A Framework for the Real-Time Walkthrough of Massive Models

D. Aliaga, J. Cohen, A. Wilson, H. Zhang, C. Erikson, K. Hoff, T. Hudson, W. Stuerzlinger,
E. Baker, R. Bastos, M. Whitton, F. Brooks, D. Manocha

UNC TR# 98-013
Computer Science Department
University of North Carolina at Chapel Hill

ABSTRACT

We present a framework for rendering very large 3D models at nearly interactive rates. The framework scales with model size. Our framework can integrate multiple rendering acceleration techniques, including visibility culling, geometric levels of detail, and image-based approaches. We describe the database representation scheme for massive models used by the framework. We provide an effective pipeline to manage the allocation of system resources among different techniques. We demonstrate the system on a model of a coal-fired power plant composed of more than 15 million triangles.

CR Categories and Subject Headings: H.2 – Database Management, I.3.3 – Picture/Image Generation (Display Algorithms), I.3.4 – Graphics Utilities (Application packages), I.3.6 – Methodology and Techniques (Graphics data structures), I.3.7 – Three-Dimensional Graphics and Realism (Virtual reality), J.2 – Physical Sciences and Engineering (Engineering), J.6 – Computer-Aided Engineering (Computer-aided design)

Keywords: interactive walkthrough, framework, scalability, massive models, visibility culling, occlusion culling, levels of detail, image-based rendering, database prefetching.

1 INTRODUCTION

Computer-aided design (CAD) applications and scientific visualizations often need user-steered interactive displays (*walkthroughs*) of very complex environments. Structural and mechanical designers often create models of ships, oil platforms, spacecraft, and process plants whose complexity exceeds the interactive visualization capabilities of current graphics systems. Yet for such structures the design process, and especially the multidisciplinary design review process, benefits greatly from interactive walkthroughs.

Ideally, such a walkthrough needs to maintain a frame rate of at least 20 frames per second to avoid jerkiness. Many such massive CAD databases contain millions of primitives, and even high-end systems such as the SGI Infinite Reality Engine cannot render them interactively. Moreover, we observe model sizes to be increasing faster than hardware rendering capabilities.

A few years ago, as part of a simulation-based design team, we needed to do a real-time walkthrough of such a model, a single ship compartment modeled as some 750,000 triangles. We determined to attack this problem and for two years investigated many different known and new algorithmic approaches for accelerating rendering. Such techniques had already been extensively studied in computer graphics, computational geometry, and computer vision.

The principle for the ideal algorithmic approach is simple*: Do not even attempt to render any geometry that the user will not ultimately see.* Such techniques cull a primitive before sending it to the rendering pipeline if, for example, it is outside the view frustum, facing away from the viewpoint, too small or distant to be noticed, occluded by objects, or satisfactorily shown as a detail in a painted texture rather than as geometry. Whereas each algorithmic technique by itself reduces the number of rendered primitives, no one technique suffices for interactive walkthroughs of most massive models (more than one million primitives). Moreover, each technique achieves great speedups only for particular subsets of the primitives (e.g. distant ones, coplanar ones, models with high depth complexity). Any *general* system for interactive walkthroughs needs to integrate many such techniques.

Image 1: CAD model consisting of 15 million primitives

Choosing a 15-million-triangle model of a coal-fired electric power plant (Image 1) as our challenge model and driving problem, we have built a scalable framework for integrating many such techniques, and achieved frame rates of 5-15 frames per second for that model on an SGI Onyx with Infinite Reality graphics.

We have pursued the following goals:

- **Interactivity.** We aim at 20 frames per second.
- **Modularity.** We want to be able to incorporate and be able to substitute a variety of acceleration techniques into the framework.
- **Automaticity.** Each of the model re-representation and rendering acceleration techniques should be performed automatically without the intervention of human judgement or action.

- **Scalability.** The framework should require human set-up that is at most sublinear in the number of elements. Run-time overhead should grow sublinearly with the number of elements. The framework should operate effectively on models that cannot be contained in graphics engine memories.
- **Applicability.** The system should be applicable to real-world massive models.

1.1 System Strategy Overview

The fundamental idea in our system is to render objects "far" from a viewpoint using fast image-based techniques and to render all objects "near" the viewpoint as geometry using multiple integrated rendering acceleration techniques, including several types of culling and levels-of-detail. Our scheme limits the data required to render from any viewpoint to textured polygon impostors (for the far geometry), the near geometry, and miscellaneous rendering parameters. To implement this scheme, we partition model space into viewpoint cells and associate with each cell a cull box. The cull box defines the separation into near and far geometry. Associated with each cell is the data needed to render from any viewpoint within the cell.

The framework performs extensive preprocessing to partition the model space into viewpoint cells, to render textures that will be used to substitute for distant geometry, to construct simplified object models at multiple levels of detail, and to determine sets of possible occluders. It organizes these auxiliary data structures so that they can be prefetched into memory dynamically. It sets up the run-time environment, establishing the memory-management tactics to be used for the various acceleration techniques and the policies for dynamically allocating CPUs and renderers among the accelerators.

The first acceleration technique we use is a version of texture impostoring as a substitute for geometry. In a preprocess we partition the space of the model into equal-sized *viewpoint cells*, each defined by its centerpoint. We currently define for each cell a viewpoint emphasis function (VEF) which determines how important it is to render that cell rapidly and/or precisely. (We plan to invert this process in the next version, first defining a continuous viewpoint emphasis function over the whole model space and then using it to automatically partition the space into diverse-sized cells containing equal emphasis.) Around each cell we place a biggish cull box. We generate for each of the six inside faces of the box a *textured depth mesh* (TDM) that images all the geometry outside the box as viewed from the cell centerpoint. At run time, we cull away all the geometric primitives outside the current cull box; the textured depth meshes are displayed instead. As the viewer moves from cell to cell, simple prediction enables the relevant TDMs to be prefetched speculatively. The use of a textured depth mesh radically reduces the texture popping and other artifacts that otherwise occur as the viewpoint moves within and between cells.

For our first implementation of the power plant model, we use cubical viewpoint cells on 2 meter centers, with cubical cull boxes (26 meters) on a side. The sizing of the cull boxes is a trade-off between more geometry rendering (for big boxes) and more texture fetching (for small boxes). The current viewpoint emphasis function is 1 for the some 10,000 cells containing a walkway, stair, or ladder; 0 elsewhere. Creating the function required a fair amount of hand-specification, a process that must be automated for scalability. Image-based acceleration is only applied in the VEF=1 cells. So whereas one can view from any point inside or outside the power plant, and get an accurate view, one only gets this form of acceleration for viewpoints reachable by a person really walking in the power plant. That limitation makes the preprocessing feasible for this model and effectively matches some design review scenarios—for instance, maintenance.

For the geometry remaining inside the current cull box, we cull to the view frustum, cull back-facing triangles, and select for each object a level-of-detail-simplified surrogate model, depending upon the object's distance from the current viewpoint. In our current system, we model objects at four levels of detail.

If the current cull box has been marked as containing very many triangles (=100K in our case), we then perform occlusion culling, so that geometry hidden behind other objects is culled before rendering. For some viewpoints, this makes a big improvement; for others, the occlusion test costs more than it saves in time. The box threshold test is a heuristic for improving the chances of winning. LOD culling and occlusion culling are applied to all the geometry in the view frustum, even when the viewpoint is outside a VEF=1 cell.

1.2 Contributions

We believe the principal contribution to be the scalable framework itself, with its capability of incorporating many rendering acceleration modules. It is designed for large spatial environments with many objects. This framework includes:

- an *effective system pipeline* to manage the resources, (i.e., the CPUs, the main memory, the texture memory, the graphics engines, allocating them among the various acceleration techniques
- an *integrated database*, with a coherent representation technique, and memory management and prefetch of geometry and textures larger than hardware memory capacities, which is crucial for scalability
- *multiple hierarchies* (functional as well as spatial)
- the concept of a *general viewpoint emphasis function*, and a simple but powerful implementation of one
- *efficient and fast implementations* of, and minor extensions to, known rendering acceleration techniques.

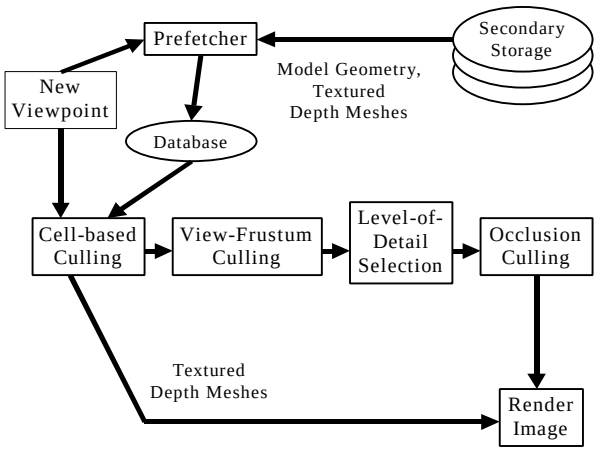


Figure 1: Run-time Pipeline. Each new viewpoint sent into the pipeline is passed to both the prefetcher and to the rendering pipeline. The viewpoint determines what geometry and meshes are retrieved from disk; it is also a parameter used by the rendering acceleration techniques.

