MMR: An Interactive Massive Model Rendering System Using Geometric And Image-Based Acceleration

Daniel Aliaga[*], Jon Cohen[+], Andrew Wilson, Eric Baker, Hansong Zhang[⊥], Carl Erikson,
Kenny Hoff, Tom Hudson, Wolfgang Stuerzlinger[°], Rui Bastos, Mary Whitton,
Fred Brooks, Dinesh Manocha

{aliaga, cohenj, awilson, baker, zhangh, eriksonc,
hoff, hudson, stuerzl, bastos, whitton, brooks, dm}@cs.unc.edu
Department of Computer Science
University of North Carolina at Chapel Hill

ABSTRACT

We present a system for rendering very complex 3D models at interactive rates. We select a subset of the model as preferred viewpoints and partition the space into virtual cells. Each cell contains near geometry, rendered using levels of detail and visibility culling, and far geometry, rendered as a textured depth mesh. Our system automatically balances the screen-space errors resulting from geometric simplification with those from textured-depth-mesh distortion. We describe our prefetching and data management schemes, both crucial for models significantly larger than available system memory. We have successfully used our system to accelerate walkthroughs of a 13 million triangle model of a large coal-fired power plant and of a 1.7 million triangle architectural model. We demonstrate the walkthrough of a 1.3 GB power plant model with a 140 MB cache footprint.

Keywords: interactive walkthrough, massive models, occlusion culling, levels of detail, textured depth mesh, image-based rendering, prefetching.

1 INTRODUCTION

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

* Currently at Lucent Technologies Bell Laboratories, Murray Hill, NJ, aliaga@research.bell-labs.com

+ Currently at Dept. of Computer Science, Johns Hopkins University, Baltimore, MD, cohen@cs.jhu.edu

⊥ Currently at Silicon Graphics Inc., Mountain View, CA, hzhang@engr.sgi.com

° Currently at Dept. of Computer Science, York University, Toronto, Ontario - Canada, wolfgang@cs.yorku.ca

Ideally, such walkthroughs need to maintain an update rate of at least 20 frames per second. Many of these massive CAD databases contain millions of primitives, and even high-end systems such as the SGI Infinite Reality cannot render them at interactive rates. Furthermore, we observe that model sizes are increasing much faster than rendering capabilities.

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 a principle culls 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 other objects, or satisfactorily shown in a painted texture rather than as geometry. No one technique suffices for creating interactive walkthroughs of most massive models (models that do not fit within memory). Moreover, each technique achieves great speedups only for particular subsets of the primitives. Any *general* system for interactive walkthroughs should combine such techniques.

1.1 System Overview

The fundamental idea in our system is to render objects "far" from a viewpoint using fast image-based techniques [Maciel95, Shade96, Schauf96, Aliaga97, Darsa97, Sillio97] and to render all objects "near" the viewpoint as geometry using levels of detail [Turk92, Rossig93, Cohen97, Garlan97, Luebke97, Hoppe97] and visibility culling [Airey90, Teller91, Hudson97, Zhang97]. Consequently, we limit the data required to render a model of any size to a reduced amount of near geometry and an approximately constant-size representation of far geometry.

Just as in progressive rendering, some parts of a scene are chosen for preferential rendering in time, so we introduce a spatial *View Preference Function* (VPF). All parts of the model can be viewed from any viewpoint inside or outside the model volume. However, views from the preferred viewpoints are allocated more resources (e.g. rendering capability, secondary storage, bandwidth, preprocessing time, and running time) so that they can be rendered at interactive frame rates.

In Color Figure A, we have outlined the box containing near geometry for a particular viewpoint in a power plant model. The darker colored geometry has been culled. The system includes extensive preprocessing to create textured polygon impostors used at run time to replace far 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 rendering time.

1.2 Contributions

This paper presents five primary contributions to ongoing walkthrough research:

- A rendering scheme which employs both images and geometric levels of detail, automatically balancing the quality and speed-up of the two approaches.
- A concept of preferred viewpoints, which are rendered more rapidly because they are allocated a disproportionate share of resources.
- An approach to rendering massive models which partitions the model into manageable virtual cells, each of which can be optimized for speed, quality, and memory usage.
- A system pipeline to manage resources (i.e., CPUs, main memory, texture memory, graphics engines) and allocate them among the acceleration techniques.
- An integrated database, with a coherent representation technique, memory management, and prefetching of geometry and textures, all of which are crucial for databases larger than physical memory.

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. A large number of systems have been developed for interactive walkthroughs. We subdivide them into five general categories:

- Architectural Walkthrough Systems
- Image-Based Rendering Acceleration Systems
- Mechanical CAD Systems
- High-Performance Libraries
- Architectures and APIs

Clark [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 widely used.

Several walkthrough systems for architectural models [Brooks86] have been presented [Airey90, Teller91, Funkho92, Luebke95]. These systems partitioned the model into cells and portals, following the division of a building into discrete rooms. The UC Berkeley Building Walkthrough System [Funkho96] used a hierarchical representation of the model, along with visibility algorithms [Teller91] and LODs. Funkhouser *et al.* [Funkho92] used an adaptive display algorithm to maintain interactive frame rates [Funkho93]. Aliaga [Aliaga97] built upon a cells and portals system, replacing geometry visible through portals with images.

