New Techniques for Out-Of-Core Visualization of Large Datasets

Wagner Toledo Corrêa

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Program in
Computer Science

January 2004

Abstract

We present a practical system to visualize large datasets interactively on commodity PCs. Interactive visualization has applications in many areas, including computer-aided design, engineering, entertainment, and training. Traditionally, visualization of large datasets has required expensive high-end graphics workstations. Recently, with the exponential trend of higher performance and lower cost of PC graphics cards, inexpensive PCs are becoming an attractive alternative to high-end machines. But a barrier in exploiting this potential is the small memory size of typical PCs.

Our system uses new out-of-core techniques to visualize datasets much larger than main memory. In a preprocessing phase, we build a hierarchical decomposition of the dataset using an octree, precompute coefficients used for visibility determination, and create levels of detail. At runtime, we use multiple threads to overlap visibility computation, cache management, and rasterization. The structure of the octree and the visibility coefficients are kept in main memory. The contents of the octree nodes are loaded on demand from disk into a cache. To find the visible set, we use a fast approximate algorithm followed by a hardware-assisted conservative algorithm. To hide I/O latency, a separate thread prefetches nodes that are likely to become visible.

We also describe a sort-first parallel extension of the system that uses a cluster of PCs to drive a high-resolution, multi-tile screen. A client process interacts with the user, and a set of server processes render the screen tiles. To avoid sending the entire dataset from the client to the severs every frame, the servers access the dataset from a shared file system or from a local copy on disk. Putting the I/O load on the server side makes the network bandwidth requirements low and the architecture scalable.

Using a cluster of 8 PCs, the system can generate high resolution images (10 megapixels) of large datasets (12 gigabytes) at interactive frame rates (5–10 frames per second). Thus, our system is a cost-effective alternative to high-end machines, and can help bring visualization of large datasets to a broader audience.

Acknowledgements

Many people helped me to write this dissertation. I thank my advisors Szymon Rusinkiewicz and Cláudio Silva for their guidance, encouragement, and patience. I thank the other thesis committee members, Bernard Chazelle, Brian Kernighan, and Kai Li, for their feedback. I thank David Dobkin for managing this committee.

The technical work I present here was mainly done under the supervision of Cláudio Silva. For the last two and a half years, I have worked closely with him at AT&T and OGI. He devoted a huge amount of time to me, and gave me opportunities to work in industry and collaborate with other researchers.

A very special thanks goes to James Klosowski, who was my co-author in many papers and my mentor at IBM Research.

Another very special thanks goes to my dearest friend, Jeff Korn, with whom I shared the best and worst times during a very turbulent graduate program.

I thank Kai Li for giving me a vote of confidence when I most needed, and for being the most inspiring teacher I have ever had.

I thank Melissa Lawson for her help with academic, legal, and personal matters.

I thank Daniel Aliaga for his encouragement at the early stages of this work.

I thank my other collaborators: António Baptista, Louis Bavoil, Adam Finkelstein, Shachar Fleishman, Thomas Funkhouser, Robert Jensen, Walter Jiménez, Michael Kazhdan, Allison Klein, Wilmot Li, Tim Milliron, Manuel Oliveira, Dirce Pesco, Sinésio Pesco, Lourena Rocha, Susan Thayer, and Jianning Wang.

I thank the other faculty members who helped me: Andrew Appel, Perry Cook, Per Mykland, Ben Shedd, Mona Singh, and Jaswinder Pal Singh.

I thank Sandy Barbu, Rebecca Davies, Patricia Florek, Tina McCoy, Jennifer McNabb, and Patricia Killian for taking care of bureaucratic and legal matters.

I thank my friends in Brazil who kept in touch with me, and help me to stay sane: Vanessa Almeida, Márcio Araújo, Juliana Assumpção, Elissandra Barbosa, Rodrigo Carceroni, João Comba, Ricardo Farias, Rodrigo de Lélis, Maurício Lima, José da Mata, Wagner Meira Jr., Adriana Oliveira, Oscar Tibúrcio, and Marcelo Walter.

I also thank the friends I made in the US: Ian Buck, Jennifer Colwell, Regis Colwell, Alessandra Dickovick, Tyler Dickovick, Pavel Diko, Ben Dressner, Georg Essl, Jon Forsyth, Juliana Freire, Ben Gum, Greg Humphreys, Robert Kalnins, Jeff Korn, Aaron Lee, Patrick Min, Robert Osada, Lena Petrović, Emil Constantin Praun, Mukund Raghavachari, Rudrajit Samanta, Erika Ruiz Sandoval, Nathalie Schulten, Ashley Shelby, Robert Shillingsburg, Yefim Shuf, Kimberly Smith, Dan Wallach, Daniel Wang, Matthew Webb, Jacqueline Wells, and Xiaodong Wen.

I thank the people who I worked with at AT&T: James Abello, David Belanger, David Korn, Jeff Korn, Eleftherios Koutsofios, Balachander Krishnamurthy, Shankar Krishnan, Stephen North, and Suresh Venkatasubramanian.

I also thank the people I worked with at IBM: Gregory Abram, Thomas Jackman, Peter Kirchner, James Klosowski, and Frederick Mintzer.

I thank UNC Chapel Hill, Stanford University, UC Berkeley, 3rdTech (Lars Nyland), Lawrence Livermore National Laboratory (Randy Frank), Sandia (Kenneth Moreland), and Boeing (David Kasik) for datasets.

I thank CNPq, Princeton, Bell Labs, AT&T, OGI, and IBM for financial support.

Above all, I thank my family, Pedro, Mercês, and Érica, for their patience with my absence and for their constant and unconditional love.

To my parents, Pedro and Mercês.

Contents

Chapter 1

Introduction

This dissertation is about a set of new techniques for interactive visualization of large datasets on inexpensive PCs. Is this chapter we state this problem more precisely, defining what we mean by interactive, large, and inexpensive. We also explain why we care about large datasets, why we want to use inexpensive PCs to visualize them, and what is challenging about solving this problem. We then present a high-level view of our approach to solve this problem, and outline the remainder of the dissertation.

1.1 Goal

The goal behind this dissertation is making interactive visualization of large datasets viable on inexpensive commodity PCs. Throughout this dissertation we use the term *interactive* visualization when we mean visualization with a target rendering speed of 10 or more frames per second (fps). We reserve the term *real-time* visualization for when we mean visualization with a target of 30 or more fps. We use the term *large* datasets to refer to a dataset that is larger than the main memory available on the PC being used. And we use the term *inexpensive* PC to refer to a PC that costs less than US$2,000. We assume that this price includes a graphics card.

1.2 Motivations

Why do we care about visualization of large datasets? And why do we want to use PCs for that? We care about visualization of large datasets because it has applications in many areas, including:

- computer-aided design and engineering
- visualization of medical data
- modeling and simulation of weapons
- modeling and simulation of weather and ecosystems
- exploration of oil and gas
- virtual training

We want to use commodity PCs to visualize large datasets mainly because PCs have a better price/performance ratio than the alternatives. Traditionally, visualization of large datasets has required expensive high-end graphics workstations. Recently, with the exponential trend of higher performance and lower cost of PC graphics cards, inexpensive PCs are becoming an attractive alternative to high-end machines.

1.3 Challenges

Performing visualization of large datasets on commodity PCs is difficult. The main challenge is the gap that exists between the size of the main memory of a commodity PC and the size of "interesting" datasets. Of course, what is a commodity PC is a moving target, and what is interesting is subjective. To make this discussion more concrete, consider the year 2003. A typical PC has about 512 MB of main memory, while a machine with 16 GB of main memory would be considered high-end. Still, a numerical weather simulation would have no trouble producing hundreds of gigabytes of data. The ubiquitous 32-bit PC cannot even address that much memory.

Not only does the gap between dataset and main memory sizes exist, but also it is widening. Although memory sizes are growing exponentially, roughly doubling every 18 months, dataset sizes are growing faster. It is easier to produce or acquire more data than to improve and lower the costs of main memory technology.

To bridge this gap, we need to develop *out-of-core*[1] algorithms, also known as external algorithms or secondary-memory algorithms. Out-of-core algorithms keep the bulk of the data on disk, and keep in main memory (or in core) only the part of the data that is being processed.

Adapting an existing in-core algorithm to work out-of-core is not trivial. Partial solutions such as paging or virtual memory are not sufficient [32, 101]. Because disk access latencies are five to six orders of magnitude greater than main memory access latencies [151], an out-of-core program is likely to have its running time dominated by disk operations, and may run many times more slowly than its in-core counterpart. To avoid severe performance degradation, an out-of-core program should try to minimize the number of disk operations and hide the disk latency by performing disk operations concurrently with other operations. The performance of out-of-core programs can be greatly improved by organizing the data in a way that increases locality of reference and by prefetching data from disk into memory before it is needed [54].

Besides the relative small memory, another limitation of commodity PCs that makes visualization of large datasets difficult is the availability of only one graphics card per PC. High-end graphics workstations such as the Silicon Graphics Onyx4 UltimateVision [128] can have up to 32 graphics pipes. Having only a single graphics

[1]The word "core" is an old-fashioned term for main memory. It dates back to the days (1961–1971) of ferrite core memory, an early form of non-volatile storage built by hand from tiny rings of magnetizable material threaded onto very fine wire to form large (e.g., 13"x13") rectangular arrays. Each core stored one bit of data. The related expression "core dump" refers to a copy of the contents of the memory, produced when a process is terminated by certain kinds of internal error [64].

pipe limits our choice of algorithms. In a multi-pipe system, multiple tasks that need to access the graphics hardware could run in parallel. On a single-pipe system, these operations would have to run sequentially.

Yet another limitation of commodity PCs is low display resolution. When interacting with large datasets, it is natural to want to visualize these datasets at high resolution. A high resolution image can give us insights that we would not gain by looking at separate low resolutions images. For example, compare our ability to understand a map on a large 34"x44" sheet of paper versus a booklet with 16 regular pages, 8.5"x11" each. Similarly, looking at a 4096×3072-pixel image of a dataset at once is much more informative than scrolling through it with a 1024×768-pixel window. Looking at a large image helps us to see the big picture.

1.4 Solutions

In this dissertation we present a system that allows us to use commodity PCs to visualize datasets much larger than main memory at interactive frame rates and at high resolution. The system uses a set of new out-of-core techniques that are simple and yet effective at hiding the weaknesses of PCs and exploiting the strengths of PC graphics cards. Considered in isolation, each of these techniques is pretty straightforward. The combination of these techniques allows us to build a system that works under our constraints and satisfies our goals (the whole is greater than the sum of its parts).

The process of visualizing a dataset using our system consists of a pipeline of steps that can be broken down into two major phases: preprocessing and rendering (Figure 1.1). In the preprocessing phase, we first build a hierarchical spatial decomposition of the dataset using an out-of-core octree. Then, we compute directional visibility coefficients for each octree node. These coefficients are used at runtime for

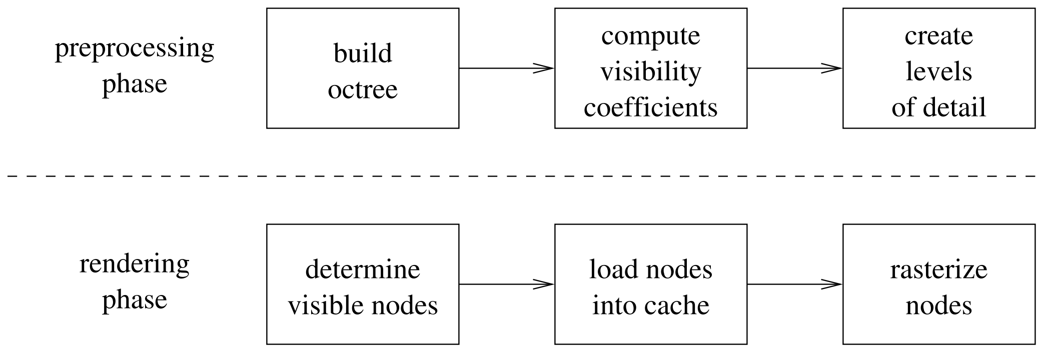


Figure 1.1: The main steps in each phase of our visualization system's pipeline.

fast and accurate approximate visible set computation. Finally, we create several levels of detail for each octree node.