1.3 Paper Organization

Section 2 summarizes related systems. Section 3 gives a system overview (run-time and preprocessing). Section 4 explains our database management and representation. Section 5 describes our rendering acceleration techniques in more detail. Section 6 presents details of the implementation. Section 7 gives some performance results. In Section 8, we discuss the limitations of both the current implementation and the framework. Section 9 lists the lessons learned from this work. Section 10 summarizes.

2 RELATED SYSTEMS WORK

There is an extensive literature on interactive display of large models. In this section, we briefly survey display algorithms and systems which have influenced our work by addressing the entire problem of interactively displaying large models. Algorithms for visibility culling, level-of-detail modeling, and image-based techniques are reviewed in Section 5.

A large number of systems have been developed for interactive walkthroughs. For the purposes of this paper, we can subdivide them into four general categories:

- Architectural Walkthrough Systems
- Mechanical CAD Systems
- High-Performance Libraries
- Architectures and APIs

One early work ([Clark76]) proposed using hierarchical representations of models and computing multiple *levels-of-detail* (LODs) to reduce the number of polygons rendered in each frame. This technique has been used by a number of visualization and flight simulator systems.

Several walkthrough systems for architectural models have been presented by [Airey90, Teller91, Funkho92]. These systems partition the model into cells and portals, where the division of a building into discrete rooms lends itself to a natural division of the database into cells. The UC Berkeley Building Walkthrough System [Funkho96] used a hierarchical representation of the model, along with visibility algorithms [Teller91] and LODs of objects. [Funkho92] had also proposed techniques for management and representation of large datasets; by using an adaptive display algorithm, the system was able to maintain interactive frame rates [Funkho93]. [Maciel95] expanded this framework to allow for a more general set of impostors (LODs, billboards, etc.).

The BRUSH system [Schnei94], developed at IBM, provides an interactive environment for the real-time visualization and inspection of very large mechanical CAD (and architectural) models. It uses multiple LODs of the objects in the scene [Rossignac93] and provides a number of powerful tools to navigate and manipulate the models.

IRIS Performer [Rohlf94] is a high-performance library that uses a hierarchical representation to systematically organize the model into smaller parts, each of which has an associated bounding volume. This data structure can be used to optimize culling and rendering of the model. Many other systems have been developed on top of Performer for interactive display of large environments, including an environment for real-time urban simulation [Jepson95].

Several industrial vendors, including Silicon Graphics and Hewlett-Packard, have proposed architectures and APIs (SGI OpenGL Optimizer, HP DirectModel, etc.) for interactive display of large CAD models [HP97, SGI97]. Currently, these systems provide standalone tools for simplifying polygonal models or performing visibility culling. They do not provide a framework that allows the incorporation of new algorithms or integration of different tools and subsequent application of the integrated system to a large model. So far uses of these architectures and APIs have been quite limited.

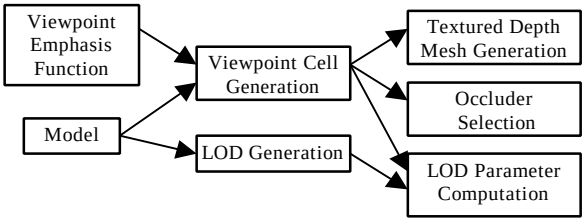


Figure 2: Preprocessing Pipeline. A model and a viewpoint emphasis function are the inputs to the preprocesses of virtual cell generation and LOD generation. These preprocesses produce textures, meshes, occluders, and LOD parameters for the run-time system.

Our research seeks both to provide such a scalable, flexible framework and to achieve new levels of performance through combination of multiple rendering acceleration techniques.

3 FRAMEWORK OVERVIEW

3.1 Run-time System

Our run-time pipeline is outlined in Figure 1. Because the entire model may not fit into memory, we employ a prefetching mechanism to load the geometry necessary for the current frame and (predicted) near-future frames from secondary storage. The prefetcher also loads the textured depth meshes necessary to replace the geometry outside the current cull box. As the model's scene graph is traversed, geometry outside the current cull box and viewing frustum is culled away, appropriate levels of detail are selected for the remaining geometry, and occlusion culling is applied. The geometry that remains is then rendered, along with this viewpoint cell's textured depth meshes.

3.2 Preprocessing

To make this run-time system possible, we must perform some amount of offline computation (Figure 2). We first use a user-supplied Viewpoint Emphasis Function to partition the model space into viewpoint cells. For each cell, we choose a cull box size and generate the textured depth meshes which will replace the geometry outside the cull box. We also generate geometric levels of detail for selected portions of the model and compute per-cell parameters that select the level of detail (to bound the amount of geometry within each cull box). We also compute for each cell the set of geometry that is likely to serve as useful occluders at run-time.

Many design environments used to develop massive models have offline procedures for gathering a current version of the model and preparing it for visualization. Our pre-computation may be performed in conjunction with these procedures. It is also not uncommon for design reviews to focus on certain portions of the model at a time, and it is possible to confine our pre-computations to those limited portions.

4 DATA REPRESENTATION AND MANAGEMENT

Data representation and database management is the major issue in integrating multiple techniques for the display of large models and, thus, is a key element in our framework. A representation should support multiple rendering acceleration techniques and be scalable across computers with differing amounts of memory. Creating such representations from a raw model should be as automatic as possible. Our solution to the representation and management problem uses a scene graph to represent the model in a bounding volume hierarchy, a viewpoint cell structure to manage the run-time walkthrough, and geometry and texture prefetching to make the system adaptable to various memory sizes.

4.1 Scene Graph

The model database is organized as a scene graph, similar to Performer or Inventor [Rohlf94]. The scene graph is a bounding volume hierarchy that supports hierarchical culling techniques. At run-time, the scene graph is traversed recursively and user-defined functions are called to process each node. Most rendering acceleration techniques are implemented as these traversal callback functions.

Organizing our database as a scene graph helps achieve our goal of modularity. New techniques can be added or old techniques modified by changing traversal callbacks. If new rendering techniques require new data, they can be added to individual nodes.

The scene graph is a bounding volume hierarchy in which each node groups spatially proximate geometry. Many real-world models have an object hierarchy which groups geometry according to non-spatial criteria, e.g. functional organization. On such models we perform a top-down spatial subdivision of polygon centers to find a usable hierarchy. An octree-style subdivision recursively subdivides the model, terminating when either a minimum number of polygons per leaf or a maximum depth is reached. We then compute a bounding volume hierarchy for the polygons: first we compute the bounding boxes of each leaf of the spatial subdivision, then we propagate these boxes to the root of the scene graph.

4.2 Viewpoint Cells

An important component of a framework for a scalable walkthrough system is a method for localizing the geometry rendered. We use a method based on viewpoint cells and cull boxes to provide this localization. The 3D space of the input model is partitioned into a set of cells. Associated with each cell is a cull box (see Figure 3 for a 2D example). The cull box is an axis-aligned box containing the cell and is considerably larger than the cell itself. When the viewpoint is in a particular cell, we cull all the geometry that lies completely outside that cell's cull box. We can use a variety of image-based techniques to represent the geometry that lies outside the cull box, as described in Section 5.

The viewpoint cells used in our framework differ from the cells and portals used in the UC Berkeley Walkthrough System [Funkho95]. The locations and sizes of those cells and portals depend upon the geometry of the model: cells correspond to rooms, and portals correspond to doors or windows. For many structural models, there are few or no interior partitions. This drives our decision to make our framework's viewpoint cells independent of the geometry in the model; rather, their sizes and locations are determined by a Viewpoint Emphasis Function, which is described in the following section.

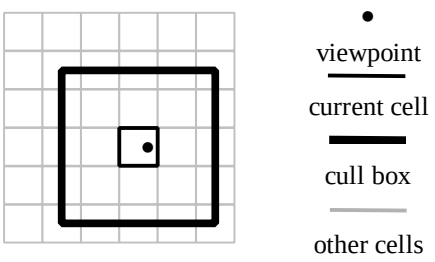


Figure 3: A set of viewpoint cells. The current viewpoint cell is in the center; its cull box is the large bold box surrounding it.

