, we assign a priority to an object-rep O depending on how close its projected screen-space error, ϵ_O, is to the error tolerance ϵ. We compute the absolute difference, $| \epsilon_O - \epsilon |$, and the priority value varies as an inverse function of this difference. Moreover, we use a simple bucketing strategy, similar to bucket sort, to classify different object-reps. The classification of an object-rep O, is based on computing its projected screen-space error ϵ_O and placing it in the bin whose interval contains ϵ_O. We use a limited number of priority levels or bins, each representing a range of projected screen-space errors as shown in Fig 5. Let us assume that $\delta > \epsilon$. We divide the range $[0,\delta)$ into B intervals and assign each of the intervals to one of the B bins. Bins whose intervals are closer to ϵ have a higher priority. For example, in Fig 5, bin B_i is associated with the interval $[i, i+1)$. The bin B_{10}, whose interval contains ϵ has the highest priority and the bins $B_9, B_{11}, B_8, B_{12}, B_7, B_6, B_5, \ldots, B_0$ represent an ordering based on decreasing priorities. Unlike bucket sort, we do not sort the items within a bin. Each bin is associated with a queue. When an item is placed in a bin, it is appended to the end of the queue associated with that bin.

During each frame, we start by classifying each object-rep in the front. After the classification step, we start processing items from the bins. We select an item from the highest priority bin whose queue is non-empty. When we select an object-rep O from a bin, our goal is to ensure that when O switches between the LODs, the new LOD is in the memory. The $\mathcal{P}_{\mathcal{I}}$ process loads $\mathcal{PAR}(O)$ and $\mathcal{CHD}(O)$ and classifies them into appropriate bins. The algorithm prefetches multiple levels of ascendants and descendants. We continue to process different object-reps in this manner until the end of the frame. The pseudo-code corresponding to the classification (CLASSIFY) and prefetching (PREFETCH) steps is given below.

```
CLASSIFY($O$)
1. Calculate projected screen-space error $\epsilon_O$ of object-rep $O$
2. Place $O$ in bin $B_i$ whose interval $[i, i+1]$ contains $\epsilon_O$
```

```
PREFETCH()
1. while (not received a fetch command from $\mathcal{P}_{\mathcal{R}}$ ) do
2.     Pick object-rep $O$ from highest priority bin that is non-empty
3.     for each object-rep $X \in PAR(O) \cup CHD(O)$ do
4.         Load X into main memory
5.         CLASSIFY (X)
```

Ultimately we combine the priority values based on projected screen-space error metric along with the angle-priority function to obtain an integrated priority function, $Priority(O)$, for an object-rep O:

$$Priority(O) = \begin{cases} \epsilon_O * AngPr(O) & \epsilon_O < \epsilon \\ \epsilon_O / AngPr(O) & \epsilon_O > \epsilon \end{cases}$$

We perform the classification procedure mentioned above based on the integrated priority function. Ascendants of an object-rep in the front can have projected screen-space errors greater than ϵ and so we use $\delta > \epsilon$. By varying δ, we can vary the number of ascendants that are prefetched.

$\mathcal{P}_{\mathcal{I}}$ flushes the bins and classifies the front only when the viewpoint changes. Another issue is that object-rep sizes can vary a lot for e.g from 50 B to 20 MB in our models. Therefore, we cannot allow loading of an object-rep to be a non-preemptible operation as that can stall $\mathcal{P}_{\mathcal{R}}$. As a result, $\mathcal{P}_{\mathcal{I}}$ loads an object-rep in terms of blocks, and checks with $\mathcal{P}_{\mathcal{R}}$ after it has loaded each block.