In the rendering phase, our system uses multiple threads (typically running on a single processor) to overlap visibility computation, cache management, and rasterization. The system keeps in main memory a description of the structure of the octree and the coverage coefficients. The contents of the octree nodes, which are the bulk of the data, are kept on disk, and are brought into the geometry cache in main memory when needed. The cache uses a least-recently-used replacement policy, which exploits well the frame-to-frame coherence typical of interactive visualization sessions.

The computation of the visible set is done in two steps. First, a fast approximate visibility algorithm determines an initial guess of the visible set. Then, a hardware-assisted algorithm augments this set to make it a conservative visible set. To hide the cost of disk operations, a look-ahead thread guesses the nodes that the user may see next, and prefetches those nodes into the geometry cache.

All the steps in both the preprocessing and rendering phases are implemented using out-of-core techniques so that the system can run on a PC with small memory. These techniques assume that the dataset is static (i.e., the geometric information does not change over time), and favor interactivity over image quality. The images produced by our system have Goraud-shading quality, as supported by the graphics card, which is acceptable for a previewer.

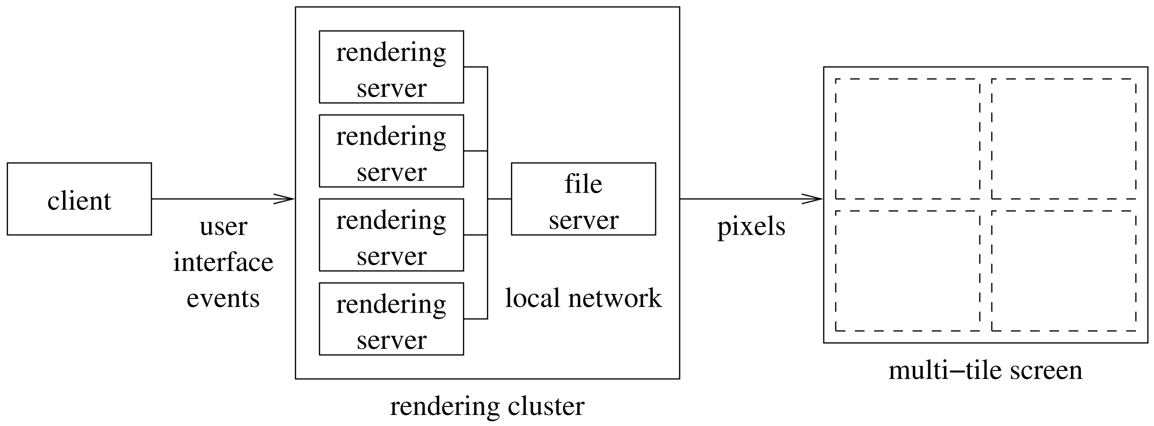


Figure 1.2: The sort-first parallel extension of our visualization system.

In this dissertation we also describe a sort-first parallel extension of the system (Figure 1.2). This extension allows us to use a cluster of PCs to drive a multi-tile screen to generate high resolution images at interactive frame rates. When running on a cluster, the system consists of a client process, possibly running on a remote machine, and many interconnected server processes, each rendering a tile of the screen.

To avoid sending the entire dataset from the client to the severs at every frame, which would create a bottleneck and prevent interactivity, the servers access the dataset from a shared file system or from a local copy on disk. The client only needs to send user interface commands to the severs, and the servers only need to synchronize with each other at the end of each frame. Each rendering server is responsible for determining the data it needs, and for pulling the data from disk into its cache. Putting the I/O load on the server side lowers the network bandwidth requirements, and makes the architecture scalable and practical.

Using this system we were able to use a small cluster (8 PCs) to generate high resolution images (10 megapixels) of large datasets (12 gigabytes) at interactive frames (5–10 frames per second). These results demonstrate that our system is a cost-effective alternative to high-end machines, and can help bring visualization of large datasets to a broader group of people.

1.5 Outline of the Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 reviews background material and previous works related to ours. Chapter 3 describes the out-of-core algorithms used in the preprocessing phase to build an octree for a given dataset, precompute visibility coefficients, and create levels of detail. Chapter 4 describes the out-of-core, multi-threaded algorithms used in the rendering phase to compute visibility, manage the memory cache, and rasterize the dataset. Chapter 5 describes the sort-first parallel extension of the rendering algorithms used to produce high-resolution images of the dataset on a multi-tile screen driven by a cluster of PCs. Finally, Chapter 6 presents conclusions and discusses directions for future work.

Chapter 2

Related Work

In this chapter we review background material and previous works related to ours. We start by discussing techniques related to management of large datasets, optimization of the graphics pipeline, and parallel rendering. We then present a chart comparing our system to previous related systems based on the set of techniques they use. We finish by discussing the reasons why we chose the techniques we use.

2.1 Management of Large Datasets

The general approach to handle datasets larger than main memory is to break the dataset into manageable pieces, and bring the appropriate level of detail of each piece of the dataset into memory on demand. Breaking the dataset into pieces is known as spatialization. Precomputing levels of detail is known as simplification. Managing what pieces come in and out of memory involves caching and prefetching.

2.1.1 Spatialization

Spatialization is the process of creating a spatial subdivision for the geometric data of a given dataset. There are many different kinds of spatial data structures: octree,

k-d tree, BSP tree, hierarchy of boxes, hierarchy of spheres, and many others [119, 140]. Using these data structures, we can speed up searches and traversals by pruning entire subtrees of the dataset, thus avoiding unnecessary computation.

Spatial data structures have been used successfully in many commercial and academic graphics systems. Octrees have been used in innumerable contexts, including view-frustum culling [23], occlusion culling [56], ray tracing [71], and volume rendering [76]. SGI's Optimizer [126] uses a hierarchy of boxes to spatialize the scene graph. Id Software's Quake 3 game [134] uses a BSP tree. The QSplat system of Rusinkiewicz and Levoy [113] uses a hierarchy of spheres.

Spatial data structures are particularly useful for visualization of datasets larger than main memory. If we have a spatial partitioning of the dataset, we can render the entire dataset, one part at time, as long as each part is small enough to fit in main memory. But how do we create the spatial partitioning in the first place?

The database literature uses the term *bulk loading* to refer to the out-of-core construction of spatial data structures. Agarwal et al. [2] and Arge et al. [8] present bulk loading algorithms for many spatial data structures, including k-d tree, quad-tree, and R-tree.

In Chapter 3 we present a fast and incremental out-of-core algorithm to build an octree whose leaves contain the geometry of a given dataset. The algorithm imposes a limit on the number of geometric primitives per leaf, and saves each leaf in a separate file in a hierarchy of directories. The algorithm also creates a small separate file that contains the overall structure of the octree. Our algorithm is similar to the algorithms of Cignoni et al. [22] and Ueng et al. [141], but there are some differences which we will discuss in Chapter 3.

2.1.2 Simplification

Another technique to deal with large datasets is simplification, which consists of precomputing approximate versions of the dataset known as levels of detail (LODs). Levels of detail can be discrete, continuous, or view-dependent.

Systems that use discrete levels of detail (also known as static levels of detail) precompute several simplified versions of each object or partition of the dataset, and at runtime display the most appropriate version based on selection criteria such as the distance to the viewer [23, 46, 49, 52, 112]. Static LODs may cause disturbing artifacts when switching from one level to another, but they are easy to precompute and impose very little overhead at runtime.

Systems that use continuous (or progressive) levels of detail precompute a continuous-resolution representation of the dataset that allows smooth transition between approximations [40, 44, 61, 83, 155]. Continuous LODs take longer to compute, and have higher runtime overhead than static LODs, but they produce images with higher fidelity for a given polygon budget.

Systems that use view-dependent levels of detail also use a continuous-resolution representation of the dataset. In addition, these systems allow a single object to have multiple levels of detail at the same time, and select higher resolutions for parts closer to the viewer and lower resolutions for parts farther from the viewer [45, 62, 86].

Recall that one of the motivations to compute simplified versions of a large dataset is to be able to display it on a machine with small memory. If we want to use the same machine to compute the simplified versions, the simplification algorithm itself needs to be out-of-core [81, 84].

In our system we use static levels of detail, precomputed using a vertex clustering technique similar to the one of Rossignac and Borrel [16, 112, 122]. In Chapter 3 we discuss this technique in more detail. For more information on LODs, we refer the reader to the recent book by Luebke et al. [87]

2.1.3 Geometry Caching and Prefetching

A critical part of any system for visualization of datasets larger than main memory is, of course, the memory management subsystem. A simple and effective approach is to keep in main memory the least-recently used (LRU) pieces of geometry [137]. This approach is particularly effective if the pieces of the dataset that are visible in any given frame fit together in the cache, and there is locality of reference, i.e., the changes in visibility from frame to frame are small.

Caching alone is typically not enough to deliver smooth frame rates. Even small changes in visibility may cause the system to stall because of bursts of disk activity. The resulting frame rates may be low and with high variance, which prevent a smooth interaction with the dataset.

One technique to alleviate this problem is speculative prefetching, which tries to bring into memory the pieces of geometry that will become visible "soon." What is considered soon may be difficult to define. We want to have the piece of the dataset that we are interested in ready in memory when we need it, but we also want to avoid polluting the cache with too many pieces that will end up not being used [108].

Prefetching is not a novel idea, and has been used in operating systems for decades [54]. In computer graphics, Funkhouser et al. [50] were one of the first to incorporate prefetching into a visualization system for large datasets. Their system partitioned the dataset into cells, and precomputed the cells that could be visible from within each cell. Whenever a user entered a cell, all other cells potentially visible from that cell would be prefetched.

The system we present here also employs prefetching, but we do not precompute cell-to-cell visibility. Instead, we estimate which cells may become visible for each position of the user at runtime. Our approach takes less preprocessing time, and produces a tighter estimate of the set of cells to be prefetched. We will discuss in more detail the differences between these two approaches shortly.

Most visualization systems try to insulate the high-level software layers of the application from the low-level layers that perform database management. A prime example of such an approach is the active data repository (ADR) of Kurc et al. [75]. The ADR framework manages the dataset stored in one or more disks, and provides an application with modular services for memory management, data retrieval, and scheduling of processes.

2.2 Graphics Pipeline Optimization

Over the years, graphics researchers have accumulated a large number of techniques to optimize rendering. These techniques include:

- back-face culling
- view-frustum culling
- occlusion culling
- detail culling
- image-based rendering
- point rendering
- hardware-assisted rendering
- computation reordering

The next subsections discuss each of these techniques. The discussion is intentionally brief. The goal is not to explain each technique in detail, but to provide the minimum background necessary to understand the techniques we chose to use in our system.

2.2.1 Back-Face Culling

Back-face culling means not rendering geometry that faces away from the user (avoiding unnecessary computation). Implementing back-face culling is trivial, and consists of a simple dot product between the face normal and the viewing direction. The OpenGL library [154] has a flag to enable back-face culling (GL_CULL_FACE). It is also possible to use spatial data structures to perform hierarchical back-face culling.

2.2.2 View-Frustum Culling

View-frustum culling means not rendering geometry that is outside the field of view of the user's camera (again, avoiding unnecessary computation). Implementing view-frustum culling is pretty easy as well, and typically consists of checking bounding volumes (such as boxes or spheres) against the planes that define the viewing frustum. Möller and Haines [93] discuss several algorithms for volume/frustum intersection.