4.2.1 Viewpoint Emphasis Function

When customizing the walkthrough system, the walkthrough designer will provide not only the model database, but also the Viewpoint Emphasis Function (VEF). Given a VEF, our framework should be able to adaptively generate the set of viewpoint cells to allocate system resources to the more important areas of the model. The VEF is a scalar function defined over the 3D space of the input model. It specifies the relative importance of various portions of the model space as potential viewpoints. This function must be represented in such a way that it is possible to compute the total importance (the integral of the scalar field) within any cell-shaped subset of the model space.

4.2.2 Implementation

In our next implementation, we have chosen to initially specify a VEF as a set of 3D points with associated emphasis values. We place no limit on the size of the point set, enabling a walkthrough designer to create a VEF with as much or as little precision as desired. We use Clarkson's program `hull` to obtain a Delaunay triangulation of the point set. This triangulation results in a set of tetrahedra which linearly approximates the VEF. By triangulating with `hull`, which uses exact arithmetic, and by slightly perturbing our input points, we ensure that we can robustly obtain a meaningful, usable approximation to the VEF.

It is straightforward to compute the emphasis contained in any box-shaped region: we clip the set of tetrahedra to the walls of the box, then total the emphasis of the tetrahedra contained in the box.

A useful property of this VEF specification method is that the walkthrough designer can start off with a very simple function (perhaps one that creates a uniform cell partitioning) and add points to the specification as he learns more about the particular model or user behavior, or as the needs of the system change. For

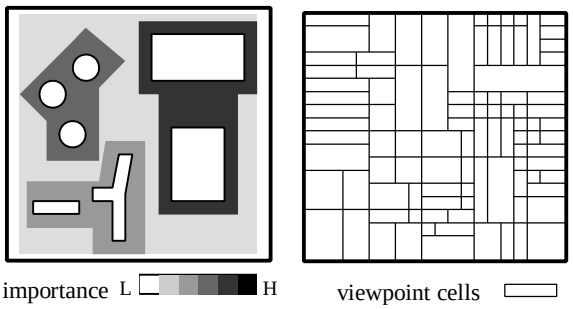


Figure 4: **(a)** Example of a viewpoint emphasis function. **(b)** A viewpoint cell distribution generated from the VEF (not drawn to scale).

instance, it is possible to place greater importance on places in the model near certain classes of objects, in certain rooms, or along certain paths.

4.2.3 Creating a Spatial Partition

Given the available resources (primarily secondary storage space), the walkthrough designer chooses the number of viewpoint cells to be generated. The framework will adaptively subdivide the model to generate a set of cells each of which has roughly equal "emphasis". The less-emphasized regions of the model space will be populated by a small number of large cells, whereas the more-emphasized regions will be populated by a large number of small cells. (Figure 4)

We construct a top-down kD tree whose leaves (axis-aligned boxes) are the viewpoint cells. To split cells, we find the plane on each axis that would result in two boxes of most nearly equal emphasis, then choose the one of these three planes that yields most nearly equal-volume boxes. This should maximize the quality of the image-based techniques we use to replace the geometry outside of the cull box.

4.3 Geometry Prefetching

To achieve our framework goal of scalability, we must address the fact that massive models are too large to fit in main memory. Fortunately, the geometry needed to render the user's view is typically only a small subset of the entire model. Partitioning the model using viewpoint cells allows us to cull away everything outside the cull box associated with the user's current cell. Therefore, only that geometry and textured depth mesh (TDM) data needed to render for the current cell must actually be in main memory. Data for other cells is paged in for those cells likely to be visited in the near future.

We employ a scheme of prefetching similar to that used in the Berkeley Walkthrough System [Funkho95]. This system exploited the structure of the architectural model by subdividing it into cells connected by portals. The *potentially visible set* (PVS) was computed by evaluating which regions of other cells were visible through portals in the user's current cell. This method can lead to rapid increases in the size of the PVS when the user turns corners and enters rooms. By creating artificial visibility constraints (the cull boxes), we can better bound the size of the PVS.

We compute the PVS for each viewpoint cell as a preprocess. We assign to each viewpoint cell a list of those objects (called Renderables) which are visible from within the cell. At run-time, these lists are used to determine which renderables and TDMs must be paged in from disk in order to render the current cell. The user's direction and speed of movement are used to predict which cells are likely to be visited in the near future. Geometry and TDMs are speculatively prefetched for these cells as well. When a cell contains an object for which multiple levels of detail have been computed, the coarsest version is loaded first. More detailed versions are paged in on demand.

Geometry is cached separately from textured depth mesh data to take advantage of cell-to-cell coherence. Since objects in the model often extend across multiple viewpoint cells, the geometry needed to render the user's current cell will typically be reused for adjacent cells. Unfortunately, no such coherence exists for the textured depth meshes due to their view-dependent nature. Caches for both geometry and TDM data are managed using least-recently-used (LRU) replacement. The pseudocode in Figure 5 summarizes the algorithm used by the prefetcher.

```
COMPUTE PREFETCH NEEDS:
Find user's current cell C
Find set of nearby cells N

IMMEDIATE NECESSITIES:
Look up geometry G required to render C
  If not loaded, page G into memory from disk

SPECULATIVE PREFETCHING:
For all cells n∈N in order of increasing distance
from eye point:
  Look up geometry G needed to render n
  Append G onto geometry prefetch queue
  Look up TDMs T visible from cell n
  Append T onto TDM prefetch queue

While C remains constant:
  Page in geometry G, TDMs T from prefetch queues
```

Figure 5: Algorithm used to prefetch model data from secondary storage

5 RENDERING ACCELERATION TECHNIQUES

5.1 Replacing Distant Geometry

Our framework strategy of replacing distant geometry with textured meshes is key to achieving our performance goal. As explained in Section 4.2, all geometry outside the cull box is culled during rendering. It is replaced by a representation that hybridizes geometry and images and closely matches the appearance of the distant geometry.

5.1.1 Previous Work

Environment maps have been used to simulate distant geometry [Greene86]. In our application the viewer walks inside a densely populated environment which prevents a direct application of this technique. QuickTime VR [Chen95] uses a cylindrical environment map to simulate the surrounding geometry, but can handle only single viewpoints.

Impostor-based methods replace distant objects with partially transparent textures mapped onto simple geometry. Whereas early techniques [Maciel95] pre-generated all textures, newer methods [Shade96, Schauf96] update the textures on demand. [Regan94] re-renders objects depending on their velocity in the image plane. Others warp the geometry [Aliaga96] to minimize effects due to transitions between geometry and textures. [Schauf97] augments the textures with depth information to avoid visibility errors.

A different class of methods creates textured meshes from color and depth information. Rendering very dense meshes is impractical in already render-bound systems, so simplified versions are used. [Darsa97] uses a mesh-decimation method to simplify the mesh. [Sillio97] sub-samples the depth information and uses edge detection to generate less complex meshes. [Pulli97] blends several textured meshes for the final rendering.

Image-based rendering methods derive new views directly from the pre-computed images. Examples of recent work include [McMill95, Levoy96, Gortle96, Mark97].

5.1.2 Textured Depth Meshes

A simple method for replacing distant geometry pre-generates images of all geometry outside the cull box as viewed from the cell center. During rendering, the images are then texture-mapped onto the faces of the cell box. Since the textures are sampled only at the cell centers, a disturbing *popping* artifact occurs when the viewpoint switches from cell to cell. To alleviate the popping, we experimented with three-dimensional image warping.

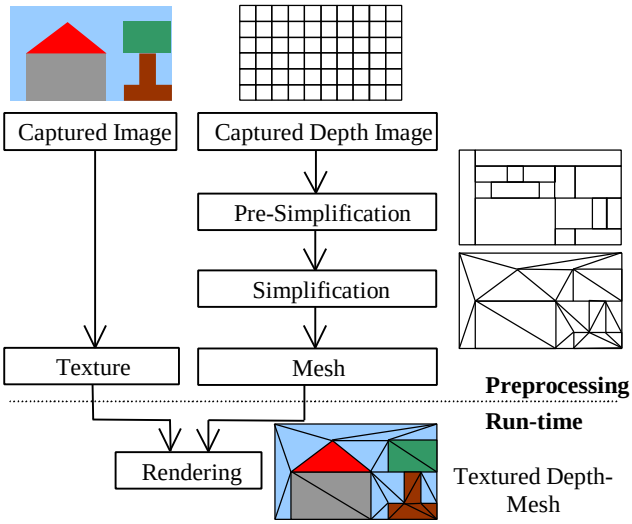


Figure 6: Offline and online components of textured-depth mesh generation. The preprocess generates a texture and a mesh from a captured image and a captured depth image; the textured depth meshes generated are displayed at run-time.

Unfortunately, image warping consumed too many resources and introduced visibility artifacts.