Several image-based rendering acceleration systems are specialized for outdoor models. Maciel and Shirley [Maciel95] expanded an LOD system to allow a general set of impostors (LODs, textured billboards, etc.). Shade *et al.* [Shade96] and Schaufler and Stuerzlinger [Schauf96] used image caching to accelerate interactive walkthroughs.

The IBM BRUSH system [Schnei94] provided real-time visualization and inspection of very large mechanical CAD (and architectural) models. It used multiple LODs of the objects in the scene [Rossignac93]. Avila and Schroeder [Avila97] described another system, for visualizing power generation engines and aircraft CAD models. Erikson and Manocha [Erikson98, Erikson99] have proposed algorithms for the visualization of large models using LODs and HLODs (hierarchical levels of detail) and have used them for interactive display of large static and dynamic environments.

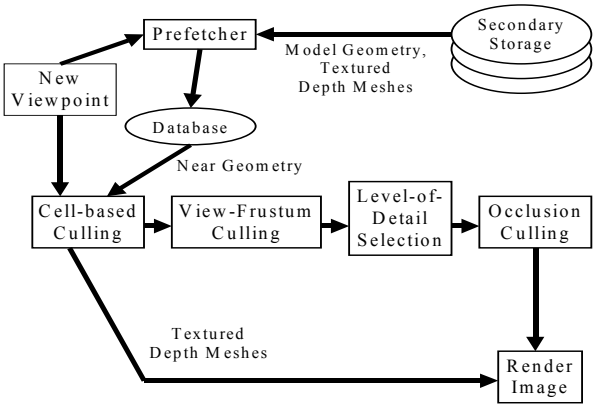


Figure 1. *System Pipeline. Each new viewpoint sent into the pipeline is passed both to the prefetcher and to the rendering pipeline. The viewpoint determines what geometry and TDMs are retrieved from disk; it is also a parameter used by the rendering acceleration techniques.*

IRIS Performer [Rohlf94], a high-performance library, used a hierarchical representation to organize the model into smaller parts, each of which had an associated bounding volume. This data structure was used to optimize culling and rendering. Other systems have been developed on top of Performer for walkthrough of large environments, including real-time urban simulation [Jepson95].

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]. These systems provide standalone tools for simplifying polygonal models or performing visibility culling.

3. SYSTEM

In this section, we detail our system pipeline (Figure 1). First, we describe the representation used for far geometry. Second, we summarize the simplification performed on near geometry. For each, we describe the preprocessing and run-time components. Third, we present an algorithm to balance the quality of near geometry and far geometry representations.

3.1 Far Geometry

The first acceleration technique we use substitutes texture impostors for distant geometry. As a preprocess, we partition the space of the model into *virtual cells*. These cells are similar to those used in architectural models, but they need not coincide with walls or other large occluders. Around each cell we place a *cull box*. The cull box divides the space into near and far geometry; the far geometry, outside the cull box, is not rendered when the viewpoint is inside the virtual cell. We generate for each of the six inside faces of the box a *textured depth mesh* (TDM) to replace the far geometry [Darsa97][Sillion97]. Together, they image the outside of the box as viewed from the cell centerpoint. Each depth mesh (similar to a height-field) is simplified.

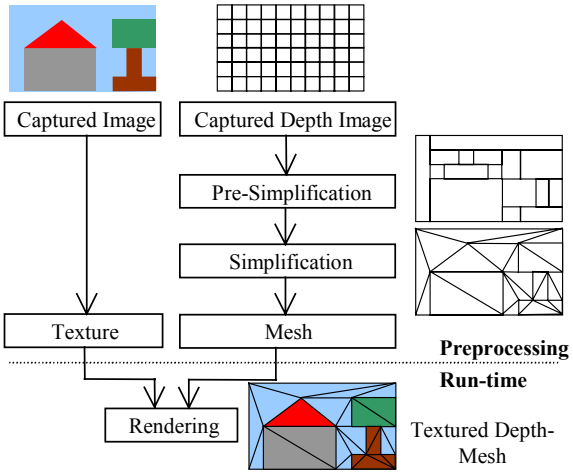


Figure 2. *Offline and online components of textured-depth mesh generation. The preprocess generates a texture and a mesh from a captured image and a captured depth image; the textured depth meshes are displayed at run time.*

Using a general simplification algorithm with error bounds to process a large number of dense meshes is time consuming. Instead, we render each face of the cull box, store the resulting image, create a depth mesh of the same resolution, and apply a fast pre-simplification algorithm to the depth mesh (Figure 2). This step simplifies rectangular planar regions, using a greedy search. The resulting mesh has approximately one tenth its original polygon count. Then, we apply the more general simplification algorithm of Garland and Heckbert [Garlan97]. No special treatment of discontinuities is used.

At run time, we cull away all the geometric primitives outside the cull box of the cell closest to the current viewpoint. We display the cell's simplified depth meshes using projective texture mapping to place the image rendered from the cell's center onto the mesh (Color Figure B). As the viewer moves from cell to cell, simple prediction enables the relevant TDMs to be prefetched.

The use of a TDM rather than a static, textured quadrilateral provides some perspective correction at the mesh vertices (similar to three-dimensional image warping [McMillan95]) which radically reduces popping and the other artifacts that can occur as the viewpoint moves between cells. Moreover, 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 into *skins* to cover regions where no information is available. As different objects can be visible in the skin regions for different cells, small popping artifacts may still appear when moving between the TDMs of adjacent cells.