We can use a hierarchical spatial partitioning of the dataset to speed up view-frustum culling [23]. Whenever a node is totally outside (or totally inside) the view-frustum, all of its descendants also are.

2.2.3 Occlusion Culling

Another technique to avoid unnecessary computation is occlusion culling, which means not rendering geometry hidden by other geometry, or in other words, only rendering the geometry that is visible. Unlike back-face culling and view-frustum culling, occlusion culling is difficult to implement. Visible surface determination is a hard problem that has been studied for decades [136].

In their survey on visibility algorithms, Cohen-Or et al. [24] classify visibility algorithms according to several criteria. Here we briefly summarize the criteria that are most relevant to this dissertation:

From-point vs. from-region: Some algorithms compute visibility from the eyepoint only, while others compute visibility from a region in space. Since the user often stays for a while in the same region, the from-region algorithms amortize the cost of visibility computations over a number of frames.

Precomputed vs. online: Many algorithms require an offline computation, while others work on the fly. For example, from-region algorithms require a preprocessing step to divide the model in regions and compute region visibility. From-point algorithms typically compute visibility at runtime.

Object space vs. image space: Some algorithms (e.g., ray tracing) compute visibility in object space, using the 3D primitives. Others (e.g., Z-buffer) operate in image space, using the discrete rasterization fragments of the primitives.

Conservative vs. approximate: Few visibility algorithms compute exact visibility. Most algorithms are conservative, and overestimate the set of visible primitives. Other algorithms compute approximate visibility, and do not guarantee finding all visible primitives.

The visibility algorithm most relevant to this dissertation is the prioritized-layered projection (PLP) algorithm of Klosowski and Silva [73]. PLP is an approximate, from-point, object-space visibility algorithm that requires very little preprocessing. The preprocessing consists of building a spatial partitioning for the dataset and computing simple statistics for each cell. At runtime PLP uses heuristics to estimate how likely it is for each cell to be visible, and adds cells to an approximate visible set up to a user-defined budget of geometry to be rendered per frame. Klosowski and Silva also developed cPLP [74], a conservative, image-space algorithm that uses PLP to obtain an initial guess, and then augments the approximate visible set to make it conservative. In Chapters 3 and 4 we will discuss PLP and cPLP in more detail, and present the extensions we have made to these algorithms.

For a detailed and comprehensive survey on visibility algorithms, please consult the article of Cohen-Or et al. [24] Here we limit ourselves to briefly mentioning some of these algorithms to illustrate the main differences between them and PLP/cPLP.

Teller et al. [138] developed the from-region visibility algorithm that was used by Funkhouser et al. [50] in their walkthrough system. The algorithm of Teller et al. requires long preprocessing times, and assumes that the models are made of axis-aligned cells. In contrast, PLP and cPLP require very little preprocessing, and make no assumptions about the geometry of the model.

Wonka et al. [152] presented a from-region visibility preprocessing algorithm with occluder fusion. Their algorithm used 2 processors to overlap visibility computation and rendering at runtime (similarly to Garlick et al. [53]). The algorithm required long preprocessing times (9 hours for a model with 8 million triangles), and was limited to 2.5D datasets. In later work, Wonka et al. [153] used a from-point approach that needed little preprocessing, but they only reported results for in-core, 2.5D datasets.

Durand et al. [41] presented a from-region visibility preprocessing algorithm that could handle 3D environments, as opposed to 2.5D [152], but the algorithm required long preprocessing times (33 hours for a model with 6 million triangles). Schaufler et al. [120] also presented a from-region 3D visibility preprocessing algorithm, but their largest test model had only 0.6 million triangles.

Chhugani et al. [20] developed a system that precomputes from-region visibility and levels of detail per region. Their system focuses on image accuracy, and is able to interactively render large datasets with less than one pixel of screen-space deviation and correct visibility. Unfortunately, for a model with 13 million triangles, and using a cluster of 16 PCs, the accuracy guarantee costs 128 hours of precomputation.

Hall-Holt and Rusinkiewicz [58] developed the visible zone algorithm for conservative visibility computation with incremental updates. Their algorithm is able to achieve real-time frame rates for 2D and 2.5D datasets.

2.2.4 Detail Culling

Detail culling means not rendering geometric details that are likely to be unimportant to the final image. Detail culling relates to the generic strategy of computing an answer for a problem at the lowest acceptable accuracy. Detail culling is also known as level-of-detail (LOD) management.

As we have discussed above, typically LOD data structures are precomputed. At runtime, the rendering engine selects the appropriate level of detail. Funkhouser and Séquin [49] described LOD management as an optimization problem that tries to maximize image quality (benefit) given the time and geometry constraints (costs). Avila and Schroeder [9] and El-Sana and Chiang [42] also developed systems for interactive out-of-core rendering based on LOD management. Andújar et al. [7] and El-Sana et al. [43] have combined level of detail management with occlusion culling in in-core rendering systems.

Continuous and view-dependent LODs tend to produce images with better quality than static LODs, but static LODs are more appropriate for today's graphics hardware. It is much faster to use display lists or vertex arrays [154] to display a static LOD than to loop through the individual triangles of a continuous LOD.

2.2.5 Image-Based Rendering

Image-based rendering techniques generate new image from precomputed samples of the plenoptic function [1]. The plenoptic function is a 7D function that returns the color visible from point (p_x, p_y, p_z) and direction (v_x, v_y, v_z) at time t. Because of its high dimensionality, densely sampling this function is not feasible, and researchers have investigated using sparse samplings of lower-dimension slices of this function.

The lumigraph [55] and the light field [77] data structures are samplings of 4D slices of the plenoptic function. The lower dimensionality comes from fixing t and limiting the user to look at a convex object from the outside. The concentric mosaics data structure [131] is a sampling of a 3D slice of the plenoptic function that confines the viewing position on a plane and uses a single angle to define the viewing direction.

Precomputed images of synthetic models or photographs of real environments can be combined with approximate geometry to generate a sampling of a 4D slice of the plenoptic function [36, 37, 72]. A single image used for texture mapping [149] can be thought of as a sampling of a 2D slice of the plenoptic function. Many systems have used image impostors to replace geometry and accelerate rendering [5, 6, 34, 38, 88, 91, 129, 130, 132]. Image impostors can be thought of as a special case of LOD.

Image-based rendering techniques have the potential to simultaneously ease modeling and speed up rendering. In particular, these techniques can deliver very high quality images at an almost constant cost per image. Unfortunately, preprocessing requirements for image-based rendering techniques to handle large datasets are very high. Our system does not use any image-based technique.

There is a large number of IBR techniques, covering a spectrum from pure geometry to pure imagery. A detailed survey of these techniques is outside the scope of this dissertation. For further information, we refer the reader to the SIGGRAPH course notes on image-based rendering [35].

2.2.6 Point Rendering

Large datasets may have many more polygons than the available screen has pixels. As a consequence, many triangles may have a projected area smaller than a pixel. In this case, it makes sense to render point samples instead of triangles. Recently, many researchers have developed point-based rendering systems [13, 57, 78, 104, 113, 121].

Among these systems, the QSplat system [113] and its extension for streaming datasets over a network [114] share our goal of visualizing datasets larger than main memory on commodity hardware. Point rendering has also been used to render 3D surfaces [33, 79] and fuzzy objects such as clouds, fire, and plants [15, 109, 133].

2.2.7 Hardware-Assisted Rendering

As graphics algorithms mature, their implementations become available in hardware through simple application programming interfaces (APIs) such as OpenGL [154]. Graphics cards are getting faster and more sophisticated at an amazing rate, and exploiting the new algorithms available in hardware through OpenGL extensions is key to developing competitive systems. Two examples of OpenGL extensions that we exploit in our system are vertex arrays and occlusion queries.

The vertex array extension uses blocks of vertices, colors, and normals to draw primitives. The types of primitives include points, lines, triangles, triangle fans, and triangle strips. The vertex array extension allows us to setup pointers to blocks of data, and then call a single function (glDrawElements) that takes care of transferring the data from main memory to graphics card memory, and then rendering it. Rendering using glDrawElements is typically much faster than looping over the data and calling the OpenGL functions for each vertex.

There are several types of occlusion query extensions. The HP occlusion test [124], lets us send a piece of geometry to the graphics hardware, and ask if that piece of geometry would have been visible. A more sophisticated extension, the NVIDIA occlusion query [110], lets us send various pieces of geometry to the graphics hardware at the same time, and get for each of them the number of pixels that would have been affected. In Chapter 4 we describe how we exploit these extensions to accelerate conservative occlusion culling.

2.2.8 Computation Reordering

Sometimes a given computation consists of independent operations, and the final result does not depend on the order in which the operations are executed. In this case, it may be possible to reschedule the execution of the operations to exploit coherence and obtain substantial performance improvements.

Pharr et al. [105] developed a ray tracing system for datasets larger than memory that employed computation reordering. They achieved large rendering speedups by rescheduling the ray intersection computations. Our system is different from theirs in some aspects: they focus on photorealism, while we focus on interactivity; and they use a regular grid to spatialize the dataset, while we use an octree. But our systems share a basic idea: do as much computation as possible with the data currently in memory, and delay computations that need data currently on disk. In particular, our rasterization phase does not rasterize the visible octree nodes in a fixed order. Instead, the nodes in memory are rasterized first, while nodes on disk are being fetched to be rasterized later. The final image is unaffected by the out-of-order execution, because the Z-buffer algorithm sorts the primitives at the pixel level.

Another way improve rendering performance is by doing attribute clustering. Typically, the rendering engine keeps track of a rendering state, which includes attributes such as the current material (for example, that is how OpenGL works [154]). If many primitives share the same attributes, it is usually faster to render them together, because we then save time that would be wasted on context switches.

A similar technique is mode sorting. Suppose that some primitives in the dataset are to be rendered as polygons, and other primitives are to be rendered as lines. Switching from polygon rendering mode to line rendering mode takes time. If we reorder the traversal of the primitives to avoid mode changes, rendering will be faster.

2.3 Parallel Rendering

Researches have investigated the use of parallel machines for computer graphics for decades. In 1983, Ullner [142] presented a ray tracing machine. In 1990, Garlick et al. [53] presented the idea of exploiting multiprocessor workstations to overlap visibility computations with rendering.

Many other approaches to parallel rendering have been proposed over the years. Molnar et al. [94] classify parallelization strategies in three categories based on where in the rendering pipeline sorting for visible-surface determination takes place. Sorting may happen during geometry preprocessing, between geometry preprocessing and rasterization, or during rasterization. The three categories of parallelization strategies are sort-first, sort-middle, and sort-last:

Sort-first algorithms [66, 99, 117, 118] distribute raw primitives (with unknown screen-space coordinates) during geometry preprocessing. These approaches divide the 2D screen into disjoint regions (or tiles), and assign each region to a different processor, which is responsible for all of the rendering in its region. For each frame, a pre-transformation step determines the regions in which each primitive falls. Then a redistribution step transfers the primitives to the appropriate renderers. Sort-first approaches take advantage of frame-to-frame coherence well, since few primitives tend to move between tiles from one frame to the next. Sort-first algorithms can also use any rendering algorithm, since each processor has all the information it needs to do a complete rendering. Furthermore, as rendering algorithms advance, sort-first approaches can take full advantage of them. One disadvantage of sort-first is that primitives may cluster into regions, causing load balancing problems between renderers. Another disadvantage is that more than one renderer may process the same primitive if it overlaps screen region boundaries.

Sort-middle algorithms [4, 47, 96] distribute screen-space primitives between the geometry preprocessing and rasterization stages. Sort-middle approaches assign an arbitrary subset of primitives to each geometry processor, and a portion of the screen to each rasterizer. A geometry processor transforms and lights its primitives, and then sends them to the appropriate rasterizers. Until recently, this approach has been the most popular due to the availability of high-end graphics machines. One disadvantage of sort-middle approaches is that primitives may be distributed unevenly over the screen, causing load imbalance between rasterizers. Sort-middle also requires high bandwidth for the transfer of data between the geometry processing and rasterization stages.