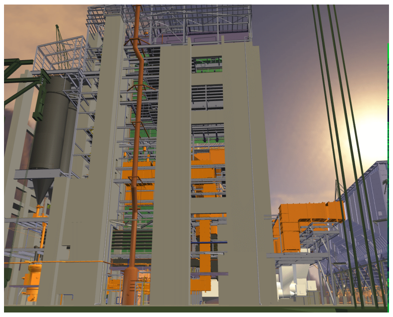


Figure 6: *Powerplant model (*PP*) is a* 0.6 *gigabyte model consisting of over* 13 *million triangles and* $1,200$ *objects (see Table 1). Our out-of-core system can render this model on a machine with* 128 *MB memory using a memory footprint of* 15 *MB and the resulting frame rate is comparable to an in-core rendering algorithm that needs over* 0.6 *GB of main memory.*

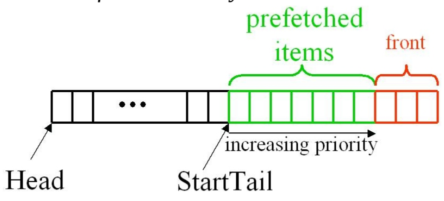


Figure 7: *Object-reps for replacement are chosen from the head of the InMemoryList. Object-reps in the front (in red) and those that have been prefetched (in green) lie beyond StartTail in the InMemoryList in an increasing order of priority. As a result they are less likely to be evicted.*

4.3 Replacement Policy

The prefetching and fetching algorithms load object-reps into the main memory from the disk. Given an upper bound on the size of main memory, any out-of-core rendering algorithm needs a mechanism to remove or replace some of the object-reps from the main memory. We use a variation of the standard LRU (least recently used) policy. It ensures that we do not remove the object-reps that belong to the front or object-reps that were recently fetched from the disk.

$\mathcal{P}_\mathcal{I}$ maintains the set of in-core object-reps in the memory in a doubly linked list, *InMemoryList*. It performs a number of operations on this list. These include:

1. At the beginning of Stage II (see Fig 3), $\mathcal{P}_\mathcal{I}$ stores a pointer *StartTail* to the tail of *InMemoryList*.
2. In course of subsequent fetching and prefetching, when $\mathcal{P}_\mathcal{I}$ accesses an object-rep, it performs the following updates:
   - If the object-rep is already in memory, it moves it from its existing position in *InMemoryList* to the new position immediately after *StartTail*.
   - If the object-rep is not already in memory, it loads it and inserts it after *StartTail*.
3. When $\mathcal{P}_\mathcal{I}$ needs to select replacement candidates, it chooses the object-rep from the *InMemoryList*, starting at the head of the list. However, if the particular object-rep is currently being rendered, it skips it and moves to the next object-rep in the linked list.

In course of fetching and prefetching operations, each time $\mathcal{P}_\mathcal{I}$ accesses an object-rep, it positions it to lie immediately after *StartTail*. As a result, the object-reps that are accessed lie beyond *StartTail* in an order which is reverse of the order in which they were accessed.

Let $\mathcal{O}_\mathcal{F}$ be the set of object-reps that correspond to the front and $\mathcal{O}_\mathcal{P}$ be the set of object-reps that have been prefetched. $\mathcal{P}_\mathcal{I}$ accesses the object-reps in $\mathcal{O}_\mathcal{F}$ during Stage II before it accesses the object-reps in $\mathcal{O}_\mathcal{P}$ during Stage III of the current frame or Stage I of next frame. Consequently, the object-reps in $\mathcal{O}_\mathcal{F}$ lie beyond the object-reps in $\mathcal{O}_\mathcal{P}$(see Fig 7). Moreover, it accesses the object-reps in $\mathcal{O}_\mathcal{P}$ in a decreasing order of priority. As a result, all the object-reps in $\mathcal{O}_\mathcal{P}$ are located beyond *StartTail* and are arranged in an increasing order of priority (see Fig 7). This ensures that when replacement object-reps are chosen from the head of *InMemoryList*, the object-reps "closest" to the front are the least likely candidates for replacement.