3.2 Near Geometry

After culling away the portions of the scene outside the current cull box, we render the near geometry. We cull to the view frustum, cull back-facing triangles, select for each object a simplified surrogate model (with the degree of simplification depending upon the object's distance), and perform occlusion culling (Color Figure C).

During preprocessing, we use the algorithm of Erikson and Manocha [Erikson99] to compute four levels of detail for each of the objects in the model, each with half the complexity of its predecessor. Objects significantly larger than the average cull box size are partitioned; then, each partition is separately simplified.

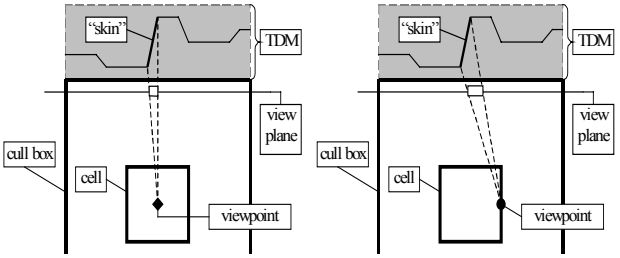


Figure 3*. TDM Error Metric. (Left) Skin as seen from center of virtual cell. (Right) Skin as seen from the cell boundary. TDM error metric measures the maximum skin size, in screen-space pixels, for each cell.*

Small cracks can appear between partitions that share geometry but are being viewed at different levels of detail.

We cull geometry hidden behind other objects if the current cull box has enough triangles to approximately outweigh the cost of occlusion culling. Our occlusion culling implementation is based on that of Zhang *et al.* [Zhang97]. Briefly, we preprocess each virtual cell to select potential occluders. At run time, the algorithm uses a two-pass scheme. First, the potential occluders of the current cell are rendered to create a hierarchical occlusion map. Then, during the culling pass, we cull objects whose screen-space bounding box is hidden by previously rendered occluders.

3.3 Balancing The Quality Of Near And Far Geometry Representations

Both the near and far geometry representations are simplifications of the underlying model geometry. They both introduce visual errors that we would like to balance for uniform quality across the rendered frame. Using a pair of error metrics, we compute, for each virtual cell, a cull box size and LOD error threshold to balance the combined error of the near and far geometry representations.

3.3.1 Error Metrics

We define two measures to quantify our visual errors. First, we measure the error introduced by near geometry simplification. Our visibility culling algorithms do not alter the visual content but our LOD simplification does. Thus, we choose to use an upper bound on the difference (in pixels) between an object's silhouette and the simplified object's silhouette as the error metric.

Second, we measure the error in the far geometry representation. This error comes from two sources: (1) the process used to simplify a screen-resolution depth mesh to a manageable mesh, and (2) the stretching of TDM skins as the viewer moves away from a cell center. For simplicity, we consider the error due to mesh simplification to be constant and measure worst-case error by finding the maximum distance (in pixels) that a skin stretches as one moves to a cell boundary (Figure 3).

3.3.2 Reducing Polygonal Complexity While Balancing Error

To achieve walkthroughs with fast frame rates, we can render only a modest number of primitives. Given a desired polygon budget, we devote a fixed number of polygons to rendering our TDMs. We use the remainder of our budget by balancing the size of the cull box and the levels of detail of unculled geometry. Simply adjusting the size of the cull box often does not yield the best image quality. Small cull boxes are bad, because stretching of skins in the TDM is more exaggerated. Levels-of-detail can be applied to the geometry inside the cull box, but as the size of the cull box grows, coarser LODs must be used to maintain the target

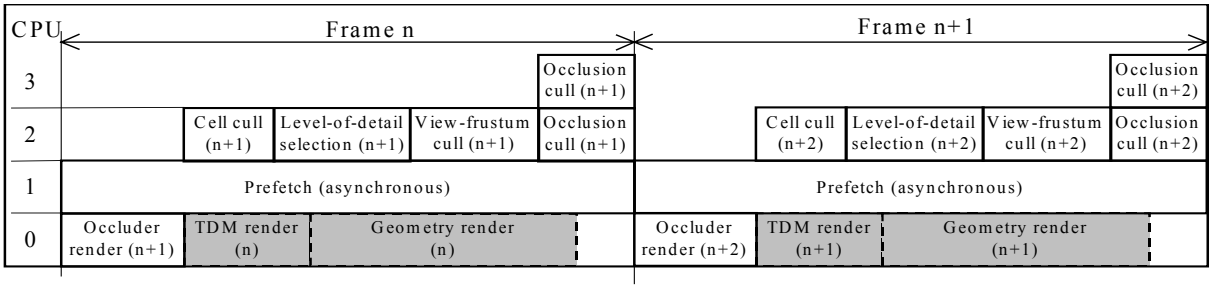


Figure 4. *Multiple Processor Pipeline. We show the tasks performed during frame n and frame n+1. The work is distributed among four processors. Processor 0 performs all the rendering operations for the near geometry and TDMs. Processor 1 executes the asynchronous prefetching algorithm. Processor 2 steps through the four main culling operations: cull geometry outside the current cull box, select the current LODs, cull geometry to the view frustum, and cull occluded objects. Processor 3 helps accelerate the occlusion culling operations.*

number of primitives. To improve the image quality, we find the cull box size and LOD error threshold that reduces a weighted sum of the TDM and LOD visual errors (after several trials, we converged on a weight ratio of 0.15 pixels of TDM error to one pixel of LOD error).