Sort-last approaches [59, 95, 148] distribute pixels during rasterization. They assign an arbitrary subset of the primitives to each renderer. A renderer computes pixel values for its subset, no matter where they fall in the screen, and then transfer these pixels (color and depth values) to compositing processors. Sort-last approaches scale well with respect to the number of primitives, since they render each primitive exactly once. On the other hand, they need a high bandwidth network to handle all the pixel transfers. Another disadvantage of sort-last approaches is that they only determine the final depth of a pixel during the composition phase, and therefore make it difficult (if not impossible) to use certain rendering algorithms, e.g., transparency and anti-aliasing.

Here we will focus on recent parallel rendering systems, especially on systems geared towards using clusters of commodity PCs and rendering on multi-tile displays.

Samanta et al. [117, 118] developed a sort-first rendering system using a network of commodity PCs. The main focus of their work was on load balancing the geometry processing and rasterization work done on each of the PCs, rather than on handling very large models. To achieve a well balanced system, they developed dynamic screen partitioning schemes that predict the rendering costs of groups of triangles and attempt to minimize the amount of overlap between triangles and screen partitions. A limitation of their system was that in some cases the screen partitioning scheme could become the bottleneck. Another limitation was the lack of scalability with respect to model size, as the model had to be replicated in main memory on each of the rendering nodes of their cluster.

In subsequent work, Samanta et al. [116] developed a hybrid sort-first/sort-last parallel rendering algorithm, which scaled better with processor count and screen resolution. Their new approach performs dynamic, view-dependent partitioning of both the 3D model and the 2D screen. The objectives that they are addressing are balancing the rendering load on the nodes as well as minimizing the screen space overlaps which require the subsequent pixel transfer and compositing step. Once again, the geometry is replicated on each of the nodes, and the dynamic partitioning phase could become a bottleneck and limit the frame rate.

In more recent work, Samanta et al. [115] address the replication problem, storing (in main memory) copies of the model only in k of the available n nodes, where $k < n$. Still, neither the preprocessing phase nor the rendering phase would be able to handle a model larger than main memory. The system we present here can handle arbitrarily large models (limited only by the size of the available secondary memory).

Mueller [99, 100] has performed extensive experiments using a sort-first rendering system. He emphasizes that sort-first has an advantage over sort-middle, because it can exploit the frame-to-frame coherence inherent in interactive applications. He also points out that sort-first has an advantage over sort-last, because it does not require high communication bandwidth for pixel traffic. Part of Mueller's work was on the load-balancing problem. He designed a dynamic scheme for partitioning the screen so that each processor has a balanced rendering load. His algorithm is the basis for the work of Samanta et al. [117]. Mueller also worked on the database management problem, focusing on retained-mode databases that fit in the memory of the graphics hardware. In contrast, we focus our work on immediate-mode databases that are larger than the main memory of the host hardware.

WireGL [18, 65, 66, 67] is a system that allows the output resolution of an unmodified graphics application to be scaled to the resolution of a tiled display, with little or no loss in performance. WireGL replaces the OpenGL driver on the client machine, intercepts the OpenGL calls, and sends the calls over a high-speed network to servers which render the geometry. WireGL includes an efficient network protocol, a geometry bucketing scheme, and an OpenGL state tracking algorithm. WireGL is able to sustain rendering performance of over 70 million triangles per second on a 32-node cluster. It assumes, however, that the entire model fits in the main memory of the client machine. Another limitation is that the geometry bucketing algorithm assumes that the geometry primitives that are close to each other in the GL stream are also close together spatially, which may not be the case.

Chromium [68] is a system that, as WireGL, replaces the OpenGL driver. Chromium is much more flexible than WireGL, and lets a programmer create applications using stream processing units (SPUs). For example, a “pack” SPU on the client side intercepts the OpenGL calls, packs the OpenGL stream into buckets, and sends the buckets over the network to a rendering server. The rendering server unpacks the OpenGL stream from the network, and uses a “render” SPU to generate pixels. SPUs are free to change the OpenGL stream, and can be chained. For example, there are SPUs to invert the colors, or add alpha blending, or display hidden lines.

Chromium has been used to re-implement WireGL, and implement other sort-first and sort-last systems [14]. One disadvantage of Chromium is that it does not have built-in support for large datasets. Although it is conceivable to use Chromium to build an out-of-core rendering system, such a system does not yet exist. Another disadvantage of Chromium is that if the client application does not (or cannot) exploit display lists, the application performance will suffer. In our experience, even when Chromium is running on the client machine (making the network overhead disappear), an immediate mode application typically achieves only 10% of its native performance.

Lombeyda et al. [85] developed a parallel system for interactive volume rendering using commodity hardware. Zhang et al. [157] employed a cluster of PCs for visualization of isosurface of massive datasets.

2.4 Related Systems

Is this section we compare our system to previously published systems. When evaluating a system, we asked the following questions:

- Can it handle large datasets?
- Does it run on commodity PCs?
- Is the preprocessing fast?
- Does it use occlusion culling?
- Is occlusion culling from-point?
- Does it support LODs?
- Does it use image impostors?
- Does it use prefetching?
- Does it exploit hardware support?
- Does it render in high resolution?
- Can it handle arbitrary 3D models?
- Can it handle dynamic geometry?
- Does it run unmodified programs?

Table 2.1 summarizes the answers to these questions for the systems most related to ours. We now briefly review each of the systems in Table 2.1 in chronological order.

Clark [23] proposed back in 1976 many of the major techniques still used today by rendering systems. His ideas included hierarchical view frustum culling, hierarchical simplification and LOD management, hierarchical occlusion culling, and working set management (on-demand loading and least-recently-used replacement). It is unclear, however, whether or not Clark had a working system that implemented all his ideas.

Airey et al. [3] described a system that combined LOD management with the idea of precomputing visibility information for models made of axis-aligned polygons.

Funkhouser et al. [50] were the first to publish a system that supported models larger than main memory and performed speculative prefetching. Their approach relied on the from-region visibility algorithm of Teller et al. [138], which requires long

Year	System	Large datasets	PCs	Fast preprocessing	Occlusion culling	From-point occlusion	LOD	Image impostors	Prefetching	Hardware support	High resolution	Arbitrary 3D geometry	Dynamic geometry	Unmodified program
1976	Clark's	●			●		●					●		
1990	Airey's				●		●			●				
1993	Funkhouser's	●			●		●		●	●				
1996	VTK		●				●			●	●	●		
1997	Cox's	●		●			●			●		●		
1998	Optimizer	●	●	●	●	●	●			●	●	●		
1999	MMR	●			●		●	●	●	●		●		
2000	Prince's	●	●	●			●		●	●		●		
2000	QSplat	●	●	●			●		●	●	●	●		
2001	Jupiter	●	●	●	●		●			●		●		
2001	Moreland's	●	●	●			●			●	●	●		
2001	Samanta's		●	●						●	●	●		
2001	Wald's	●	●	●	●	●						●		
2002	Chromium		●	●						●	●	●	●	●
2002	GigaWalk				●	●	●			●		●		
2002	OpenSG	●	●	●			●			●		●		
2002	Varadhan's	●					●		●	●		●		
2002	XFastMesh	●	●	●			●			●		●		
2003	Lindstrom's	●	●	●			●			●		●		
2003	Wald's		●	●	●	●						●	●	
2003	Yoon's		●		●	●	●			●		●		
2003	iWalk (ours)	●	●	●	●	●	●		●	●	●	●		

Table 2.1: Comparison of systems related to ours.

preprocessing times, and assumes that the models are made of axis-aligned cells. Our approach is based on the from-point visibility algorithm of Klosowski and Silva [73], which requires very little preprocessing, and handles arbitrary 3D geometry.

The visualization toolkit (VTK) [123] is a generic collection of libraries and tools for development of rendering systems. Many systems have been built on top of VTK, but VTK has little support out-of-core rendering, thread-safety, and occlusion culling.

Cox [32] presented a paged segment system to manage the scene database cache. Cox showed that an application that controls paging itself achieves much better performance than an application that relies on the operating system's management of virtual memory.

The OpenGL Optimizer [125, 126] is a commercial package available from SGI that provides an application programming interface (API) for visualization of large models. There is a large overlap in goals between our system and Optimizer, but our methods differ. Unfortunately, Optimizer is expensive, geared towards high-end hardware, and it is not available for Linux. Optimizer is being discontinued and replaced by Performer [127], which is available for Linux, and we hope will eventually support all the features of Optimizer on commodity hardware. A similar product is TGS's commercial version of Open Inventor [139].

Aliaga et al. [5] developed the massive model rendering (MMR) system. MMR employed a large number of acceleration techniques, including replacing distant geometry with image impostors, managing levels of detail, and culling occluded geometry. MMR was perhaps the first published system to handle models with tens of millions of polygons at interactive frame rates. On the other hand, MMR required weeks of preprocessing time and expensive high-end graphics workstations. Our system requires much less preprocessing time, and runs on commodity PCs.

Prince [107] presented an out-of-core extension for the progressive meshes data structure [61]. Prince used a regular grid to spatialize the dataset, and did not use occlusion culling. His system relied on system calls of the Windows API to manage virtual memory, and was limited to datasets of at most 2GB on 32-bit machines. It is unclear how Prince implemented prefetching, because his system did not support asynchronous data loading. It is also unclear how well his system would perform for truly large datasets, because Prince only reported results for datasets that were smaller than the memory of the test machine.

Rusinkiewicz and Levoy [113] developed QSplat, a point-based rendering system for massive meshes. QSplat employs all the acceleration techniques our system employs, except for occlusion culling. QSplat is able to render billion-triangle meshes at interactive frame rates with very acceptable image quality. An extension of QSplat supports streaming massive meshes over a slow network connection [114].

Bartz et al. [11] presented the Jupiter toolkit for visualization of large datasets. Jupiter is a joint effort between HP and the University of Tübingen. The toolkit supports occlusion culling and level-of-detail management. Out-of-core and parallel rendering are currently being added.

Moreland et al. [98] presented a sort-last parallel rendering system for visualization of large datasets on a display wall driven by a cluster of PCs. Their system scales very well with data size, and is able to generate 12-megapixel images of a model with half a billion triangles at almost interactive frame rates.

Samanta et al. [115, 116, 117, 118] developed a parallel rendering system for display walls driven by a cluster of PCs. As we have mentioned, the focus of their research was on load balancing algorithms, not on out-of-core rendering.

Wald et al. [146] developed a ray tracing system for out-of-core rendering of large models on a cluster of PCs. A key difference between our work and theirs is that they use ray tracing, and we use the Z-buffer. Although ray tracing allows them to use more sophisticated rendering algorithms, the Z-buffer allows us to exploit better hardware support and produce higher resolution images.

Humphreys et al. [68] developed Chromium, which we have discussed in the previous section. We mention it here again to emphasize that Chromium's goal is to provide mechanisms, not algorithms. It is also important to note that Chromium can scale the resolution of an *unmodified* client application. This feature is important if the client application is only available in binary format, or if the application requires a commercial license per rendering node.

Baxter et al. [12] developed GigaWalk, an in-core rendering system for high-end machines that used multiple threads to combine occlusion culling with hierarchical level-of-detail management.

Reiners et al. [111] developed the OpenSG scene graph system. The OpenSG project shares many of our goals, and the system is similar in spirit to other scene graph systems such as Performer [127] and Jupiter [11]. Voß et al. [144] have recently added multi-threading and clustering support to OpenSG.

Varadhan and Manocha [143] described a system for out-of-core rendering that combined hierarchical LODs [46] and prefetching, but their system does not perform occlusion culling, and their preprocessing step is in-core.