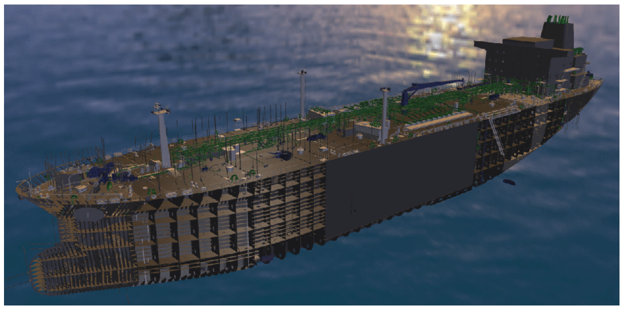


Figure 8: *Double Eagle tanker model (*DE*) is a* 3 *gigabyte environment consisting of over* 82 *million triangles and* $127,000$ *objects (see Table 1). Our out-of-core system renders this model on a machine with* 128 *MB memory using a memory footprint of* 35 *MB and the frame rate is comparable to an in-core rendering system which requires over* 3 *GB of main memory.*

5 IMPLEMENTATION AND RESULTS

We have implemented the out-of-core rendering algorithm described above and used it to render complex environments. Our current implementation uses view frustum culling and simplification culling on a scene graph that consists of LODs and HLODs. It uses two CPUs for each of the two processes, $\mathcal{P}_\mathcal{R}$ & $\mathcal{P}_\mathcal{I}$ and a single graphics pipeline. Our system uses C++, GLUT, and OpenGL. We performed tests on two machines: an SGI workstation with two 195 MHZ R10000 MIPS processors, MXI graphics board, 128 MB of main memory (*Machine 1*) and an SGI Onyx with multiple 500 MHZ R14000 MIPS processors, Infinite Reality3 graphics pipelines and 16 GB of main memory (*Machine 2*). We performed tests on *Machine 1* in order to show that our out-of-core algorithms require a limited memory footprint and on *Machine 2* to compare the performance of our out-of-core rendering algorithm with that of an in-core rendering algorithm.

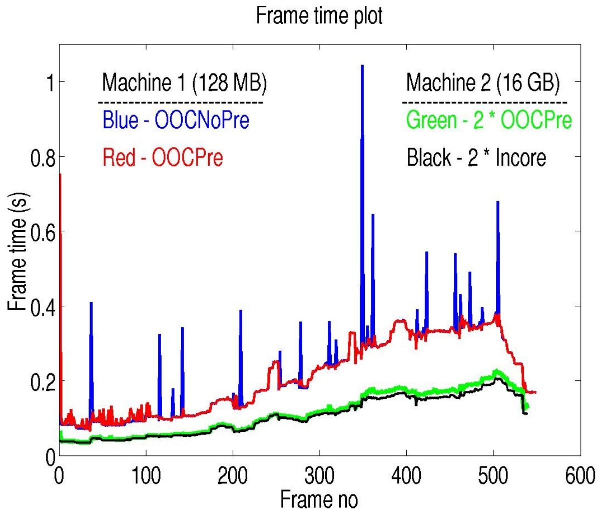
Env	**Poly**	**Objects**	**Nodes**	**Height**	**Skeleton**	**Data**
	$\times 10^6$	$\times 10^3$	$\times 10^3$		MB	MB
PP	12.2	1.2	1.8	13	0.38	596
DE	82.4	127	190	14	10	$2,974$

Table 1: *Benchmark Models: Statistics for the Powerplant model (*PP*) & Double Eagle Tanker model (*DE*).*