To combine the speed of the textured cell boxes with the correct perspective of three-dimensional image warping, we adopted an approach similar to Darsa or Sillion based on TDMs. During preprocessing, color and depth information (MxN pixels) are stored for each textured cull box face. Each face is then converted into a depth-mesh (similar to a height-field) of MxNx2 triangles. The mesh is simplified to reduce resource requirements. Rendering this simplified depth-mesh with projective textures shows the correct perspective effects. (Image 2)

This approach has multiple benefits. The visual results are good, as all major perspective effects are correct and minimal popping occurs during a transition between cells. Projective texture mapping yields a better image quality than standard texture mapping since artifacts from texture interpolation due to regional oversimplification are much less noticeable. No holes appear; instead, the mesh stretches to cover regions where no information is available (known as *skins*). As different objects can be visible in the skin regions for different cells, small popping artifacts may appear during cell transition.

5.1.3 Implementation

The system preprocesses the model by visiting each viewpoint cell and rendering all geometry outside the associated cull box from the center of the cell. Six images are generated, one for each face of the box. The color and depth information from the framebuffer is reduced to 256 colors, compressed, and stored on disk. The depth information for each texture is converted to a depth-mesh.

A general simplification algorithm with error bounds (e.g. [Garlan97]) takes too long to process a large number of such dense meshes. Therefore, we apply a pre-simplification algorithm to the depth mesh. It identifies rectangular planar regions using a greedy search method. For polygonal models, the resulting mesh has approximately 10 percent of the original polygon count. The more general simplification algorithm is then applied. No special treatment of discontinuities (as in [Sillio97]) is needed, since the simplification method uses error bounds. The simplified depth mesh is stored as a triangle-stripped mesh in a binary format.

To render this image-based representation of distant geometry, the color image and the depth mesh are read from disk and displayed using projective texture mapping. No texture coordinates are needed for projective texture mapping, which reduces storage and processing overhead. The center of projection for each texture is set to the original viewpoint for the texture (the center of the cell). See Figure 6 for an overview of the online/offline components.

5.2 Geometric Levels of Detail

After culling away the portions of the scene outside the cull box of the current virtual cell, we render the geometry that remains inside. In order to reduce the number of rendered polygons, we substitute simplified models for objects of high geometric complexity. In this section, we describe the algorithms used for computing the levels of detail.

A number of algorithms have been proposed for computing LODs in the last few years. These include algorithms based on vertex clustering [Rossig93], edge collapses [Hoppe93, Ronfar96, Cohen97], vertex removal [Turk92, Schroe92], multi-resolution analysis [Eck95], quadric error metrics [Garlan97], and simplification envelopes [Cohen96]. We can either compute static LODs based on these algorithms or use view-dependent simplification algorithms, such as those based on progressive meshes [Hoppe96, Hoppe97, Xia97] or on hierarchical dynamic simplification [Luebke97]. The view-dependent algorithms have several desirable properties. However, we have chosen static LOD for our framework due to the following reasons:

- View-dependent algorithms add both memory and CPU overhead to the system, and we are using these resources for other rendering acceleration techniques and memory management routines.
- View-dependent algorithms are relatively complex to implement and integrate into our framework compared to static simplification algorithms.
- Due to their dynamic nature, view-dependent algorithms run in immediate mode, which is two to three times slower than display lists on current Silicon Graphics systems.

5.2.1 Goals

Our criteria for an algorithm to compute static LODs are generality (handling all kinds of models), fidelity, efficiency, and

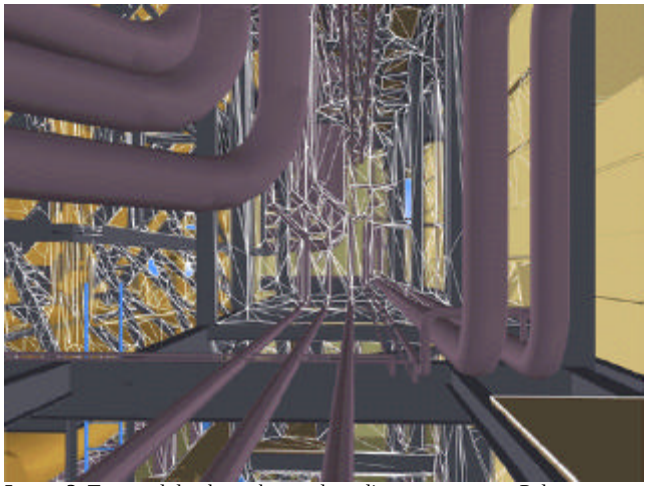


Image 2: Textured depth meshes replace distant geometry. Polygons outlined in white are part of a mesh.

the ability to obtain significant reduction in polygon count. Many simplification algorithms assume that the input model is a polygon mesh, which may not be the case in practice. Furthermore, many objects may be composed of tens of thousands of polygons and for interactive display we need to generate drastic simplifications of such objects. This requires modifying the topology to guarantee a sufficiently reduced polygon count. In our system, we will often be very close to the objects if they are rendered at all; we cannot rely on distance from the viewer to hide simplification errors, so the simplifications must look as good as possible.

5.2.2 Computation of Static LODs

One of the best simplification algorithms to date that produces reasonable low-triangle count approximations quickly while not trying to preserve topology has been proposed by [Garlan97]. It works well for a number of objects. The Garland and Heckbert algorithm has a tendency to make pieces of different objects disappear. However, when dealing with pipes, which are common in mechanical CAD models (Image 3), we want a different behavior. For example, even though the individual coils of pipes may be close together, the algorithm makes no attempt to merge an array of separate coils, but simplifies each coil independently. Rather than making the pipes completely disappear, connecting the pipes while simplifying gives a better impression of the original object. Garland and Heckbert's algorithm requires the user to input an error threshold to compute virtual edge pairs and determine the possible number of topology simplifying collapse operations. This error threshold is conceptually intuitive, but it may be hard to come up with a good value to use for any particular model. Furthermore, if there are hundreds of objects in a scene, it is an arduous task to figure out the correct error tolerance for each object. We extend Garland and Heckbert's algorithm to automatically compute virtual edges using an adaptive spatial subdivision. This facilitates topology simplification, which results in LODs being merged together, and automatically and adaptively identifies an error threshold to use. In practice, it produces reasonable results.

5.2.3 Implementation

In our current implementation, the simplification of an object is pre-computed and stored in a series of levels of detail. The algorithm partitions spatially large objects; each partition is treated separately by the simplification algorithm, so small cracks can appear between partitions that share geometry but are being viewed at different levels of detail.

5.3 Visibility Culling

In addition to simplifying objects, our framework uses several visibility culling algorithms to reduce the number of rendered primitives. One is view-frustum culling, which uses the model hierarchy and bounding volumes of nodes in the scene graph to eliminate portions of the model outside the view frustum. Another is hardware backface culling for solid objects. Although the use of viewpoint cells has greatly reduced the depth complexity of the model, in many cases large portions of the model inside a viewpoint cell may still be unculled. In these situations, our framework also uses occlusion culling.

5.3.1 Occlusion Culling

The goal of occlusion culling algorithms is to cull away portions of the model which are inside the view frustum but are not visible from the current viewpoint. This problem has been well studied in computer graphics and computational geometry, and a number of algorithms have been proposed. Our criteria for an effective occlusion culling algorithm include generality (applicability to a

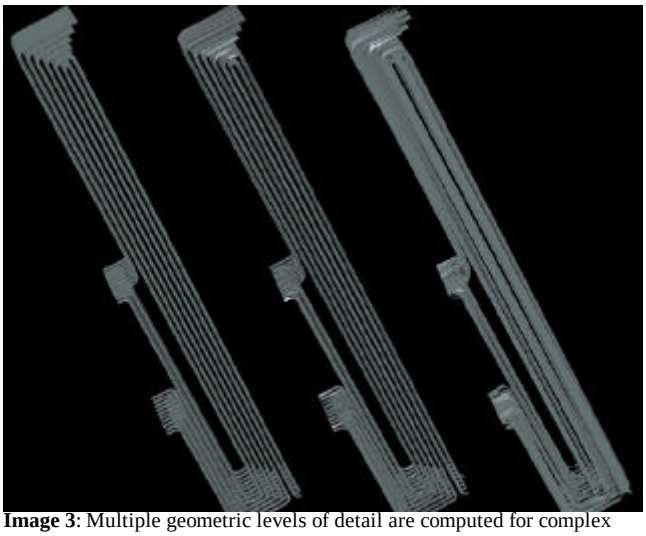


Image 3: Multiple geometric levels of detail are computed for complex objects.

wide variety of models), efficiency, and significant culling. As a result, we do not use algorithms based on cells and portals [Airey90, Teller91, Luebke95], which assume that there exists a natural partition of the model into regions with low visibility between regions and which only work well for architectural models. General algorithms for occlusion culling can be classified into object-space approaches and methods that use a combination of object space and image space. Efficient object-space algorithms for general polygonal models currently use only convex objects or simple combinations of convex objects as occluders [Coorg97, Hudson97]. As a result, they are unable to combine a "forest" of small non-convex or disjoint objects as occluders to cull away large portions of the model. [Greene93] proposed a hierarchical Z-buffer algorithm that uses an object space and an image space Z-hierarchy. However, it requires special-purpose hardware. There is a simple variation of the hierarchical Z-buffer algorithm which reads back the framebuffer and builds the Z-hierarchy in software. It can be implemented on current graphics systems. The other effective occlusion algorithm is based on a hierarchy of occlusion maps [Zhang97]. A comparison between these two algorithms is presented in [Zhang97]. Both of these are two-pass algorithms and can be integrated within our framework. Our system currently uses hierarchical occlusion maps (HOM). This allows it to perform approximate culling, which is useful for achieving interactive frame rates. (Image 4)

In order to achieve significant culling on high depth complexity models and spend a very small fraction of the overall frame time on occlusion culling, we perform automatic occluder preprocessing and use multiprocessing at run-time.