As one increases the size of a cell's cull box and holds the number of unculled primitives constant, errors contributed by the TDM never increase, while errors due to an increasing LOD error threshold never decrease. Based on this monotonic behavior of our error metrics, we use nested bisection routines to balance and minimize our errors.

4 DATA REPRESENTATION AND MANAGEMENT

Data representation is a major issue when combining multiple acceleration techniques. Though each technique may have its own ideal data structures, we use a common representation that allows efficient traversal and avoids replication of data at all costs. Our system uses two main data structures: a *scene graph* and a *cell graph*.

4.1 Scene Graph

The system stores the model database in a scene graph hierarchy. We implemented this from the ground up, on top of OpenGL, for maximum flexibility. Such a system could also be built on top of Iris Performer [Rohlf94] or Inventor. Each node in our scene graph may have an arbitrary number of children, and each has a bounding box used for culling. Any node in the scene graph may also have attached a set of geometry stored as triangle strips (a *renderable*).

The scene graph is automatically constructed from the model database. We would like to organize our scene graph spatially, but many real-world models have object hierarchies grouped according to non-spatial criteria, e.g. functional organization. We thus use the model's grouping as the upper layer of our hierarchy, and below that construct an octree-like bounding volume hierarchy. This subdivision terminates when one of three stopping criteria is reached: a minimum number of polygons per leaf, a maximum depth, or a minimum bounding volume size.

We store all geometry as triangle strips in the leaf nodes. There is an important trade-off between the average triangle strip length and the size of leaf-node bounding volumes. On average, larger leaf nodes allow longer triangle strips, enabling faster rendering. Smaller leaf nodes allow more accurate culling, but limit the length of triangle strips. This trade-off must be considered when choosing termination conditions for the octree subdivision. In our system, we empirically determined a leaf-node bounding volume size that produces triangle strips that provide an overall performance gain.

At run time, the scene graph is traversed once, in depth-first order. Our acceleration techniques are implemented as callback functions. LOD support is handled with special LODNodes, similar to Performer. During our traversals, a single child of each LODNode is active; the distance to the node's bounding box, the viewing parameters, and the LOD error threshold determine the active child.

4.2 Cell Graph

The virtual cells are organized into a graph structure to facilitate prefetching. Cells that are adjacent in space are connected by edges in the graph. Each cell node stores its location, its size, and five other fields:

- the size of its associated cull box,
- a LOD error threshold,
- the IDs of potential occluders,
- the IDs of the cull box's TDMs, and
- the IDs of renderables contained in the cull box.

Since the cells are spatially connected and the viewpoint does not move much from one frame to the next, we can always quickly find the cell containing the viewpoint. Then, the IDs and the speculative prefetching algorithm (next section) are used to find the neighboring cells whose TDMs and renderables to fetch.

5 SYSTEM IMPLEMENTATION

Our MMR system is written in C++, using OpenGL and GLUT. We ran our performance tests on a SGI Onyx2 with four 195 MHz R10000's, 1 GB of memory, Infinite Reality Graphics, two RM6 boards and 64 MB of texture memory.

5.1 Multiple Processors

Our system uses three processors to set up for frame *n+1* and render frame *n* (Figure 4). On the fourth processor, an asynchronous process prefetches the TDMs and renderables. We divide each frame's work into four phases: interframe, cull, render, and prefetch. Distributing these tasks realizes a significant performance increase, but introduces one frame of additional latency.

5.1.1 Interframe Phase

The interframe takes place on the same processor that will later render the current frame's geometry (frame *n*) and imposes barrier synchronization between the cull phase and render phase. It also

```
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 use prediction rules to
  enumerate in order of increasing distance
  from viewpoint:
    Look up geometry G needed to render cell 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 and TDMs T from queues
```

Figure 5. *Prefetch Algorithm.*

determines which cell contains the viewpoint and chooses the TDMs for frame *n+1*.

Our implementation of occlusion culling requires the graphics pipeline to render occluders. We perform this task as part of the interframe phase, limiting it to roughly 5% of the frame time. Since this rendering occurs on the same processor as the render phase, we avoid costly graphics context switches.

5.1.2 Cull Phase

During the cull phase, we traverse the scene graph and select the geometry to render for frame *n+1*. Because traversal of the scene graph is expensive, we perform our four culling operations in a single traversal. First, each scene-graph node is tested for overlap with the current cull box. Second, if the boxes overlap, we cull the node against the view frustum. We also determine which of the six TDMs of the current cell are visible. Third, if the node being visited is an LOD node (whose children represent the same object at different levels of detail), we use the viewpoint and per-cell LOD error threshold to select an LOD. Fourth, if the node is still visible, we occlusion cull.

Combining acceleration techniques is complicated by the fact that different techniques are better suited to different models. Occlusion culling performs best on scenes having high depth complexity; by discarding all geometry outside the current virtual cell's cull box, we sharply limit depth complexity, restricting the utility of occlusion culling. Thus, we disable occlusion culling when a view from inside the cull box contains fewer than 100,000 polygons.

5.1.3 Render Phase

During the render phase, we first render the visible textured depth meshes of the current cell, then quickly traverse the scene graph and render geometry that was marked as visible during the cull phase.

5.1.4 Prefetch Phase

Speculative prefetching is implemented as an asynchronous process running on a dedicated processor. We maintain a priority queue of geometry and TDMs likely to be needed in the near future, with higher priorities assigned to the objects of closer cells. The prefetch process traverses this queue, loading the requested data from disk.