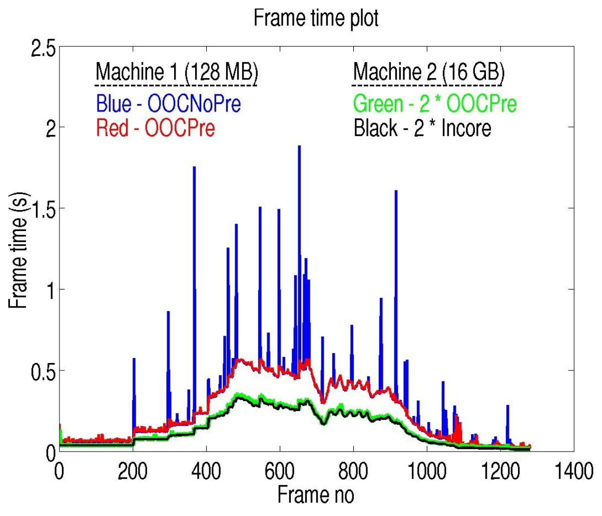
We have tested our algorithm on two large environments, a Powerplant model (*PP*) (Fig 6) and Double Eagle Tanker model (*DE*) (Fig 8). The size of PP and DE models is about 600 MB and 3 GB, respectively. The scene graphs and the LODs and HLODs associated with each node were computed using the algorithm presented in [16]. Table 1 tabulates the polygon count, number of objects, number of scene graph nodes, scene graph height, size of the scene graph skeleton, and size of scene graph data for the two models.

We generated different paths in these environments to test the performance of our algorithms. The user's motion in these paths had a lot of pan, and sudden motion. Moreover, the front size in these paths varies considerably. Typically, the front size is large when the viewer is inside an environment. We recorded multiple paths, with the same number of frames in each path, and the image quality is governed by the user-specified error tolerance. As a result, it is sufficient to measure the frame rate in order to test the performance of our algorithm for a given memory footprint. In the rest of this section, we compare the performance of three different algorithms and systems:

- *Incore*: It is the in-core system that loads the entire scene graph and all the object-reps in the main memory.



(a) Powerplant (*PP*)



(b) DoubleEagle (*DE*)

Figure 9: *Frame time: Plots in blue & red show the frame times for the out-of-core system without prefetching (*OOCNoPre*), & out-of-core system with prefetching (*OOCPre*), respectively on* Machine 1 *that has 128 MB of main memory. Plots in green & black show two times the frame times for* OOCPre *& the in-core system (*Incore*) respectively on* Machine 2 *that has 16 GB of main memory. The* Incore *system needed more than* 0.6 *and* 3 *giga-bytes of main memory for rendering the* PP *&* DE *models respectively. The* OOCNoPre *&* OOCPre *systems used a memory footprint of* 15 *MB and* 35 *MB for* PP *&* DE *respectively. We see that the* OOCPre *system matches the performance of the* Incore *system.*

- *OOCNoPre*: It initially loads the scene graph skeleton and performs parallel rendering & I/O management without any prefetching. It fetches data from the disk in Stage II, but is idle during Stages I & III (see Fig 3).
- *OOCPre*: It initially loads the skeleton and performs both parallel rendering & I/O management as well as prioritized prefetching (*OOCPre*).

Figure 9 shows the relative performance of different systems for *PP* & *DE* models, respectively. We imposed a memory footprint of 15 MB and 35 MB for *PP* and *DE* respectively. The memory footprint holds the object-reps loaded by the $\mathcal{P}_{\mathcal{I}}$ and does not account for the size of the scene graph skeleton, which is constant for each model. Fig 10 shows the front size in each frame for the two models. Front size is calculated by summing up the sizes of the object-reps in the front. We observe that the maximum front size is about 7 MB & 8 MB for *PP* & *DE* respectively. Fig 11(b) shows the front size for DE for a sample path that uses a lower error tolerance and a much higher image fidelity. In this case, the maximum front size is about 35 MB. The maximum number of object-reps in the front is about 300 for the *PP* & varies in the range 2300-6500 for the *DE*.