5.3.2 Occluder Preprocessing

Occluder preprocessing is done from the center of each viewpoint cell. We sample visibility by projecting geometry within the cull box onto the six surfaces of the cull box. The visible objects are put onto an occluder candidate list for this cell. Visible objects are sorted, in descending order, by the their area of projection, which is obtained by counting the number of pixels having the same object identifier.

5.3.3 Run-Time Culling

At run-time, we select occluders from the candidate list for the viewpoint cell until a maximum occluder polygon budget is reached. Before they are rendered as occluders, view frustum

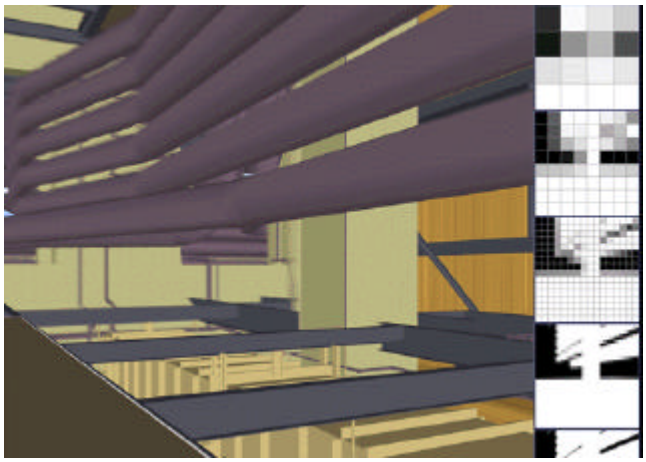


Image 4: A scene rendered from a particular viewpoint with the corresponding set of hierarchical occlusion maps displayed on the right side of the screen. The flooring, column, and a portion of the pipes have been selected as occluders.

culling is performed; potential occluders are not used if they lie outside the view frustum.

HOM culling is a two-pass algorithm: first occluder selection and rendering, then occlusion culling. The first pass requires access to the graphics hardware, and the second pass cannot start until the first is finished. To maximize parallelism, we render the occluders for frame n+1 *before* the visible geometry of frame n is rendered, so that HOM culling for frame n+1 can proceed in parallel with the rendering of frame n. The occlusion culling pass is easily parallelizable, which we take advantage of in our implementation.

6 IMPLEMENTATION

Our implementation of the framework is written in C++, using OpenGL and GLUT libraries. We ran our tests on a SGI Onyx with four 250 MHz R4400 processors, two gigabytes of main memory, and an InfiniteReality Graphics System with two RM6 boards and 64 megabytes of texture memory available to the user. In the sections below, we will give details on our multiprocessor implementation.

6.1 Multiple Processors

Our implementation uses the rendering acceleration techniques previously described. A pipelined multiprocessor architecture is used so that setup for frame n+1 occurs simultaneously with the rendering of frame n. At the same time, an asynchronous process prefetches the textures. An additional processor helps with occlusion culling.

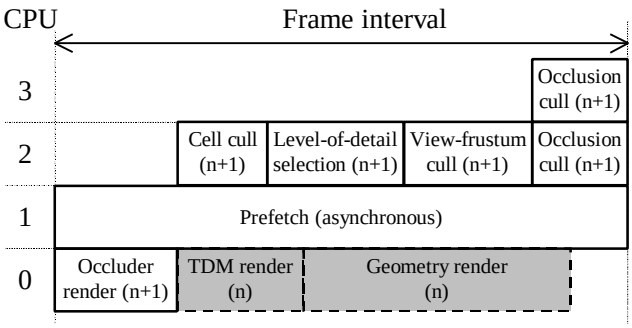


Figure 7: Multiprocessor Pipelined Implementation: Frame n+1 is being culled, while frame n is being rendered. Meanwhile the asynchronous prefetch process is speculatively loading data for future frames.

The operations performed by the framework can be abstracted into four phases (Figure 7). The phases are explained below.

Interframe Phase

The interframe is the first phase of a logical frame. It imposes a barrier synchronization between the cull and render phases. During this phase, the data structures for the next culling phase are initialized. The current viewpoint and current cell are determined. If the prefetcher has not loaded the textured depth meshes for the current cell, we fetch them from disk.

In our implementation, the occluders for the current cell are rendered into a 256x256 framebuffer and the hierarchical occlusion map and depth estimation buffer are constructed during the interframe phase as well. These operations to support HOM require the graphics pipe's resources, so we force the interframe phase to occur in the same process as the render phase. This prevents unnecessary graphics context switches, which might significantly reduce performance on single-stream graphics architectures.

Cull Phase

During the cull phase we traverse the scene graph. As we traverse, each node is first culled against the current viewpoint cell's cull box. If it is inside the cull box, we cull it against the view frustum. Finally, if the node is a visible LOD node (specifically, a node whose children are the LODs themselves), then we use a distance metric to select the current LOD. If the node is still visible, then we perform occlusion culling on it. If the prefetcher has not loaded a renderable, it is loaded now.

Our implementation employs two processors for the cull phase, one of which is dedicated to occlusion culling. Furthermore, since occlusion culling only provides a benefit in high depth-complexity areas in the model, we only enable occlusion culling when more than 100,000 triangles remain after other culling techniques have been applied and we limit the cost of rendering occluders to approximately 5% of the frame time.

Render Phase

We quickly traverse the scene graph and render only that geometry which was marked for rendering during the cull phase.

Prefetch Phase

The prefetch phase is implemented as a free-running process on its own processor. Given the current viewpoint and current cell, it fetches the necessary textured depth meshes and geometry using the algorithm described in Section 4.3. If the viewpoint changes too quickly for the prefetch process to keep up with TDM fetching, the render phase does not block; instead, no TDMs are displayed. (The render process *will* block if nearby geometry cannot be fetched fast enough; this does not happen on any of our sample paths.) As soon as the viewpoint becomes stationary or slows down enough, the proper textured depth meshes will be rendered.

6.2 Cells and Cell Boxes

In our implementation, we created viewpoint cells using model-specific data. We divided the space over the 50 stories of the power plant walkways such that a viewpoint on the walkways is never farther than one meter from the center of any cell. The cell centers were set at average human eye height above the walkways to maximize quality. This created a total of 10,565 cells. We used a constant cell box size for all cells. (Image 5, 6)

For the paths we recorded through the model, we generated four out of the six 512x512 textured depth meshes per cell box – the

walls, but not the ceiling or floor. The texture images were stored in compressed, 256-color PNG files.

6.3 Cell Parameters

To properly tune our implementation, we need to (automatically) compute various per-cell parameters.

For each viewpoint cell, we maintain a local LOD scale value. The distance to an object is multiplied by the local scale value before being used to look up which level of detail we should be displaying. During the preprocessing phase, LOD scales are computed for each cell to try to reduce the geometric complexity of renderables needed for that cell to below 200,000 triangles.

Our occlusion culling implementation needs to determine which polygons make good occluders (Section 5.3.2). As a preprocess, we perform occluder selection for each viewpoint cell.

7 PERFORMANCE RESULTS

7.1 Model Statistics

Our test model is a coal-fired power plant with an original complexity of 13 million triangles. The model was given to us with one-millimeter resolution. The main power plant building is over 80 meters high and 40x50 meters in plan. It contains 90% of the model's triangles. Additional structures surrounding the building (chimney, air ducts, etc.) contain the rest of the triangles. The database, including LODs, occupies approximately 1.3 GB of disk space (Image 7-10).

We create up to four LODs for each object over 100,000 primitives (which sum up to 7.7 million triangles). Each LOD contains half the complexity of the previous level. For faster rendering, we create triangle strips for each renderable. The swap operation used in triangle strips increases the total number of triangles to 15,240,565, but triangle strips still achieve a speedup over independent triangles. The power plant's scene graph is composed of over 198,580 nodes and 129,327 renderables.