5.2 Prefetch Algorithm

During preprocessing, we use the virtual cells' cull boxes to compute the potentially visible near geometry for each cell. At run time, we maintain a list of cells that we predict the user will visit soon, then build a prefetch queue by examining the potentially visible geometry for each cell. Our prediction algorithm is based on the user's velocity and viewing direction (Figure 5), and follows these rules:

- If the user is moving slowly, we assume that she is interested in the immediate vicinity. We fetch data for adjacent cells in order of increasing distance from the current cell, spiraling outward from the current cell.
- If the user is moving quickly, we assume there will be no abrupt changes in velocity or viewing direction. We fetch data for cells that lie in the user's path. Furthermore, we only fetch the TDMs we predict will be in the view frustum when the user reaches the cell.
- If the user's viewing direction changes more than 30 degrees between frames, regenerate the prefetch queue.

Model geometry and TDMs are kept in separate caches in main memory. Both caches are managed using a least-recently-used replacement policy. The coarsest LOD is fetched first. A lower priority request to load the appropriate LOD is appended to the prefetch queue. If the user stays within the same area, eventually the appropriate LOD for all objects will be loaded.

6 PERFORMANCE RESULTS

6.1 Model Statistics

We tested our system with two models: a 13 million triangle coal-fired power plant model and a 1.7 million triangle architectural model.

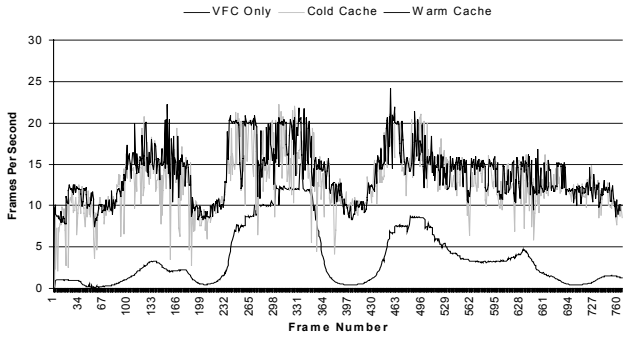
The main power plant building is over 80 meters high and 40x50 meters in plan and it contains 90% of the entire model's triangles. Surrounding chimney, air ducts, etc. contain the rest of the triangles. The geometry for the model, including LODs, occupies approximately 1.3 GB of disk space. To create the virtual cells, we divided the space over the 54 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. This created a total of 10,565 cells. The cell centers were set at average human eye height above the walkways. We created LODs for objects with over 100,000 primitives (which totals to 7.7 million triangles). The swap operation used for creating triangle strips increases the model size to 15 million triangles, but still yields a speedup. The scene graph has 198,580 nodes and 129,327 renderables.

The architectural model is of a radiositized house. We have no LODs; TDMs are the main source of rendering acceleration. We created 255 cells spanning the floor plan. The scene graph has 668 nodes and 552 renderables.

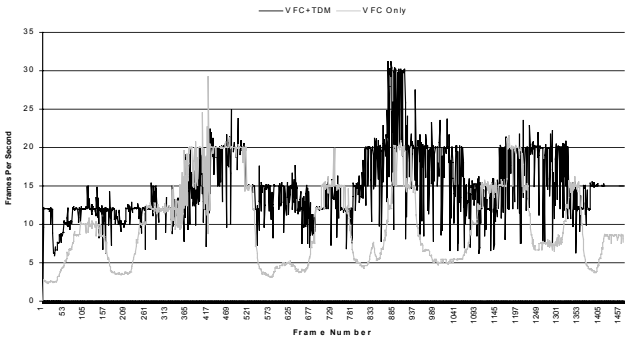
For the paths we recorded, we generated six 512x512 textured depth meshes per virtual cell. The texture images are stored in compressed, 256-color PNG files.

Acceleration Method	Percent of Remaining Polygons culled	Polygons culled	Polygons remaining	Average percent reduction over five views
None			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 our techniques to reduce the polygon count on the power plant model. First three columns are data from a single view; the final column is averaged over five views.*



Graph 1. *Power plant model frame rates achieved by our system with view-frustum culling only, a cold cache and a warm cache.*



Graph 2. *Architectural model frame rates achieved by our system with view-frustum culling only and with TDMs.*

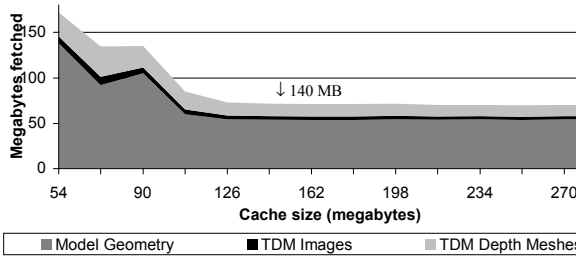
6.2 Polygon Reduction

To demonstrate polygon reduction, we rendered five views of the power plant model 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. 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 further supports our strategy of combining multiple techniques. On average over the five images, only 0.9% of original model polygons remain and must be rendered as polygons.