5.1 Parallel Rendering & I/O Management

The overall performance of the *OOCNoPre* system matches with that of the *OOCPre* system at many places in the sample paths. However, the frame time plot of *OOCNoPre* system has several spikes, which typically correspond to relatively large changes in the viewpoint. Such motions are not uncommon in a walkthrough experience, when a user is exploring new parts of the environment. It can lead to drastic changes in the front. The object-reps in the new front may not have been in the main memory and this can result in more misses, which increases the fetching time. The overall rendering or frame time doesn't get affected as long as the time to fetch object-reps in $\mathcal{L}_{\mathcal{O}}$ is less than the time to render object-reps in $\mathcal{L}_{\mathcal{I}}$. This is the main benefit of performing rendering & I/O management in parallel. However, a large number of misses can cause the fetching time to dominate the rendering time and this results in spikes in the frame time plot. Fig 10 shows the amount of data that is fetched from a disk in each frame and compares it with the front size for those paths. The *OOCNoPre* system fetches a large amount of data from the disk and this results in the slowdown or a break in the system's performance. The memory usage is equal to the amount of data stored in main memory at any given time. The memory usage of *OOCNoPre* does not exceed the memory footprint limit.

5.2 Prioritized Prefetching

Fig 9 compares the performance of *OOCPre* with *OOCNoPre* and *Incore* for the two models. They highlight the benefits of prioritized prefetching. We see that the frame time plot for *OOCPre* does not have any spikes. Moreover, the *OOCPre* system does not need to fetch a large amount of data from the disk (Fig 10) and its frame does not have major variations due to the I/O bottleneck. Moreover, the performance of *OOCPre* matches that of the in-core system, *Incore*. Also the memory usage of *OOCPre* does not exceed the memory footprint limit.

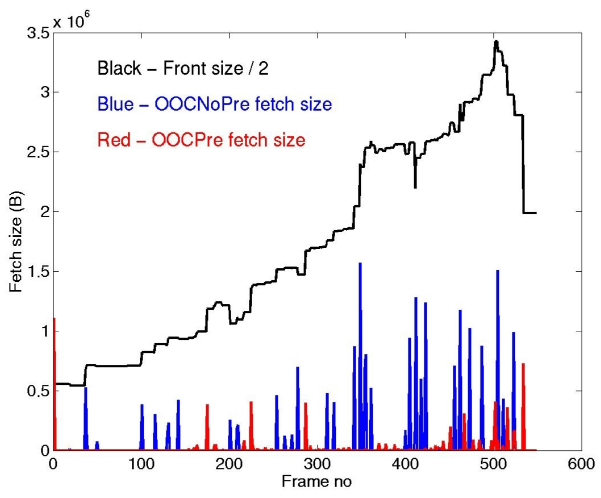
We tested the performance of our prefetching scheme for the Double Eagle model with a sample path and a smaller user-specified tolerance. Fig 11(b) shows the front size for this scenario and the fetch size for the *OOCPre* system. We notice that the maximum front size can be 35 MB and the maximum number of object-reps in the front can be 6500. For a 100 MB footprint, the resulting system fetches only a small amount of data from disk.

Inspite of large changes in the viewpoint, our out-of-core system, *OOCPre* resulted in very few misses on the sample paths in these complex environments (Fig 11(b)). Although the front can have up to 6500 object-reps, our priority-based prefetching scheme was able to accurately predict the object-reps that were likely to be used during subsequent frames. As a result, our system is able to handle very large front sizes.