7.2 Polygon Reduction

We rendered five views of the power plant and recorded the number of polygons culled by each acceleration technique in the pipeline. Table 1 gives results in polygon count and percentage reduction for one view in the first three columns and the average percentage reduction over the five views in the last column. [Details are available at (web address suppressed).] While the average culling percentage for LOD is over 60%, the value was 0% for one of the five views. Though not unexpected, this observation gives further support to our strategy of integrating multiple techniques to achieve our performance goals.

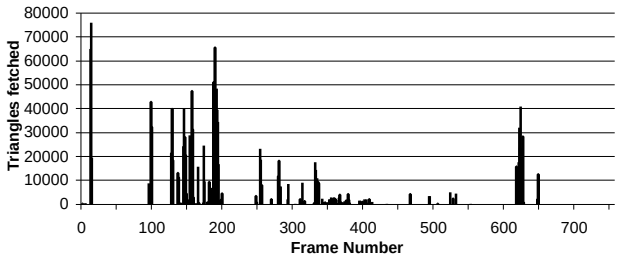
On average over the five images, only 0.9% of original model polygons remain and must be rendered as polygons. The integrated techniques consistently reduce the number of polygons to be rendered from over 15 million to about 150, 000.

Method	% Polygons culled	Polygons culled	Polygons remaining	Average % reduction over 5 views
Model Size			15,207,383	
Texture Mesh	96	14621479	585904	96
View Frustum	38	225398	360506	47
Level of Detail	45	161205	199301	47
Occlusion	3	6417	192884	10

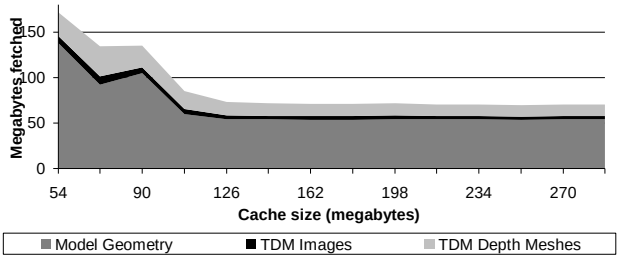
Table 1: Performance of techniques to reduce the polygon count. First three columns are data from a single view; the final column is averaged over five views.

7.3 Run-time

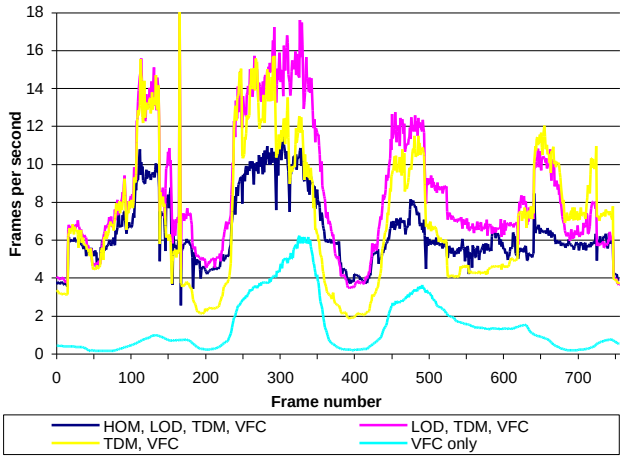
We recorded three paths through complex regions of the power plant. We played back the paths using different combinations of rendering acceleration techniques. Graph 3 shows the frame rates achieved on our SGI Onyx. Typically, we display between 5 and 15 frames per second. It is interesting to note that the savings achieved with occlusion culling (HOM) do not always outweigh its computational overhead. The objects along the sample path



Graph 1: Numbers of triangles fetched from disk during a walkthrough.



Graph 2: Performance of prefetching (total bytes fetched over path). A path through the model was recorded and then played back using different cache sizes.



Graph 3: Frame rates achieved by combining rendering acceleration techniques. HOM = Hierarchical Occlusion Maps, LOD = Geometric Levels of Detail, TDM = Textured Depth Meshes, VFC = View Frustum Culling.

are mostly long and thin and make poor occluders. Better results are achieved when large structural members such as pillars are visible.

We found that a relatively small cache is sufficient to hold the texture and model data immediately necessary for a walkthrough of the model. A user's movement through the model was recorded and then played back several times using different cache sizes. Graph 1 shows the amount of data fetched from disk along our sample path as a function of cache size. The total number of bytes

fetched from disk (including model geometry, depth meshes, and textures) was used as a measure of performance. We achieved the best results by allocating 60 megabytes for model geometry and 80 megabytes for textured depth meshes. Larger cache sizes produced no substantial improvement in the amount of traffic from disk, suggesting that capacity misses in both caches have become asymptotically low. Run-time fetching of model geometry has therefore saved us over 95% of the 1.3GB of RAM needed to hold the entire database in memory at once.

Graph 2 shows the temporal distribution of the disk I/O caused by prefetching while replaying the sample path. The bursts of activity take place when the potentially visible set of objects changes substantially – i.e. when the user's viewpoint moves from one viewpoint cell into an adjacent one. Fetching of textured depth mesh data follows a similar pattern: TDMs are fetched in bursts of 1 to 10 every 20 to 30 frames.

7.4 Preprocessing

We summarize the preprocessing times in Table 2. Each of the rendering acceleration techniques requires preprocessing. The largest amount of preprocessing time is spent generating and simplifying the textured depth meshes.

Preprocessing Technique	Time for sample paths	Extrapolated time for entire model
Generation of cells from prototypical VEF (entire model)	2 hours 45 min	2 hours 45 min
Generation of cell textures and depth meshes	56 min	221.5 hours
Pre-simplification of depth meshes	40 min	20 hours
Garland-Heckbert simplification of depth meshes	2 hours 10 min	250 hours
Generation of static levels of detail (entire model)	9 hours 30 min	9 hours 30 min
Computation of LOD scales (entire model)	1 hour	1 hour
Selection of per-cell occluders	5 minutes	23 hours 20 min
TOTAL PREPROCESSING TIME	17 hours	525 hours

Table 2: Breakdown of preprocessing times.

8 LIMITATIONS AND ONGOING WORK

We have presented a scalable framework for interactive walkthrough that is suitable for spatially large models. Our viewpoint cell mechanism relies on the model occupying a large spatial extent. We have not focused on high object-density objects (e.g. a CAD model of a car engine). Such models raise a different set of problems. The low spatial coherence of the model will strain different parts of the framework (prefetching, geometry simplification, etc).

Models with moving parts present another difficult set of issues. Most of the rendering acceleration techniques we see today are for static models. We wish to explore which algorithms can be combined to efficiently render models with limited dynamic elements.

As discussed, we are devising algorithms to automatically partition the model space into viewpoint cells.

We also want to incorporate a wider variety of visibility and LOD algorithms. Although we have tried alternate image-based algorithms for cull box faces and alternate (static and dynamic) LOD generation techniques to demonstrate modularity, this is a concept that needs to be explored farther. We would like to have an explicit set of criteria describing the classes of algorithms that our framework will accept.

9 LESSONS LEARNED

In the process of creating and implementing the massive model rendering framework, we came across various design problems.

With a massive model it is crucial to carefully construct a *single* database representation that supports all the expected rendering acceleration techniques. Some algorithms have simple data structures, whereas others have much more complex ones (for example, [Hoppe97] and [Luebke97]). We cannot afford to replicate data.

Furthermore, traversing the database is a very expensive operation (simply visiting all the scene-graph nodes of our test model takes over one-third of a second!). Algorithms that must frequently access the entire database do not scale well. When rendering massive models, the constants present in different algorithms become very significant.

A single algorithm might provide a performance increase over naive rendering, but when two algorithms are combined they do not necessarily achieve their combined speed up. For example, in our implementation, cell box culling and occlusion culling compete with each other. Both perform best when high depth complexity is present, yet one must be performed first, reducing the efficacy of the other.

A related issue is choosing the best order in which to apply multiple rendering acceleration techniques. We have chosen an order (Section 6.1) that works well for our framework and our class of model. This would not necessarily be the case for a different framework or different class of target models.

10 SUMMARY

We have presented a scalable framework for the rapid display of massive models. We have demonstrated our framework with an interactive walkthrough of a 15 million triangle model. We achieve an order of magnitude improvement over single rendering acceleration techniques.

We have described a database representation scheme for massive models. Additionally, we have presented a method to localize geometry, an essential component of a scalable walkthrough system. Our viewpoint cells partition the model space into manageable subsets we can fetch into main memory at run-time.

Furthermore, our framework includes an effective pipeline to combine rendering acceleration techniques from the areas of image-based representation, geometric simplification, and visibility culling.