DeCoro and Pajarola [39] developed XFastMesh, a system for interactive out-of-core rendering of large datasets The system supports view-dependent levels of detail, but does not support occlusion culling, and depends on an in-core preprocessing step.

Lindstrom [82] developed a system for out-of-core building and rendering of multiresolution surfaces. His system supports view-dependent levels of detail, but does not support occlusion culling.

In more recent work, Wald et al. [145] developed a parallel ray tracer capable of interactively rendering *dynamic* geometry, but they only report results for models smaller than main memory.

Yoon et al. [156] presented an in-core rendering system for high-end PCs that combines view-dependent level of details and occlusion culling.

The last row in Table 2.1 shows the features supported by our system, which we have named **iWalk**. No other system supports all the features that iWalk supports. On the other hand, iWalk does not support a few features supported by other systems. Among these systems, MMR is the only one that supports image impostors. Although image impostors may allow MMR to generate images with higher fidelity at the lowest levels of details, image impostors require long preprocessing times and a large amount of storage. Another feature not supported by iWalk is dynamic geometry. Only Chromium [68] and the recent in-core system of Wald et al. [145] support dynamic geometry. Finally, only Chromium is able to run unmodified applications.

To conclude this section, let us make it clear that the comparison in Table 2.1 is intentionally incomplete. We have ignored many factors that are not critical to us, but may be important in other contexts. These factors include:

- vendor support
- community support
- platform availability
- user interface
- documentation
- code maturity
- code license
- view-dependent LODs
- volume rendering
- photorealism level
- load balancing
- collision detection

2.5 Discussion

During the development stages of our system, we kept two main design goals in mind: we wanted all the steps, including preprocessing, to work on a PC with small memory, and we wanted to deliver interactive frame rates. Guided by these goals, for each stage of the pipeline we developed techniques that work out-of-core and that favor interactivity over image quality. Because of the huge difference in performance between main memory and disk, a major focus of the design of these techniques was trying to save memory for the geometry cache and avoid disk accesses. The combination of these techniques is a system that is simple, practical, scalable, and that strikes a good balance between interactivity and image quality. The system works around the weaknesses and exploits the strengths of current PC hardware.

The algorithms we chose to use for each step of the pipeline are appropriate for the specific task we are interested in, i.e., using commodity PCs to visualize datasets larger than main memory. These choices may not be appropriate for a system that can afford to keep the entire dataset in memory, or for a system whose goal is to generate photorealistic images. Similarly, techniques appropriate for those systems would not be the best for our goals and constraints.

For the spatialization data structure, we chose an octree. Although a regular grid would have been simpler, an octree allows us to perform hierarchical view-frustum culling. A hierarchy of boxes or a hierarchy of spheres would have been good choices as well, but our visibility algorithms assume that the leaves of the hierarchy form a spatial decomposition.

For the visibility algorithms, we chose PLP and cPLP. Because PLP is an approximate algorithm, it might produce objectionable artifacts if used alone by the rendering thread. Thus, we also use cPLP (implemented using the new OpenGL occlusion query extensions) in combination with LODs in the rendering thread. On the other hand, PLP is perfect for the prefetching thread. Because prefetching is speculative, an approximate visible set is good enough. In addition, because PLP does not need to access the disk or the graphics card, the prefetching thread runs without disturbing the other threads.

Another advantage of PLP and cPLP is that they are from-point algorithms. If memory is plentiful, from-region visibility may be a better alternative than from-point visibility. But if memory is at a premium, from-point visibility is more indicated, because it gives a smaller visible set, which in turn takes less space in the cache, and requires fewer disk accesses. In addition, from-point algorithms require less preprocessing time than from-region algorithms.

For the rendering primitives, we chose to use triangles, because they are the common denominator of higher order primitives, and current graphics cards are optimized to rasterize triangles. Another good choice would have been using points as primitives.

We chose *not* to use display lists. Rendering is fastest in current graphics cards if the geometry is stored in display lists, but displays lists take up a lot memory. A display list must make a copy of all data it requires to recreate the call sequence that created it. The OpenGL implementation also needs some extra memory to manage the display lists of a given context. If the dataset is small, this memory overhead may not be a problem. But if the dataset is large, display lists may actually hurt performance, because they could cause memory thrashing. [103]

We chose to use static LODs. Although continuous and view-dependent LODs produce smooth transitions between approximations, static LODs are better suited for today's graphics cards. Each static LOD can be stored and rendered as a vertex array, fully utilizing the potential of the graphics card. Continuous and view-dependent LODs tend be CPU-bound, and leave the graphics card under-utilized.

For the parallel extension of our system, we chose a simple sort first architecture, mainly because sort-first allows each renderer to run the entire graphics pipeline for the primitives in its tile. A sort-middle approach requires fast access to the intermediate results between the geometry processing and rasterization stages of the graphics pipeline, which current PC graphics cards do not provide. A sort-last approach would have prevented us from using occlusion culling based on image-space queries.

Early results of this work have been published elsewhere [26, 27, 28, 29, 30, 31]. In this dissertation we present new results and techniques. Since the publication of those papers, we have addressed some of the issues listed there as future work. In particular, we have added level-of-detail management and fast conservative occlusion culling to the system. In addition, we have tested our system using much larger datasets. Finally, we have updated the numbers for the experiments presented in those papers to reflect our current hardware.

The large number of recent publications on out-of-core rendering indicates that visualization of large datasets is far from being a solved problem. We hope to show that the techniques we present here are simple, yet useful and powerful, and contribute to the advancement of this field.

Chapter 3

Out-Of-Core Preprocessing

Recall that one of our goals is to visualize datasets that are larger than the main memory available in an inexpensive PC. Our approach is to keep the bulk of the dataset on disk, and load on demand from disk into a memory cache the visible parts of the dataset at the appropriate level of detail. Is this chapter, we describe the preprocessing algorithms that partition the model, compute coefficients that are used for visibility estimation, and create the levels of details for each part of the dataset.

3.1 Partitioning the Dataset Using an Octree

The first preprocessing step is to build an octree [119] that partitions the dataset into manageable pieces. A brute-force, in-core algorithm to build the octree would need a machine with large enough memory to hold the entire dataset. We avoid this brute-force approach, because we do not want to use a separate expensive machine with large memory just to build the octree. The out-of-core algorithm we present here builds the octree directly on a machine with small memory.

The algorithm first breaks the model in *sections* that fit in main memory, and then incrementally builds the octree on disk, one pass for each section, keeping in memory only the section being processed.

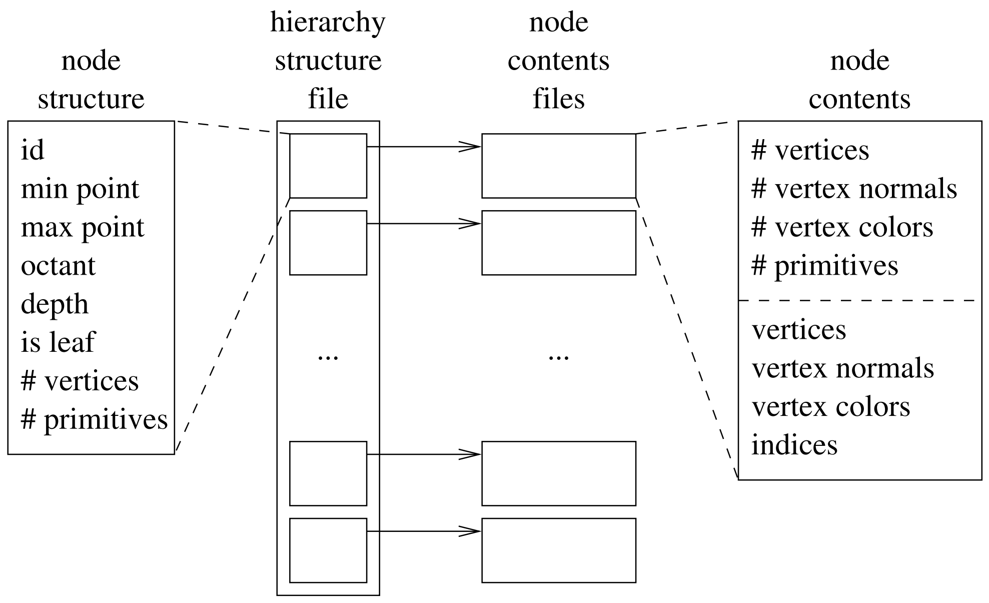
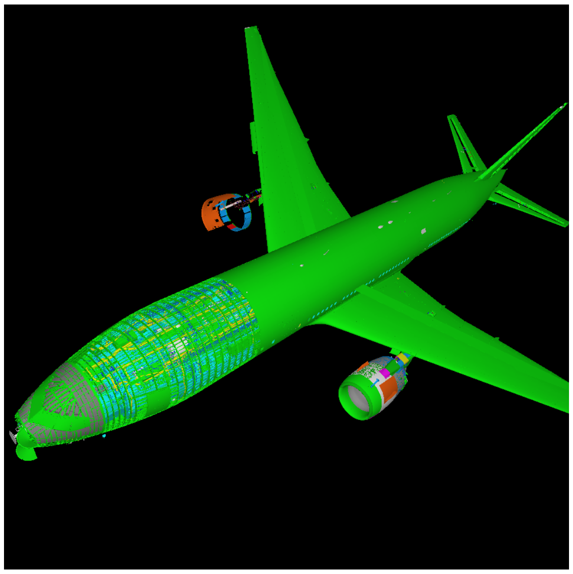


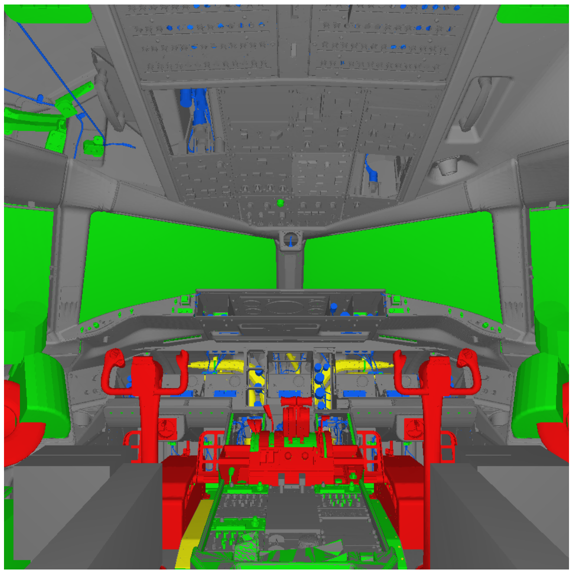
Figure 3.1: The layout of an octree on disk. The out-of-core spatialization algorithm builds an octree for a dataset, saving the skeleton of the octree in the hierarchy structure (HS) file, and the geometric contents of each node in a separate file.

To store the octree on disk, our algorithm saves the geometric contents of each octree node in a separate file, and creates a *hierarchy structure* (HS) file (Figure 3.1). The HS file has information about the spatial relationship of the nodes in the hierarchy, and for each node it contains the node's bounding box and auxiliary data used for visibility culling. The HS file is the main data structure that our rendering approach uses to control the flow of data. A key assumption we make is that the HS file fits in memory. That is usually a trivial assumption. For example, the size of the HS file for the Boeing 777 dataset (Figure 3.2) is only 1.2 MB.