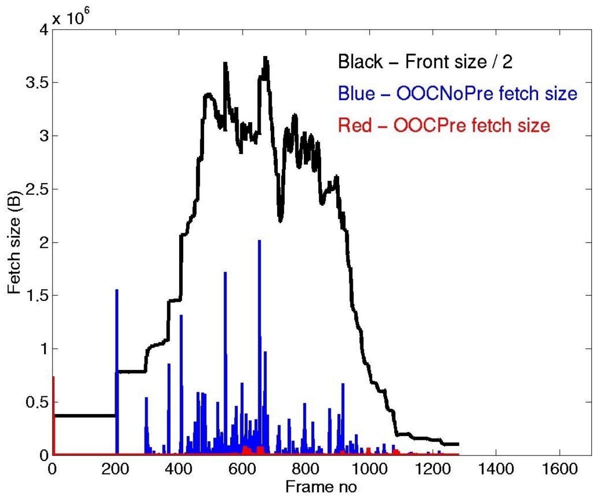
5.3 Size of Memory footprint

Fig 11 highlights the size of data that is fetched from the disk for *OOCPre* as we vary the memory footprint. Notice that as we increase the memory footprint, the amount of data that is fetched from the disk during Stage II of the I/O process, decreases substantially for *OOCPre*. As a result, there are very little or no spikes in the frame rate for the *OOCPre* system, as compared to the *OOCNoPre* system. Moreover, the *OOCPre* system matches the peak performance of the *Incore* system. We achieve this performance with a relatively small memory footprint, typically of the order of few tens of megabytes. For example, for the Double Eagle model, we achieve peak performance using a 35 MB footprint (Fig 9(b)).

The prioritized prefetching is very useful even when the system



(a) Powerplant (*PP*)



(b) Double Eagle (*DE*)

Figure 10: *Fetch size: The black graph shows the half the front size. The blue & red plots show the fetch sizes (in Stage II) for* OOCNoPre *&* OOCPre *systems on* Machine 1*, respectively. The* OOCNoPre *&* OOCPre *systems used a memory footprint of* 15 *and* 35 *MB for* PP *&* DE *respectively. For the Double eagle model the* OOCNoPre *system needed to fetch about* 2 *MB from disk whereas the* OOCPre *system had to fetch less than* 0.1 *MB for the same frame.*

is using a small memory footprint, say 8MB (Fig 11(a)). However, in such a case the frame time plot for *OOCPre* has spikes. These spikes correspond to the case when front size is close to 8 MB. Since the memory footprint was 8 MB, this implies that $\mathcal{P}_{\mathcal{I}}$ can only prefetch relatively small amount of data. In such cases, the performance of *OOCPre* is only slightly better than that of *OOCNoPre*. However when the front size is smaller than 8 MB, $\mathcal{P}_{\mathcal{I}}$ is able to prefetch sufficient number of object-reps and thereby result in few misses. This benchmark also demonstrates the effectiveness of our replacement policy which ensures that object-reps in the front and those which have been prefetched are not removed from the main memory.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we have presented an algorithm for out-of-core rendering of massive geometric environments. We represent the model using a scene graph and precompute levels-of-detail for different nodes. We use a parallel approach to render the scene as well as fetch objects from the disk in a synchronous manner. We have presented a novel prefetching technique that takes into account the LOD-based and visibility-based events, which can cause changes in the front between successive frames. The resulting algorithm has been applied to two complex environments whose size varies from few hundreds MBs to a few GBs. It scales with the model sizes and the memory requirements of the algorithm are output sensitive, typically few tens of MBs for our sample paths.

There are many avenues for future work. We would like to combine our out-of-core scheme with occlusion culling [5, 21], in addition to view frustum culling and simplification culling. Our current implementation uses pre-computed static LODs and it may be worthwhile to explore hybrid schemes that combine static LODs for small objects in the scene with view-dependent simplification of large objects. It may be useful to incorporate external-memory techniques such as blocking for efficient I/O. We would like to investigate motion prediction schemes to further improve the performance of our prefetching algorithm. We would also like to develop target-frame rate schemes that use a bounded memory footprint.

Acknowledgement

This work has been supported in part by ARO Contract DAAD19-99-1-0162, NSF awards ACI 9876914 and ACI 0118743, ONR Young Investigator Award (N00014-01-1-0067), a DOE ASCI grant, and by Intel Corporation. The Double Eagle model is courtesy of Rob Lisle, Bryan Marz, and Jack Kanakaris at Newport News Shipbuilding. The Power Plant model is courtesy of an anonymous donor. We would like to thank the members of UNC Walkthrough group for useful discussions and support. Special thanks to Shankar Krishnan and Claudio Silva for their help.

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 *ACM Symposium on Interactive 3D Graphics*, 1999.

[2] D. Aliaga and A. Lastra. Automatic image placement to provide a guaranteed frame rate. In *ACM SIGGRAPH*, 1999.