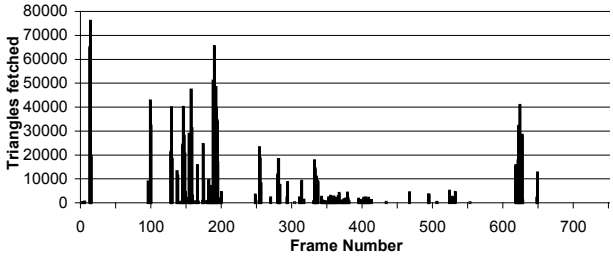
6.3 Run-Time

We recorded a path through each model (Graphs 1 and 2). During the paths, we display 10 to 30 frames per second. The sudden decreases in performance for the cold cache times of Graph 1 are due to prefetching. These downward spikes are not present when data are already loaded.

We found that relatively small caches are sufficient to hold the texture and model data immediately necessary. We used the power plant model to determine the smallest cache size that does not hinder performance. A user's movement through the model was recorded and then played back using several cache sizes. Graph 3 shows the data fetched from disk along our sample path as a function of cache size. The total number of bytes fetched (including model geometry, TDMs, and textures) was used as a measure of performance. We achieved the best results by allocating 60 MB for model geometry and 80 MB for textured depth meshes. Starting with a cold cache, total cache sizes larger than 140 MB produced no substantial improvement; misses have



Graph 3. *Performance of prefetching with different cache sizes*



Graph 4. *Temporal distribution of I/O for recorded path.*

become asymptotically low. Run-time prefetching of geometry and TDMs has saved us over 89% of the 1.3GB of RAM needed to hold the entire database in memory.

Graph 4 shows the temporal distribution of the disk I/O caused by prefetching while replaying the same path used to gather data for Graph 3. The bursts occur when the potentially visible set of geometry changes substantially – i.e. when the user moves from one virtual cell into an adjacent one. Fetching of textured depth mesh data follows a similar pattern: bursts of 1 to 10 meshes every 20 to 30 frames.

6.4 Preprocessing

Tables 2 and 3 summarize the preprocessing times. Each of the acceleration techniques requires some preprocessing. The largest amount is spent generating and simplifying the TDMs. On average, power plant meshes simplify to 10,000 triangles (161 KB including texture) and the architectural model's to 1,900 triangles (38 KB including texture). Much of the other preprocessing can be parallelized and does not require a graphics workstation. Only the generation of the textures requires a graphics workstation.

Preprocessing Technique	**Time for sample paths**	**Extrapolated time for entire model**
Automatic cull box size vs. LOD error threshold balancing	1 hour	50 hours
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
Selection of per-cell occluders	5 minutes	23 hours 20 min
TOTAL PREPROCESSING TIME	5 hours	575 hours
TOTAL PREPROCESSING SPACE	24 MB	10 GB

Table 2. *Power Plant Model. Breakdown of preprocessing times.*

Preprocessing Technique	Time for entire model
Generation of cell textures and depth meshes	6 hours
Pre-simplification of depth meshes	2 hours 40 min
Garland-Heckbert simplification of depth meshes	4 hours 20 min
TOTAL PREPROCESSING TIME	13 hours
TOTAL PREPROCESSING SPACE	58 MB

Table 3. *Architectural Model. Breakdown of preprocessing times.*

7 LIMITATIONS AND FUTURE WORK

Our virtual cell mechanism is more suitable for models that have a large spatial extent. We have not focused on other classes of models, such as high object-density models (e.g. engine room of a submarine or aircraft carrier). Such models raise a different set of problems, e.g. the many details in the area immediately surrounding the viewer will strain parts of the system (prefetching, geometry simplification, etc). Furthermore, because of the large number of details appearing and disappearing due to changing occlusions, it would be difficult to sample all the surfaces sufficiently for a high-fidelity image-based representation. In addition, for our system, we manually decided where to create the virtual cells. Ideally, this should be done automatically. We find the concepts of Space Syntax, developed by Hillier [Hillier96] of University College London, to be promising for automatic View Preference Function generation.

Our run-time prefetching scheme allows us to render models larger than memory. To generate TDMs, however, we must render subsets of the model potentially larger than memory. At present, this process is not optimized to employ prefetching. In order to render extremely massive models, we would need to page even the skeleton of the scene-graph hierarchy.

When combining rendering acceleration techniques, it is crucial to know which method is perceptually more appropriate for each model subset. We have described an algorithm for balancing the errors introduced by TDM rendering and LOD simplification. Our conservative metrics, although they do quantify visual artifacts, do not necessarily measure the perceptual error. Moreover, the constants used for our weighted-sum method were determined very subjectively . We need to do a more comprehensive study and analysis of quantifying the visual impact of each rendering acceleration method.

Models with moving parts present another difficult set of issues. Many of the acceleration techniques of today, particularly the image-based ones, are for static models. We wish to explore how algorithms can be combined to render models with limited dynamic elements.

Finally, during the implementation of this large system we have made several choices: we chose a particular order to apply our rendering acceleration techniques and we empirically determined the value of several system parameters. As future work, we need to investigate the scalability and overall effect of choosing different values from this multi-dimensional parameter space. For example: if the system is limited by the number of primitives to render (render-bound), we should emphasize culling techniques in order to reduce the number of primitives; if the system is limited by the overhead of the simplification and culling algorithms (cull-bound), as might be the case with our occlusion culling method, we should automatically do less culling. In addition, we need to more precisely evaluate the time-space-quality tradeoff of the different methods. What is the best we can do with a fixed space budget? What is the best we can do with limited preprocessing time? There are many such research questions that we need to investigate.