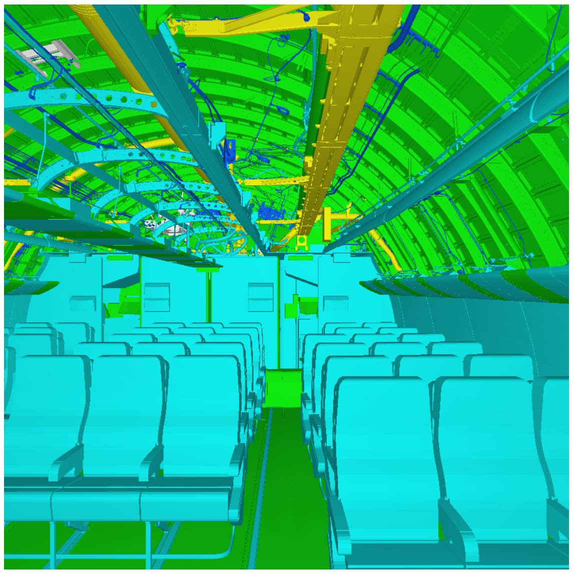
Figure 3.3 shows the high-level view of the out-of-core algorithm to build an octree for a given dataset given a maximum number of vertices per leaf. We begin by breaking the dataset into sections, which is very simple. Let N be the number of primitives in the dataset, and n the number of primitives that the machine can hold in memory (typically, N is much larger than n). We can create $\lceil N/n \rceil$ sections of at most n primitives each, without bringing the entire dataset into memory, by reading at most n primitives at a time, and writing them to a separate file. Chiang et al. [21]



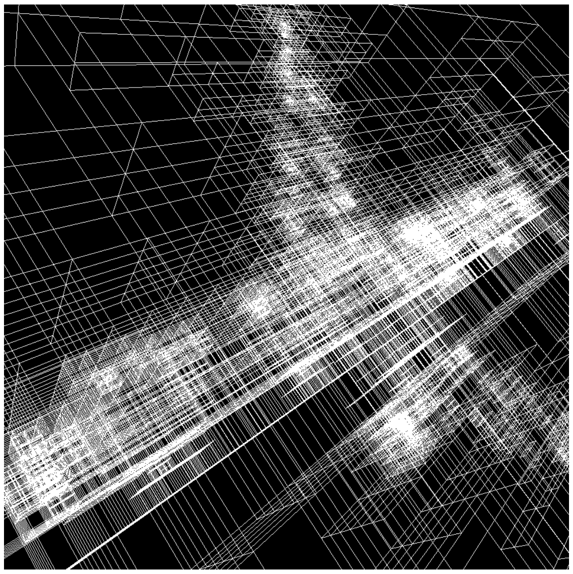
(a) exterior view



(b) interior view



(c) another interior view



(d) octree

Figure 3.2: The Boeing 777 dataset with 352 million triangles (7.5 GB of data). The size of the octree using at most 480,000 vertices per leaf is only 1.2 MB.

```
octree_build(dataset, max_vertices_per_leaf)
{
    break dataset in sections that fit in memory;
    compute dataset bounding box b;
    create empty octree with b and max_vertices_per_leaf;
    save octree structure;
    for (each section) {
        octree_insert_section(octree.root, section);
    }
}
```

Figure 3.3: Pseudocode for building an octree.

propose a technique that splits the dataset in spatially coherent sections. Many CAD models and datasets resulting from simulations already come as a set of small pieces, so this step may not be necessary.

In the next step, we create an empty octree using the bounding box of the dataset and the given maximum number of vertices per leaf. If the dataset was already given as a set of sections, we can compute the bounding box with a single pass over the dataset, bringing into memory one section at a time. Otherwise, we can compute the bounding box while breaking the dataset into sections.

Before proceeding, we save the structure of the octree on disk. This allows us to make the insertion of a section completely self-contained, and the whole process incremental. In particular, if we add new sections to the dataset in the future, we do not have to recompute the octree from scratch. The necessity for this incremental approach became evident when we were building octrees for models of real-world environments acquired by multiple passes of 3D scanning.

In the final high-level step, we insert the sections of the dataset into the octree one at a time. Figure 3.4 shows the pseudocode for inserting a section into an octree. We begin by loading the structure of the octree and the data for the section. Note that we do not load the data inside the previously existing octree nodes. We only load the structure file, which as we have mentioned, is very small.

```
octree_insert_section(octree, section)
{
    load octree structure;
    load section;
    for (each primitive in section) {
        octree_route_primitive(octree.root, primitive);
    }
    octree_save_data(octree);
    save octree structure;
    free section;
}
```

Figure 3.4: Pseudocode for inserting a section into an octree.

For each primitive of the section, we route the primitive, i.e., we find the octree leaf that should store the primitive. Figure 3.5 shows the pseudocode for routing a primitive. We recursively search for the leaf that intersects the primitive. If the primitive intersects multiple leaves, we replicate the primitive in all intersected leaves. When we reach a leaf, we check if it is full, i.e., if the number of vertices in the leaf has reached the specified maximum. If the leaf is not full, we insert the primitive there. Otherwise, we create eight children nodes for the leaf, making it an internal node, and redistribute its data among its children.

Finally, we save the data files of the octree nodes affected by the insertions. Figure 3.6 shows the pseudocode for saving the octree data. For each octree node that used to be a leaf before the insertion of the current section, we perform the following steps. If the node is still a leaf, we merge the new data with the old data (if any). If the result of the merge exceeds the allowed maximum number of vertices per node, we redistribute the data, which will make the leaf into an internal node. Then, we write the data files of the current subtree. If the node used to be a leaf and now is an internal node, we check if the node used to have data. If it did, we merge all the new data of the current subtree with the old data, redistribute the data, and write the data files. If the node used to be empty, we just write the data files for the new data in the current subtree.

```
octree_route_primitive(node, primitive)
{
    if (node is leaf) {
        if (node is not full) {
            insert primitive into node;
        } else {
            create eight children for node;
            distribute data among children;
        }
    } else {
        for (each child) {
            if (primitive intersects child) {
                octree_route_primitive(child, primitive)
            }
        }
    }
}
```

Figure 3.5: Pseudocode for routing a primitive.

The final leaves may have different numbers of primitives and volumes, but each leaf will contain at most the predefined number of vertices. The important point is that all insertions are local to a leaf, and therefore never require reading from disk more than one octree node of a fixed maximum size.

If we are building a dataset incrementally, a new section may not fit inside the bounding box of the original dataset. In this case, to avoid rebuilding the octree for the entire dataset, we grow the octree toward the new section. We create seven siblings for the current root node, and a new root that will be the parent of the old root and its new siblings. We repeat this until the octree does contain the new section, and then proceed with the insertion as before.

The final number of files corresponding to the leaves of the octree may be large (e.g., tens of thousands). If we save all the files in the same directory, opening a file might involve a linear search on the file name. To avoid this problem, we save the octree leaves in a hierarchy of directories, where each directory stores at most a certain number of files (typically 25).

```
octree_save_data(octree)
{
    for (each node that used to be a leaf) {
        if (node is leaf) {
            if (node has new data) {
                if (node had old data) {
                    read old data;
                    merge with new data;
                    free old data;
                    remove old data file;
                }
                if (new data is too big) {
                    split node;
                    redistribute data;
                }
                write data files in this subtree;
            }
        } else {
            if (node had old data) {
                merge new data of this subtree;
                read old data;
                merge with new data;
                free old data;
                remove old data file;
                redistribute data;
                write data files in this subtree;
            } else {
                write data files in this subtree;
            }
        }
    }
}
```

Figure 3.6: Pseudocode for saving the octree data.

Our spatialization algorithm has three important features:

- It is an out-of-core algorithm. When adding a section, we only need memory for the section itself, the hierarchy structure file, and the contents of one leaf. The section fits in memory by construction, the size of HS file is negligible, and the size of the contents of a leaf is limited by the maximum number of vertices per leaf. Thus, we can create octrees for extremely large data.
- It is an incremental algorithm. If new objects are added to the dataset, only the spatial regions touched by those objects need to be updated, as opposed to rebuilding the entire hierarchy. This is particularly useful for applications that build models incrementally, such as 3D scanning.
- It is fast. For each section, the algorithm only reads a modified node once, doing the insertion in the most efficient way.

Some researchers have developed similar algorithms. Ueng et al. [141] presented an out-of-core algorithm to build an on-disk octree for large unstructured tetrahedral meshes. Both their algorithm and ours save the structure (or skeleton) of the octree in a file, and the contents of the octree nodes in separate files. Also, both algorithms enforce a maximum amount of data per octree node. The main difference is that, when adding a new section to an existing octree, their algorithm may need to read the same node multiple times, while our algorithm only needs to read an affected node at most once per section.

Cignoni et al. [22] developed an out-of-core algorithm for simplification of large datasets. Their algorithm first builds a raw (not indexed) octree-based external memory mesh (OEMM), and then traverses the raw OEMM twice to build an indexed OEMM. Our preprocessing algorithm is similar to the first phase of their simplification algorithm. The main difference is that they build the octree starting from the leaves at a predefined depth, and then merge adjacent leaves with few primitives. We build the octree starting from the root, and then split leaves with too many primitives. We expect our algorithm and theirs to have similar running times.

Many other researchers have developed spatialization algorithms with the same goal, but different implementations. The algorithm of Wald et al. [146] creates a BSP tree for the dataset. Pharr et al. [105] and Prince [107] use a regular grid.

McMains et al. [90] have developed an out-of-core technique to build a topological data structure for a large dataset of unordered polygons. Their data structure supports much more functionality than we need. The extra connectivity information is not useful to us. We are only interested in interactive rendering. Using a simple octree allows us to have very fast preprocessing times.

3.2 Computing Visibility Coefficients

The next preprocessing step is computing visibility coefficients for each octree leaf. As we will see in more detail in Chapter 4, these visibility coefficients are used at runtime by the prioritized-layered projection (PLP) [73] algorithm to estimate the octree nodes that are visible from the current viewpoint. The basic idea is to compute a value that estimates how likely it is for a node to block the light passing through it. In their original paper, Klosowski and Silva estimated this likelihood based on the number of primitives in the leaf.

We improve upon Klosowski and Silva's approach by precomputing a set of view-dependent values based on the screen coverage of the primitives in a node relative to the screen coverage of the node's bounding box. For each octree leaf, we place an arbitrary number of sample viewpoints around the octree leaf. We pick each viewpoint so that when we look from the viewpoint towards the center of the node we are able to see the entire node, and we maximize the projected screen area of the node's bounding box.

For each sample viewpoint, we rasterize the node's bounding box in green and then the node's contents in red over a black background, without depth tests. We then read back the frame buffer, and count the number of green and red pixels, n_g and n_r. We approximate the probability of the node blocking light from this viewing direction by the ratio of red pixels to lit pixels, $n_r/(n_r + n_g)$.

We typically store 20 of such coefficients per node. At runtime, we pick the coefficient whose corresponding sampling direction is closest to the current viewing direction. These coefficients are fast to precompute, cheap to store, and give us a more accurate estimate of visibility at runtime than other simpler statistics such as the number of primitives in the node.

3.3 Creating Levels of Detail

The final preprocessing step is precomputing levels of detail. For each octree node, we compute a small set of static levels of detail. We use the vertex clustering algorithm of Rossignac and Borrel [112]. Typically we precompute 3 to 5 successive approximations of the data in each octree node. Each approximation has roughly 1/4 of data of the previous approximation. At runtime, the appropriate level of detail is selected based on the expected contribution of the octree node to the quality of the image.

Rossignac and Borrel [112] use two factors to grade a vertex: the length of the longest edge incident to the vertex, and the maximum angle between the edges incident to the vertex. Unfortunately, these factors may be misleading if the dataset has been triangulated already [87]. Since most of our test datasets were already triangulated when we obtained them, we searched for a different metric to grade the vertices. We found that we obtained better-looking simplified versions of the dataset by ignoring the angle between the edges, and weighting a vertex by the maximum area of the faces incident to the vertex.

There are many reasons why we chose to use simple vertex clustering instead of more sophisticated simplification approaches:

- Vertex clustering is very simple and very robust. In particular, it makes no assumptions about the original geometry.

- Vertex clustering is very fast. Simplifying a node after it is in memory is faster than reading the node from disk.

- Vertex clustering only needs to traverse the data once, which is important for us, because we are I/O bound.