[3] Lisa Sobierajski Avila and William Schroeder. Interactive visualization of aircraft and power generation engines. In *IEEE Visualization*, pages 483–486, 1997.

[4] C. Bajaj, V. Pascucci, D. Thomson, and X. Y. Zhang. Parallel accelerated isocontouring for out-of-core visualization. In *IEEE Parallel Visualization and Graphics Symposium*, pages 87–104, 1999.

[5] W.V. Baxter, A. Sud, N.K. Govindaraju, and D. Manocha. Gigawalk: Interactive walkthrough of complex environments. In *Eurographics Rendering Workshop*, 2002.

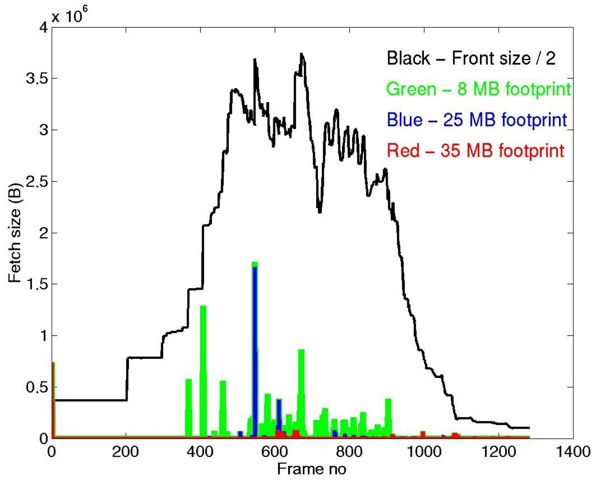
[6] F. Bernadini, J. Mittleman, and H. Rushmeier. Case study: Scanning michelangelo' florentine pieta. In *ACM SIGGRAPH 99 Course Notes Course 8*, 1999.

[7] Y.J. Chiang, M.T. Goodrich, E.F. Grove, R. Tamassia, D.E. Vengroff, and J.S. Vitter. External-memory graph algorithms. In *ACM-SIAM Symposium on Discrete Algorithms*, 1995.

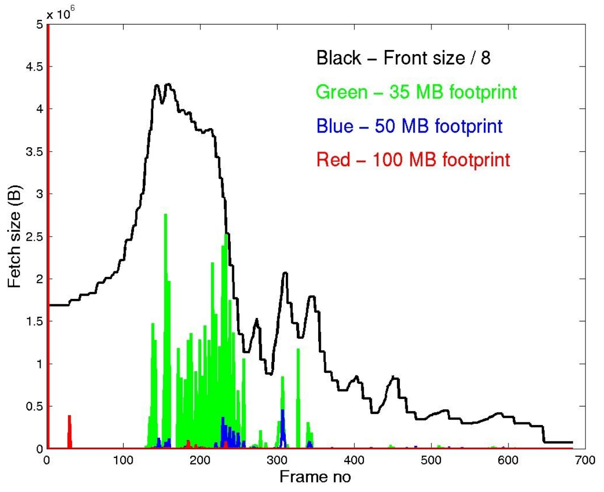
[8] Y.J. Chiang and C.T. Silva. External memory techniques for isosurface extraction in scientific visualization. In *AMS/DIMACS Workshop on External Memory Algorithms and Visualization*, 1998.

[9] J.H.P. Chim, M. Green, R.W.H. Lau, H.V. Leong, and A. Si. On caching and prefetching of virtual objects in distributed virtual environments. In *ACM Multimedia*, 1998.

[10] P. Cignoni, C. Montani, C. Rocchini, and R. Scopigno. External memory simplification of huge meshes. In *Technical Report IEI:B4-02-00, IEI, CNR, Pisa, Italy*, 2000.