8 CONCLUSIONS

We have presented a rendering system for the rapid display of massive models. We have demonstrated our system on two very large models (Color Figure D). We have also described a database representation scheme. Our system includes a method for localizing geometry, an essential component of a scalable walkthrough system. Our virtual cells partition the model space into manageable subsets that we can fetch into main memory at run time. Furthermore, our system includes an effective pipeline to combine acceleration techniques from image-based rendering, geometric simplification, and visibility culling.

We encountered various design problems, including:

- 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 (e.g. [Hoppe97], [Luebke97]). We cannot afford to replicate data.
- Any single algorithm provides a performance increase over naive rendering, but two combined algorithms do not necessarily work well together - for example, TDMs and occlusion culling compete.
- The order of applying multiple acceleration techniques that works well for our system and our class of models would not necessarily work well for a different system or with a different class of models.

9 ACKNOWLEDGMENTS

We especially thank James Close and Combustion Engineering, Inc., as donors of the power plant CAD model, an extremely valuable asset for us. We thank William Baxter, Mark Harris, and Wesley Hunt for helping with the results for the Brooks House model. We appreciate the help of members of the UNC Graphics Lab, especially Kevin Arthur, Mark Livingston and Todd Gaul. We also thank our funding agencies: ARO, DARPA, Honda, Intel, NIH National Center for Research Resources, 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. *ACM Symposium on Interactive 3D Graphics*, 1990, pp. 41-50.

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

[Avila97] L. Sobierajski Avila and William Schroeder. "Interactive Visualization of Aircraft and Power Generation Engines. *IEEE Visualization*, October 1997, pp. 483-486.

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

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

[Cohen97] J. Cohen, D. Manocha, and M. Olano. Simplifying Polygonal Models Using Successive Mappings. *IEEE Visualization*, October, 1997, pp. 395-402.

[Coorg97] S. Coorg and S. Teller. Real-time occlusion culling for models with large occluders. *ACM Symposium on Interactive 3D Graphics*, 1997, pp. 83-90.

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

[Erikson98] C. Erikson and D. Manocha. Simplification Culling of Static and Dynamic Scene Graphs. *UNC-Chapel Hill Computer Science Technical Report* TR98-009, 1998.

[Erikson99] C. Erikson and D. Manocha. GAPS: General and Automatic Polygonal Simplification. *UNC-Chapel Hill Computer Science Technical Report* TR98-033, 1998. To appear in *ACM Symposium on Interactive 3D Graphics*, 1999.

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

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

[Funkhou96] T. Funkhouser, S. Teller, C. Sequin, and D. Khorramabadi. The UC Berkeley System for Interactive Visualization of Large Architectural Models. *Presence*, volume 5 number 1, 1996.

[Garlan97] M. Garland and P. Heckbert. Surface Simplification using Quadratic Error Bounds. *ACM SIGGRAPH*, 1997, pp. 209-216.

[Hillier96] B. Hillier, Space is the Machine. Cambridge University Press, 1996.

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

[Hoppe97] H. Hoppe. View-Dependent Refinement of Progressive Meshes. *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. *ACM Symposium on Computational Geometry*, 1997, pp. 1-10.

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

[Luebke95] D. Luebke and C. Georges. Portals and Mirrors: Simple, Fast Evaluation of Potentially visible sets. *ACM Symposium on Interactive 3D Graphics*, Monterey, CA, 1995, pp. 105-106.

[Luebke97] D. Luebke and C. Erikson. View-Dependent Simplification Of Arbitrary Polygonal Environments. *ACM SIGGRAPH*, 1997, pp. 199-208.

[Maciel95] Paulo W. C. Maciel and Peter Shirley. Visual Navigation of Large Environments Using Textured Clusters. *ACM Symposium on Interactive 3D Graphics*, April 1995, pp. 95-102.

[McMill95] Leonard McMillan and Gary Bishop. Plenoptic Modeling: An Image-Based Rendering System. ACM *SIGGRAPH*, August 1995, pp. 39-46.

[Rohlf94] J. Rohlf and J. Helman. Iris Performer: A high performance multiprocessor toolkit for realtime 3D Graphics. *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. *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. Computer Graphics Forum 15(3) (*Eurographics*), pp. 227-236.

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

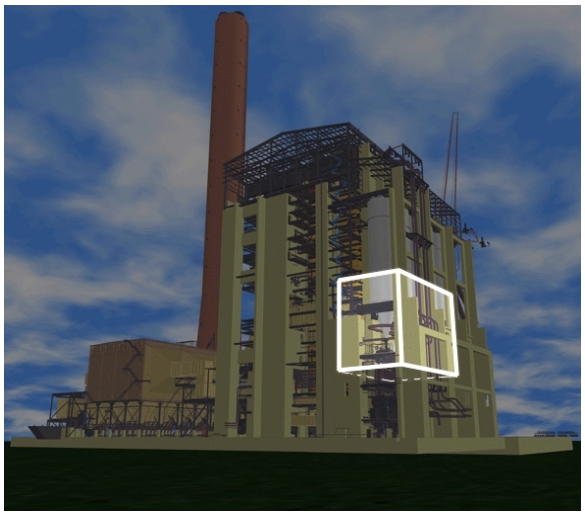
[Shade96] Jonathan Shade, Dani Lischinski, David Salesin, Tony DeRose, and John Snyder. Hierarchical Image Caching for Accelerated Walkthroughs of Complex Environments. *ACM SIGGRAPH*, August 1996, pp. 75-82.