- The quality of the approximations produced by vertex clustering is good enough for an interactive previewer such as we want, and not much worse than that of more complicated, slower methods.

- The error introduced by the simplification is bounded (in the Hausdorff distance sense) by an intuitive, user-controlled accuracy dial.

- Vertex clustering does not require the construction of a topological adjacency graph between faces, edges, and vertices.

- Vertex clustering produces static LODs, which are better suited for current hardware than dynamic LODs.

Originally, we used to employ Popinet's implementation [106] of Lindstrom's algorithm [81, 84] to compute LODs. But the computation of adjacency information was proving to be too time consuming for very large datasets. When we started experimenting with a 473-million triangle dataset, we realized we had to sacrifice fidelity to achieve practical preprocessing times.

Ho et al. [60] also use vertex clustering in their mesh compression system. To compress meshes larger than main memory, Ho et al. advocate automatically partitioning the mesh into sub-meshes that fit in memory, and compress them separately, ignoring intersections between neighboring regions of the partition. Ho et al. point out that this approach is advantageous, because it is simple and allows them to leverage existing in-core simplification techniques for each region. Hoppe [63] and Bernardini et al. [13] also partition the input dataset into pieces small enough to fit in memory, and then simplify them individually.

Isenburg and Gumhold [69] have developed an out-of-core compression technique that converts massive meshes into a streamable representation. Their focus is on one-pass decoders that allow for streaming decompression that can start producing mesh triangles as soon as the first few bytes become available. They are more interested in compression ratio than frame rates. For example, the execution time of their rendering system was bound by the computation of triangle normals, which they could have precomputed, but they chose not to.

Isenburg and Gumhold [69] point out that they dislike the approach of simplifying pieces of the dataset separately, because of the discontinuities that may be introduced between regions. We do not mind the discontinuities. In practice they are not too disturbing, and they can be easily fixed, if necessary [63].

3.4 Experimental Results

In this section we report on the performance of our preprocessing algorithms. One of our goals was to evaluate the time necessary to preprocess a dataset. For the system to be practical, the preprocessing step needs to be automatic and reasonably fast. A few minutes or even a couple of hours may be acceptable, but days would be too long.

Another goal of these experiments was to study the tradeoff between the granularity of the spatialization, i.e., the choice of the maximum number v_{max} of vertices per leaf, and the size of resulting octree. Finer granularity (small v_{max}) allows for more precise view-frustum and occlusion culling, potentially reducing the load on the graphics card. But small granularity also increases the chance of primitive replication, and increases the traversal load on the CPU. Coarser granularity (large v_{max}) reduces the traversal time, and decreases the chances of replication, but increases the chances of fetching and rendering invisible geometry. Choosing the right granularity can affect the running time by a factor greater than ten [125].

Yet another goal of these experiments was to assess the quality of the levels of detail produced by vertex clustering. A common criticism towards vertex clustering is that it may produce poor approximations of the dataset. We will se below that the quality of the approximations is good enough for an interactive previewer. If the need for better approximations arises, we can use any other simplification algorithm, because the simplification step is orthogonal to the rest of the system.

We ran experiments with two datasets. The first dataset is the UNC power plant model [147], which contains 13 million triangles (Figures 3.7–3.9). This is a challenging model, because of its high depth complexity, which calls for occlusion culling. View-frustum culling, even if combined with LOD management, would render many invisible triangles unnecessarily. Another reason why we ran tests with the power plant model is that many previous systems have used it, which allows us to make more objective comparisons.

The second dataset is the Lawrence Livermore National Laboratory (LLNL) isosurface dataset [97], which contains 473 million triangles (Figure 3.10). This is a truly massive dataset whose original size is 8.3 GB. After converting the dataset to our own format, the size went up to 9.8 GB, mainly because the original dataset did not have vertex colors. We assigned to each vertex a color that indicates its height.

We ran the preprocessing tests on a 2.4 GHz Pentium IV computer with 512 MB of RAM and a 250 GB IDE disk. The computer was equipped with a NVIDIA GeForce Quadro FX 500 graphics card. The computer's operating system was Red Hat Linux 8.0. The total cost of this machine is about US$1,000.

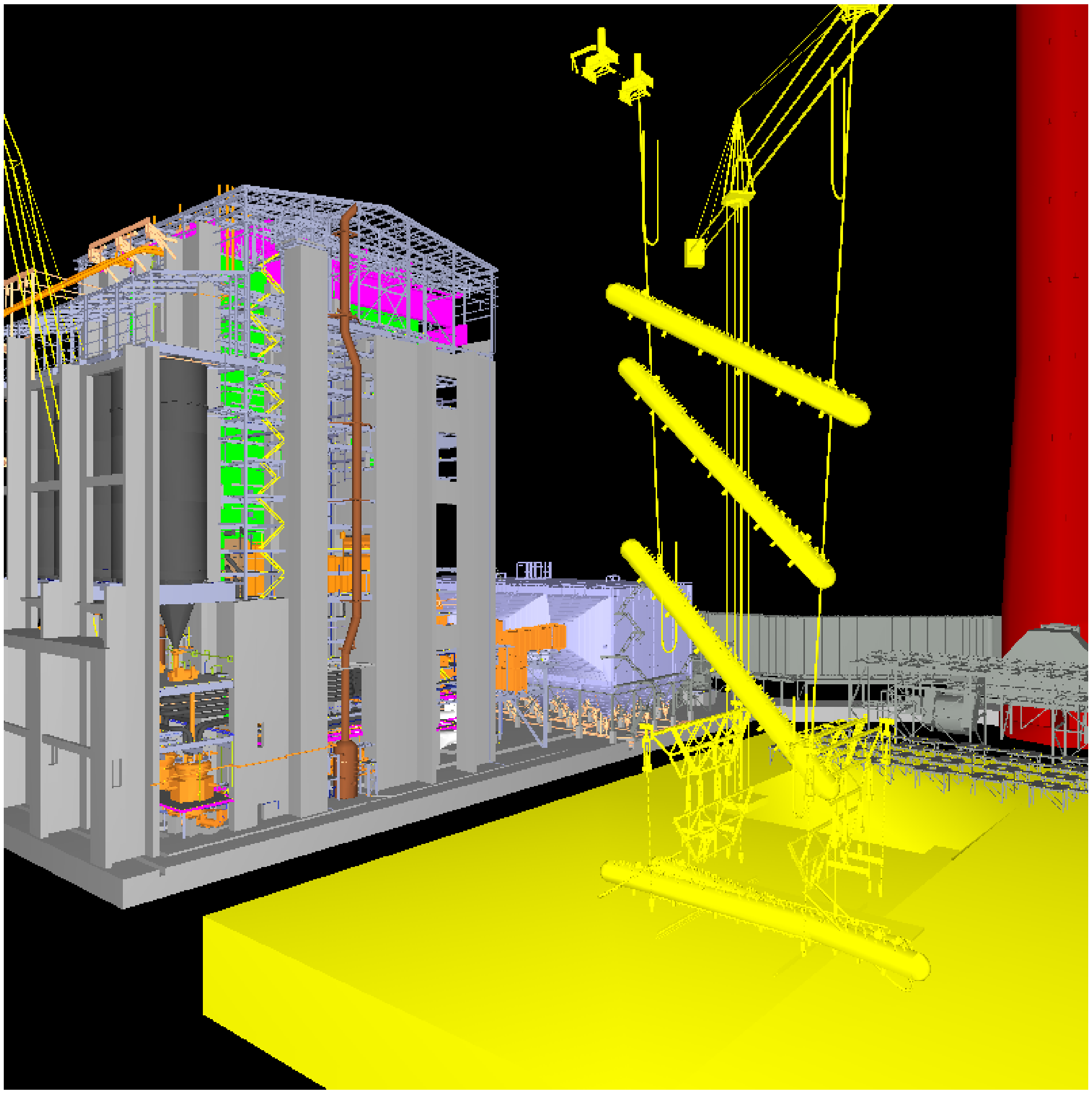


Figure 3.7: An exterior view of the UNC power plant [147] with 13 million triangles.

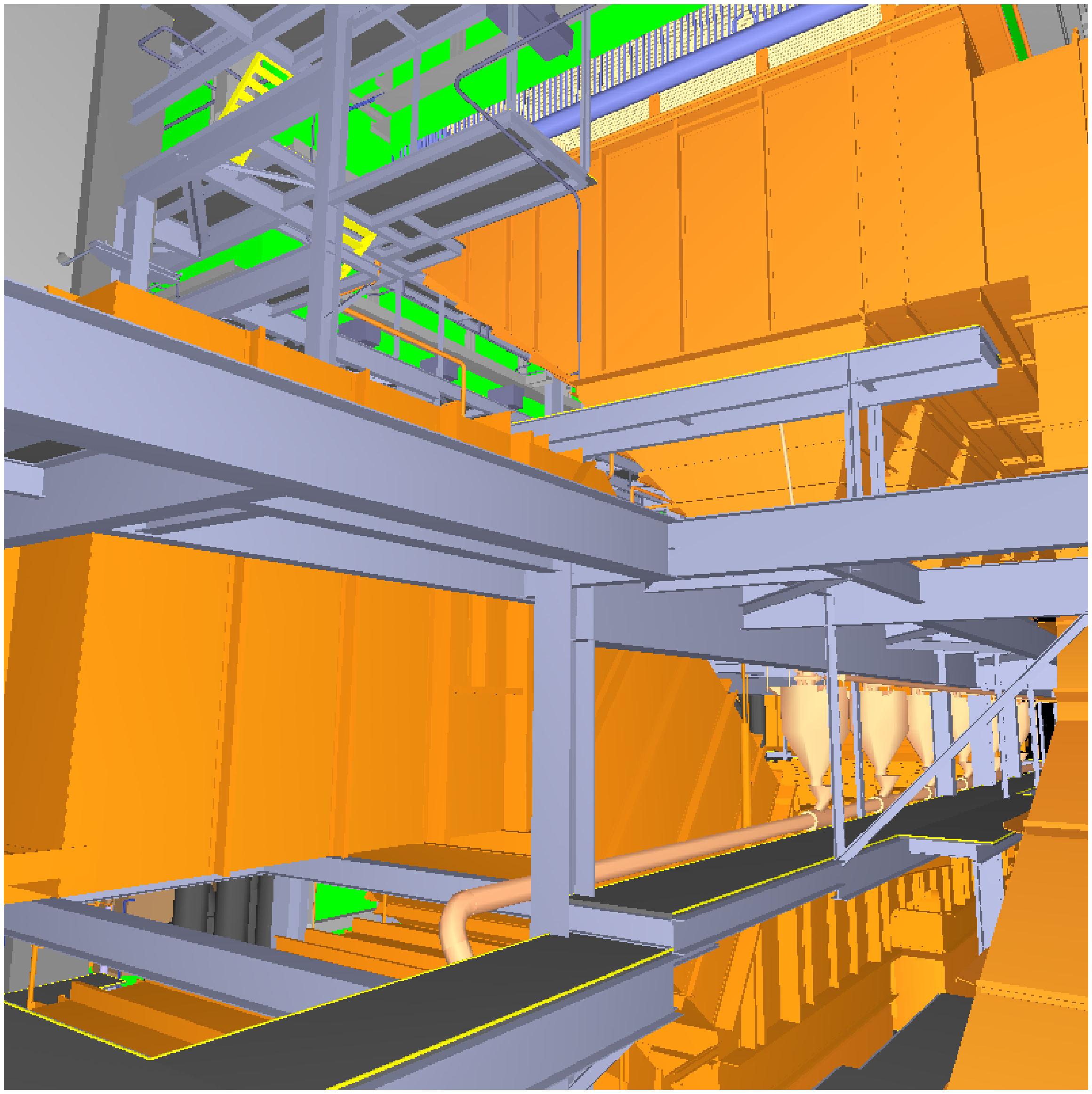


Figure 3.8: An interior view of the UNC power plant model [147].