11 ACKNOWLEDGMENTS

We especially thank James Close and ABB Engineering as the gracious donors of the power plant CAD model, an extremely valuable asset for us. In addition, we offer our gratitude to members of the UNC Graphics Lab, especially Kevin Arthur, Mark Livingston and Todd Gaul. Furthermore, we are grateful to our multiple funding agencies, including: ARO, DARPA, Intel, NIH, NSF, ONR and Sloan Foundation.

REFERENCES

[Airey90] J. Airey, J. Rohlf, and F. Brooks. Towards image realism with interactive update rates in complex virtual building environments. In *Proc. of ACM Symposium on Interactive 3D Graphics*, 1990, pp. 41-50.

[Akeley93] K. Akeley. Reality Engine Graphics. In *Proceedings of ACM Siggraph*, 1993, pp. 109-116.

[Aliaga96] Daniel G. Aliaga. Visualization of Complex Models Using Dynamic Texture-based Simplification. In *IEEE Visualization '96*, October 1996, pp. 101-106.

[Aliaga97] D. Aliaga and A. Lastra. Architectural Walkthroughs using Portal Textures. In *IEEE Visualization '97*, October 1997, pp. 355-362.

[Brooks86] F. Brooks. Walkthrough: A dynamic graphics system for simulating virtual buildings. In *ACM Symposium on Interactive 3D Graphics*, Chapel Hill, NC, 1986.

[Chen95] Shenchang E. Chen. Quicktime VR - An Image-Based Approach to Virtual Environment Navigation. In *SIGGRAPH 95 Conference Proceedings, Annual Conference Series*, ACM SIGGRAPH, August 1995, pp. 29-38.

[Clark76] J. Clark. Hierarchical Geometric models for visible surface algorithms. In *Communications of the ACM*, volume 19 number 10, 1976, pp. 547—554.

[Cohen91] F. Cohen and M. Patel. Modeling and synthesis of images of 3D textures surfaces. *Graphical Modeling and Image Processing*, vol. 53, pp. 501-510, 1991.

[Cohen96] J. Cohen et al.. Simplification Envelopes. In *Proc of ACM Siggraph 96*, 1997, pp. 119—128.

[Cohen97] J. Cohen, D. Manocha, and M. Olano. Simplifying Polygonal Models Using Successive Mappings. In *Proc. of IEEE Visualization*, Tampa, AZ, 1997.

[Coorg97] S. Coorg and S. Teller. Real-time occlusion culling for models with large occluders. In *Proc. Of ACM Symposium on Interactive 3D Graphics*, 1997.

[Darsa97] Lucia Darsa, Bruno Costa, and Amitabh Varshney. Navigating Static Environments Using Image-Space Simplification and Morphing. In *ACM Symposium on Interactive 3D Graphics*, Providence, RI, 1997, pp. 25-34.

[Eck95] M. Eck, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, and W. Stuetzle. Multiresolution Analysis of Arbitrary Meshes. In *Proc. of ACM Siggraph*, 1995, pp. 173-182.

[Eyles97] John Eyles, Steven Molnar, John Poulton, Trey Greer, Anselmo Lastra, Nick England, and Lee Westover. PixelFlow: The Realization, In Proceedings 1997 SIGGRAPH/Eurographics Workshop on Graphics Hardware, ACM SIGGRAPH, August 1997, pp. 57-68.

[Funkho92] Thomas A. Funkhouser, Carlo H. Sequin, and Seth J. Teller. Management of large amounts of data in interactive building walkthroughs. In *Computer Graphics (1992 Symposium on Interactive 3D Graphics)*, vol. 25, D. Zeltzer, Ed., March 1992, pp. 11-20.

[Funkho93] T. A. Funkhouser. Database and Display Algorithms for Interactive Visualization of Architecture Models. Ph.D. thesis. CS Division, UC Berkeley, 1993.

[Funkhou96] T. Funkhouser, S. Teller, C. Sequin, and D. Khorramabadi. The UC Berkeley System for Interactive Visualization of Large Architectural Models. In *Presence*, volume 5 number 1.

[Garlan97] M. Garland and P. Heckbert. Surface Simplification using Quadratic Error Bounds. In *Proceedings of ACM Siggraph*, 1997, pp. 209—216.

[Gortle96] Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. The Lumigraph. In *SIGGRAPH 96 Conference Proceedings, Annual Conference Series*, H. Rushmeier, Ed.: ACM SIGGRAPH, August 1996, pp. 43-54.

[Greene86] Ned Greene , Environment mapping and other applications of world projections. In *IEEE CG&A* 6(11), Nov 1986, pp. 21-29.

[Greene93] N. Greene, M. Kass, and G. Miller. Hierarchical Z-Buffer Visibility. In *Proc. of ACM Siggraph*, 1993, pp. 231-238.

[HP97] HP DirectModel. http://hpcc920.external.hp.com/wsg/products/grfx/dmodel/index.html, 1997.

[Hoppe93] H. Hoppe, T. Derose, T. Duchamp, J. Mcdonald, and W. Stuetzle. Mesh optimization. In *Proc. of ACM Siggraph*, 1993, pp. 19-26.

[Hoppe96] Hugues Hoppe. Progressive Meshes. In *SIGGRAPH 96 Conference Proceedings*: ACM SIGGRAPH, 1996, pp. 99-108.

[Hoppe97] H. Hoppe. View-Dependent Refinement of Progressive Meshes. In *Proceedings of ACM Siggraph*, 1997, pp. 189—198.

[Hudson97] T. Hudson, D. Manocha, J. Cohen, M. Lin, K. Hoff, and H. Zhang. Accelerated Occlusion Culling using Shadow Frusta. In *Proc. Of ACM Symposium on Computational Geometry*, 1997, pp. 1—10.

[Jepson95] W. Jepson, R. Liggett, and S. Friedman. An Environment for Real-time Urban Simulation. In *Proceedings of 1995 Symposium on Interactive 3D Graphics*, 1995, pp. 165—166.

[Levoy96] Marc Levoy and Pat Hanrahan. Light Field Rendering. In *SIGGRAPH 96 Conference Proceedings, Annual Conference Series*, H. Rushmeier, Ed.: ACM SIGGRAPH, August 1996, pp. 31-42.

[Luebke95] D. Luebke and C. Georges. Portals and Mirrors: Simple, Fast Evaluation of Potentially visible sets. In *ACM Interactive 3D Graphics Conference*, Monterey, CA, 1995.

[Luebke97] D. Luebke and C. Erikson. View-Dependent Simplification Of Arbitrary Polygonal Environments. *Proc. of ACM Siggraph*, 1997.

[Maciel95] Paulo W. C. Maciel and Peter Shirley. Visual Navigation of Large Environments Using Textured Clusters. In *1995 Symposium on Interactive 3D Graphics*, P. H. a. J. Winget, Ed.: ACM SIGGRAPH, April 1995, pp. 95-102.

[McMill95] Leonard McMillan and Gary Bishop. Plenoptic Modeling: An Image-Based Rendering System. In *SIGGRAPH 95 Conference Proceedings, Annual Conference Series*, R. Cook, Ed.: ACM SIGGRAPH, August 1995, pp. 39-46.

[McMill97] William Mark, Leonard McMillan, and Gary Bishop. Post-Rendering 3D Warping. In *ACM Symposium on Interactive 3D Graphics*, Providence, RI, 1997, pp. 25-34.

[Molnar92] S. Molnar, J. Eyles, and J. Poulton. PixelFlow: High speed rendering using image composition. *Proceedings of ACM Siggraph*, vol. 26, pp. 231-248, 1992.

[Pierce97] J.S. Pierce et al. Image Plane Interaction Techniques in 3D Immersive Environments. In *ACM Symposium on Interactive 3D Graphics*, Providence, RI, 1997, pp. 39-43.

[Pulli97] Kari Pulli, Michael Cohen, Tom Duchamp, Hugues Hoppe, Linda Shapiro, and Werner Stuetzle. View-based Rendering: Visualizing Real Objects from Scanned Range and Color Data. In *Rendering Techniques '97*. Springer Verlag, 1997, pp. 23-34.

[Regan94] Matthew Regan and Ronald Post. Priority Rendering with a Virtual Reality Address Recalculation Pipeline. In *Proceedings of SIGGRAPH '94*, July 1994, pp.155-162.

[Rohlf94] J. Rohlf and J. Helman. Iris Performer: A high performance multiprocessor toolkit for realtime 3D Graphics. In *Proc. of ACM Siggraph*, 1994, pp. 381-394.

[Ronfar96] R. Ronfard and J. Rossignac. Full-range approximation of triangulated polyhedra. *Computer Graphics Forum*, vol. 15, pp. 67-76, 462, August 1996.

[Rossig93] J. Rossignac and P. Borrel. Multi-Resolution 3D Approximations for Rendering. In *Modeling in Computer Graphics*: Springer-Verlag, 1993, pp. 455-465.