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

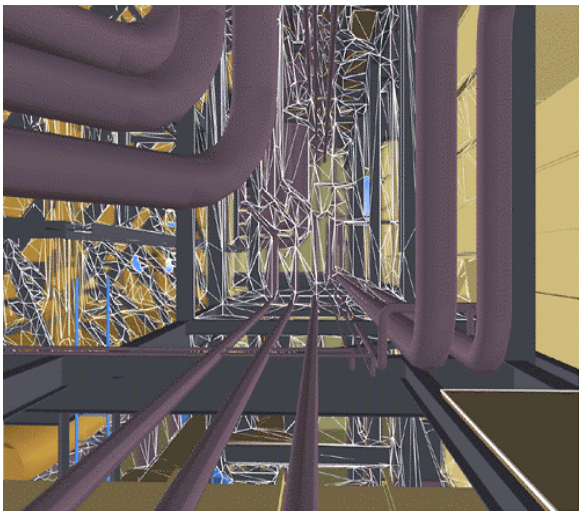
[Teller91] S. Teller and C. H. Sequin. Visibility Preprocessing for Interactive Walkthroughs. *ACM SIGGRAPH*, 1991, pp. 61-69.

[Turk92] G. Turk. Re-Tiling Polygonal Surfaces. *ACM SIGGRAPH*, 1992, pp. 55-64.

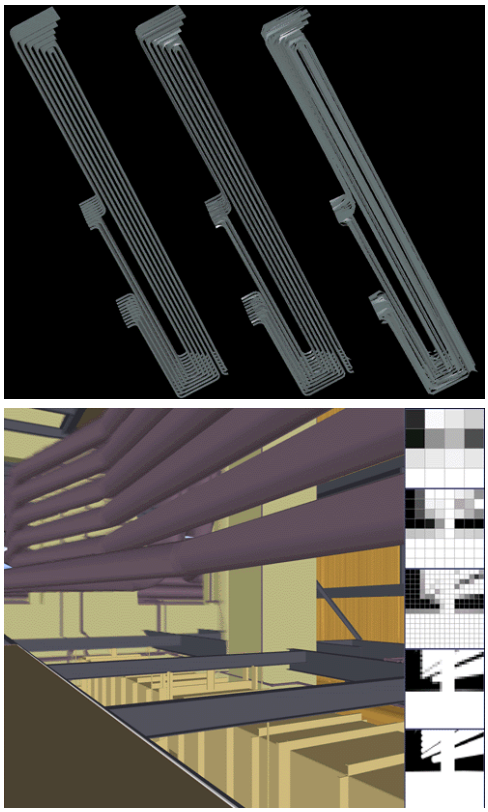
[Zhang97] H. Zhang, D. Manocha, T. Hudson, and K. Hoff. Visibility Culling Using Hierarchical Occlusion Maps. *ACM SIGGRAPH*, 1997, pp. 77-88.



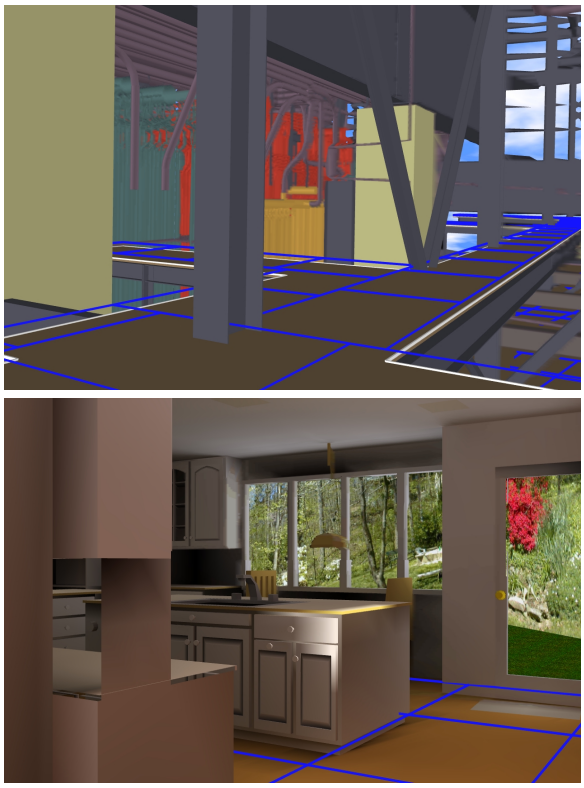
Color Figure A. *Bird's eye view of a power plant model. We have outlined in white the subset of a power plant rendered as geometry for an example viewpoint inside a cull box (model courtesy of James Close and Combustion Engineering, Inc.). This geometry is rendered using levels of detail and occlusion culling. Darker colored geometry is replaced by a fast image-based representation.*



Color Figure B. *Example TDM. Textured depth meshes replace distant geometry. Polygons outlined in white are part of a TDM. Near geometry is rendered conventionally.*



Color Figure C. *Example Rendering Acceleration Methods: (top) LOD - multiple geometric levels of detail are computed for complex objects, (bottom) occlusion culling - the flooring, column, and a portion of the pipes function as occluders that allow the hidden geometry to be safely culled (a hierarchical set of occlusion maps are displayed on the right side of the screen).*



Color Figure D. *Example Scenes: (top) power plant, (bottom) architectural model. The base of the virtual cells are outlined in blue. In the power plant, the far pipes and walkways are TDMs, nearer pipes have been LOD-simplified, and tan-colored columns make good occluders. In the architectural model, the far wall and cabinets are actually a TDM.*