(a) Double Eagle - small front



(b) Double Eagle - large front

Figure 11: *These graphs highlight the fetch sizes (in Stage II of the I/O process) used by the* OOCPre *system based on different memory footprints: In the two figures, the black plot shows half and one-eighth of the front size respectively. The green, blue & red plots show the fetch sizes corresponding to different memory footprints. We varied the front sizes and the frame rate by varying the paths and the user-specified error thresholds. The figure on the left (test performed on* Machine 1*) has a maximum front size of* 8 *MB and shows fetch sizes for different memory footprints of* 8 *MB,* 25 *MB &* 35 *MB. The figure on the right (test performed on* Machine 2*) has a maximum front size of* 35 *MB and shows fetch sizes corresponding to memory footprints of* 35 *MB,* 50 *MB &* 100 *MB.*

[11] 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.

[12] M. Cox and D. Ellsworth. Application-controlled demand paging for out-of-core visualization. In *IEEE Visualization*, pages 235–244, 1997.

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

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

[15] C. Erikson and D. Manocha. Gaps: General and automatic polygon simplification. In *ACM Symposium on Interactive 3D Graphics*, 1999.

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

[17] R. Farias and C.T. Silva. Out-of-core rendering of large unstructured grids. In *IEEE Computer Graphics and Applications*, 2001.

[18] T. Funkhouser. Database management for interactive display of large architectural models. In *Graphics Interface*, 1996.

[19] 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.

[20] Thomas A. Funkhouser and Carlo H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In James T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 247–254, August 1993.

[21] N.K. Govindaraju, A. Sud, Sung-Eui Yoon, and D. Manocha. Parallel occlusion culling for interactive walkthroughs using multiple gpus. In *UNC Technical report TR02-027*, 2002.

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

[23] P. Lindstrom. Out-of-core simplification of large polygonal models. In *ACM SIGGRAPH*, 2000.

[24] P. Lindstrom and C.T. Silva. A memory insensitive technique for large model simplification. In *IEEE Visualization*, 2001.

[25] David Luebke and Carl Erikson. View-dependent simplification of arbitrary polygonal environments. In *SIGGRAPH '97*

[26] P. Maciel and P. Shirley. Visual navigation of large environments using textured clusters. In *ACM Symposium on Interactive 3D Graphics*, pages 95–102, 1995.

[27] M.H. Nodine, M.T. Goodrich, and J.S. Vitter. Blocking for external graph searching. In *ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, 1993.

[28] John Rohlf and James Helman. IRIS performer: A high performance multiprocessing toolkit for real–Time 3D graphics. In *SIGGRAPH '94*

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

[30] G. Schaufler and W. Sturzlinger. A three dimensional image cache for virtual reality. *Computer Graphics Forum*, 15(3):C227–C235, 1996.

[31] B. Schneider, P. Borrel, J. Menon, J. Mittleman, and J. Rossignac. Brush as a walkthrough system for architectural models. In *Fifth Eurographics Workshop on Rendering*, pages 389–399, July 1994.

[32] J. Shade, D. Lischinski, D. Salesin, T. DeRose, and J. Snyder. Hierarchical image caching for accelerated walkthroughs of complex environments. In *ACM SIGGRAPH* 1996

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

[34] L. Shou, J. Chionh, Z. Huang, Y. Ruan, and K.L. Tan. Walking through a very large virtual environment in real-time. In *Proc. International Conference on Very Large Data Bases*, pages 401–410, 2001.

[35] S.-K. Ueng, C. Sikorski, and K.-L. Ma. Out-of-core streamline visualization on large unstructured meshes. In *IEEE Transactions on Visualization and Computer Graphics*, volume 3, pages 370–380, 1997.

[36] J.S. Vitter. External memory algorithms and data structures. In *External Memory Algorithms and Visualization (DIMACS Book Series, American Mathematical Society)*, 1999.

[37] J. C. Xia, J. El-Sana, and A. Varshney. Adaptive real-time level-of-detail based rendering for polygonal meshes. *IEEE Trans. Visualizat. Comput. Graph.*, 3(2):171–183, April 1997.