[SGI97] SGI OpenGL Optimizer. http://www.sgi.com/Technology/openGL/optimizer_wp.html, 1997.

[Schauf96] Gernot Schaufler and Wolfgang Stürzlinger. A Three-Dimensional Image Cache for Virtual Reality. In Computer Graphics Forum 15(3) (Eurographics '96 Proceedings), pp. 227-236.

[Schauf97] Gernot Schaufler. Nailboards: A Rendering Primitive for Image Caching in Dynamic Scenes. In *Rendering Techniques '97*. Springer Verlag, 1997, pp. 151-162.

[Schnei94] B. Schneider, P. Borrel, J. Menon, J. Mittelman, and J. Rossignac. Brush as a walkthrough system for architectural models. In *Fifth Eurographics Workshop on Rendering*, July 1994, pp. 389—399.

[Schroe92] W. J. Schroeder, J. A. Zarge, and W. E. Lorensen. Decimation of Triangle Meshes. In *Proc. of ACM Siggraph*, 1992, pp. 65-70.

[Shade96] Jonathan Shade, Dani Lischinski, David Salesin, Tony DeRose, and John Snyder. Hierarchical Image Caching for Accelerated Walkthroughs of Complex Environments. In *SIGGRAPH 96 Conference Proceedings, Annual Conference Series*, H. Rushmeier, Ed.: ACM SIGGRAPH, August 1996, pp. 75-82.

[Sillio97] Francois Sillion, George Drettakis, and Benoit Bodelet. Efficient Impostor Manipulation for Real-Time Visualization of Urban Scenery. In Computer Graphics Forum 16(3) (Eurographics '97 Proceedings), pp. 207-218.

[Teller91] S. Teller and C. H. Sequin. Visibility Preprocessing for interactive walkthroughs. In *Proc. of ACM Siggraph*, 1991, pp. 61-69.

[Turk92] G. Turk. Re-Tiling Polygonal Surfaces. In *Proc. of ACM Siggraph*, 1992, pp. 55-64.

[Xia97] J.Xia, J. El-Sana, and A. Varshney. Adaptive real-time level-of-detail-based rendering for polygonal models. In *IEEE Transactions on Visualization and Computer Graphics*, volume 3 number 2, June 1997, pp. 171—183.

[Zhang97] H. Zhang, D. Manocha, T. Hudson, and K. Hoff. Visibility Culling Using Hierarchical Occlusion Maps. In *Proc. of ACM Siggraph*, 1997

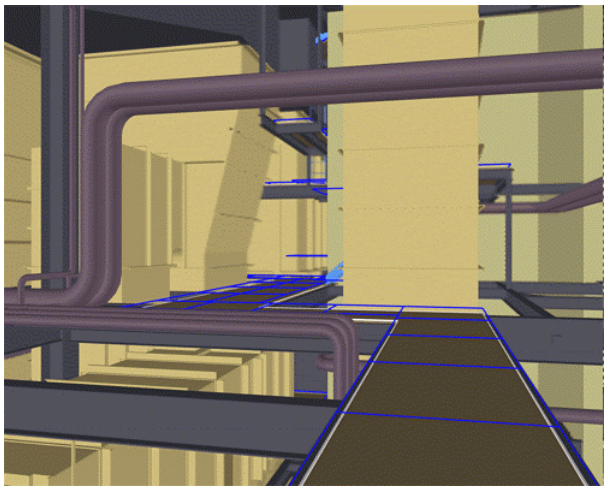


Image 5: Distribution of viewpoint cells in one part of the model. Our VEF has been chosen to lend importance to viewpoints on walkways.

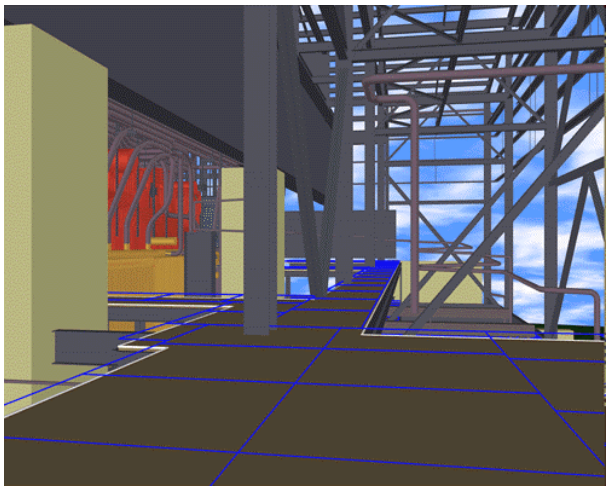


Image 6: Distribution of viewpoint cells in another part of the model.

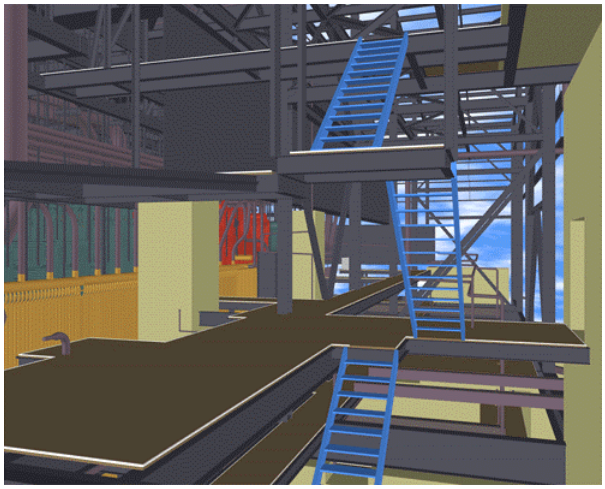


Image 7: A view from the 46th floor of the power plant model.

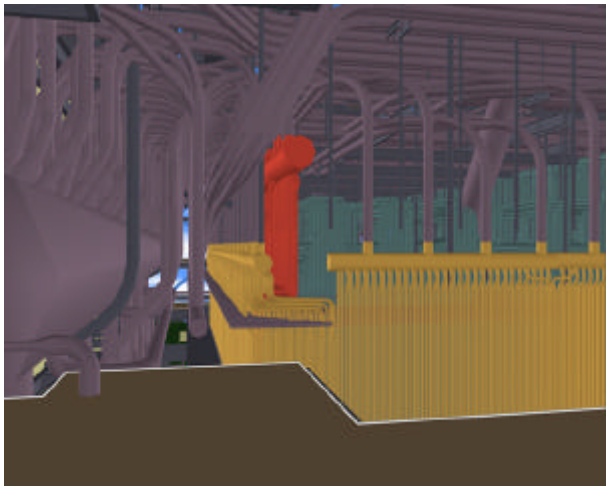


Image 8: A close-up of complex pipe arrays on the 46th floor. The piping provides a challenge for LOD algorithms.

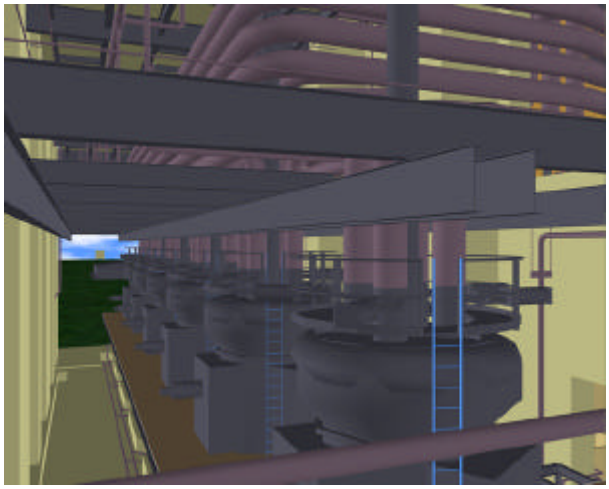


Image 9: A view of the lower power plant floors.

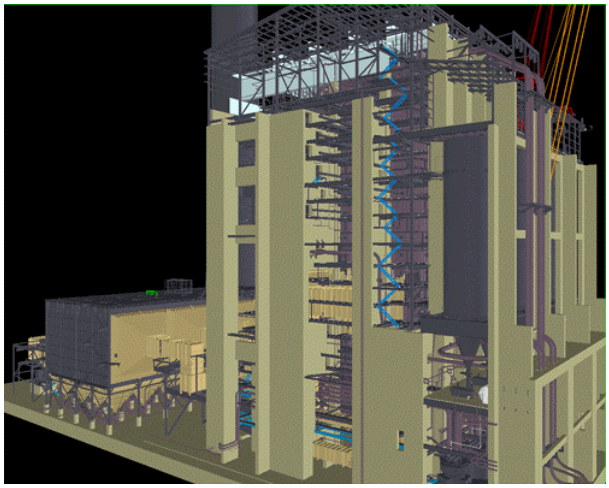


Image 10: A view of the outside of the power plant.