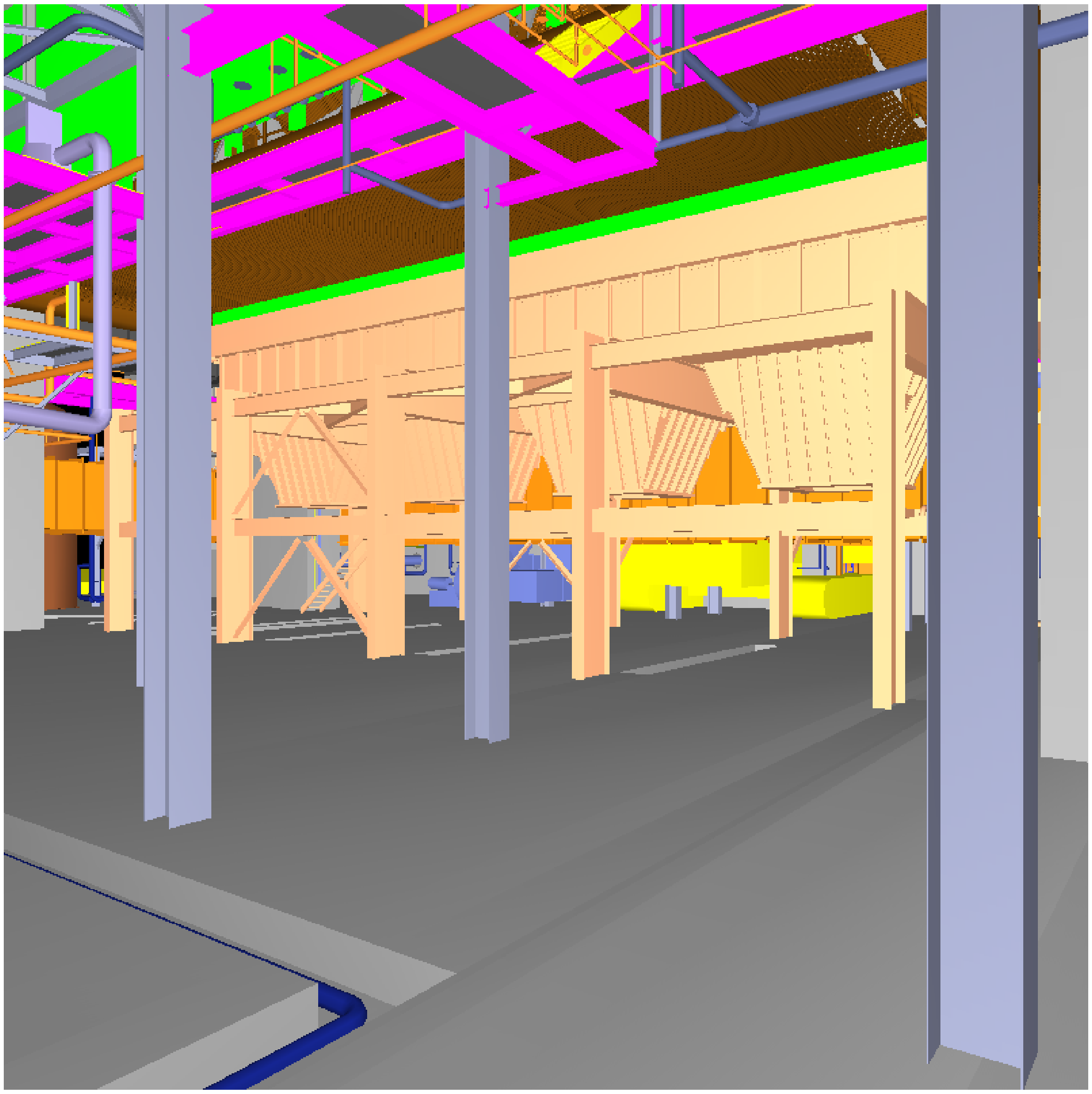


Figure 3.9: Another interior view of the UNC power plant model [147].

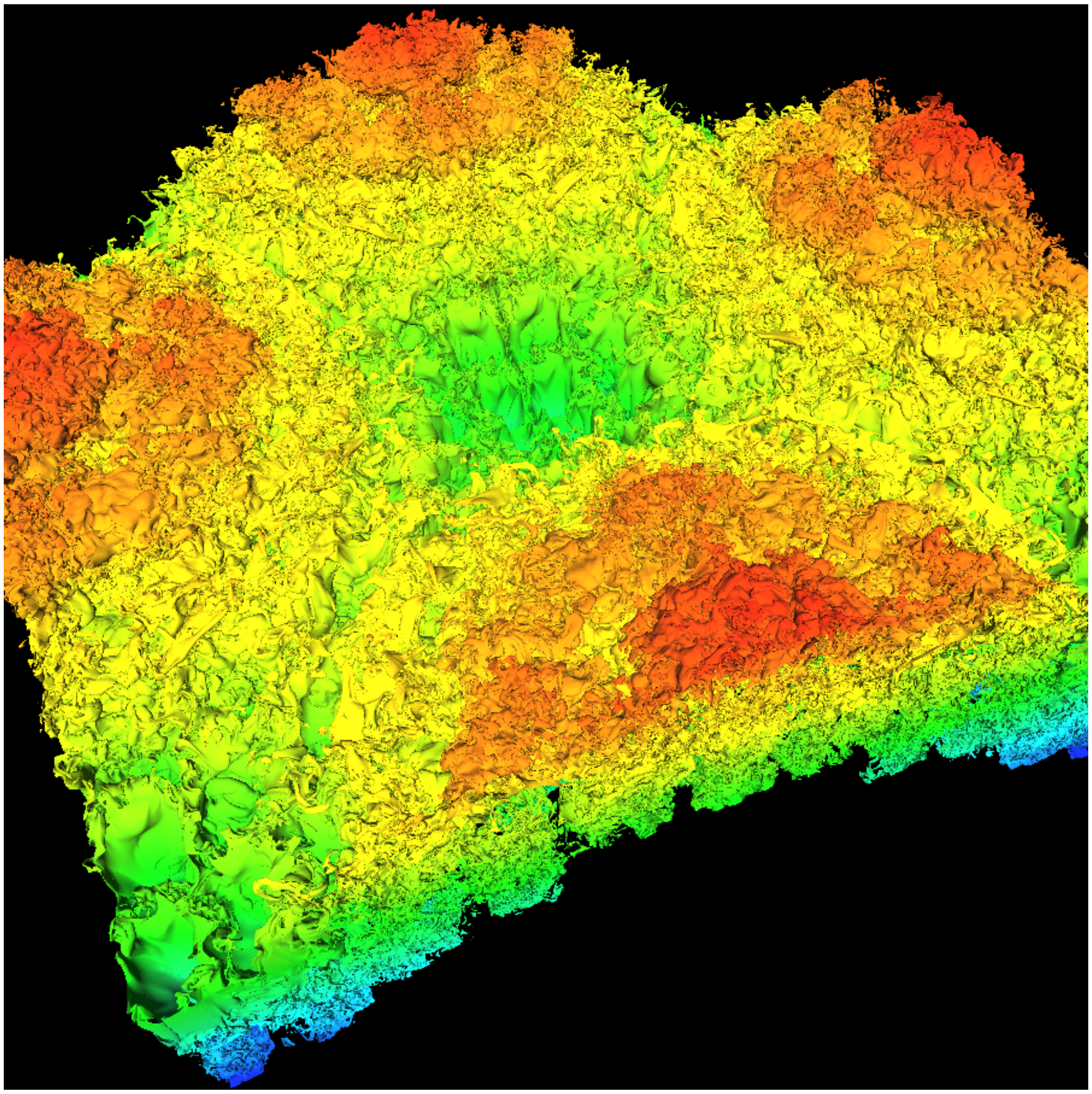


Figure 3.10: The LLNL isosurface dataset [97] with 473 million triangles.

Max vert/leaf	Build time	Size (MB)	Depth	Leaves	Nodes	Triangles
3750	10m 03s	1052	11	72,416	82,761	30,461,154
7500	7m 51s	833	11	33,944	38,793	25,985,206
15000	6m 24s	671	10	15,177	17,345	22,073,219
30000	5m 17s	578	9	6,847	7,825	20,088,458
60000	4m 45s	510	9	3,354	3,833	18,301,106
120000	4m 16s	465	8	1,744	1,993	17,509,750
240000	3m 57s	426	8	701	801	16,215,938

Table 3.1: Building the octree for the power plant model.

3.4.1 UNC Power Plant Results

Building the Octree

The power plant model consists of 21 sections, each of which fits in the main memory of the test machine. We used our out-of-core incremental spatialization algorithm to build the octree for the entire model, one section at a time. Table 3.1 shows the results for the construction of the octree for the power plant model. We varied the maximum number of vertices per leaf from 3,750 to 240,000. The finer the granularity, the longer it took to build the octree. The running time is in the order of minutes, and it is dominated by disk reads and writes. Other researchers report much longer running times to spatialize this model [5, 146].

Because of triangle replication, the finer the granularity, the larger the size of the octree. Figure 3.11 shows a chart with the total octree size plotted versus the maximum number of vertices per leaf. Based on this chart, and runtime trial and error, we chose 15,000 vertices per leaf for the rest of experiments with this model.

Figure 3.12 shows the power plant model from another angle with the structure of the octree superimposed. The octree shown is the one created using 120,000 as the maximum number of vertices per leaf. Note that the grid is irregular, i.e., some nodes are larger (in volume) than others, which reflects the different density of triangles per volume of different regions of the model.

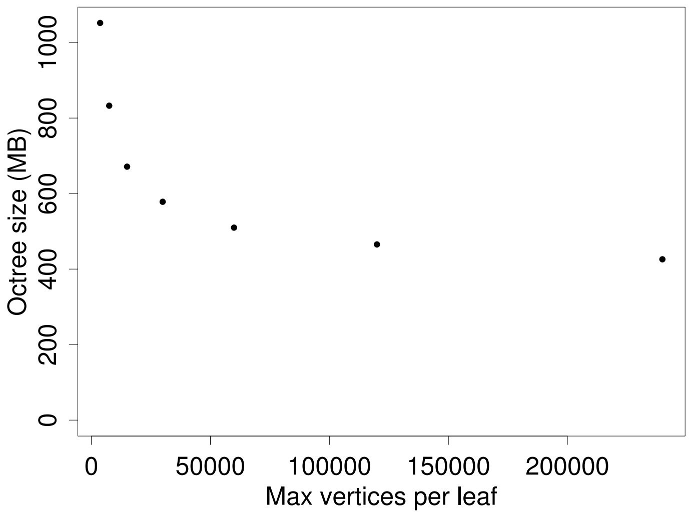


Figure 3.11: Octree size versus maximum number of vertices per leaf.

Computing Visibility Coefficients

We rasterized each octree node on a 64×64-pixel window, and used 20 sample viewpoints. The total running time to compute the visibility coefficients was 2 minutes and 36 seconds. The total size of the visibility coefficients was 711 KB. Thus, both the time and storage requirements for the visibility coefficients are negligible.

Creating Levels of Detail

We created at most 5 levels of detail for each octree node: the original data plus 4 approximations starting with a grid of 128 voxels per axis. Each approximation had roughly 1/4 of the data of the preceding level of detail. The total running time to create the level of details was 8 minutes and 5 seconds. The total size of the additional data was 268 MB. Figure 3.13 shows closeup views of several levels of detail of the powerplant model. Vertex clustering does a good job at preserving the overall shape of the model, even for very low polygonal counts, especially if we consider how simple and fast the algorithm is. Figure 3.14 shows those same levels of detail using regular views, i.e., from the distance that they would be seen at runtime. In these views, the artifacts created by vertex clustering are much less noticeable.

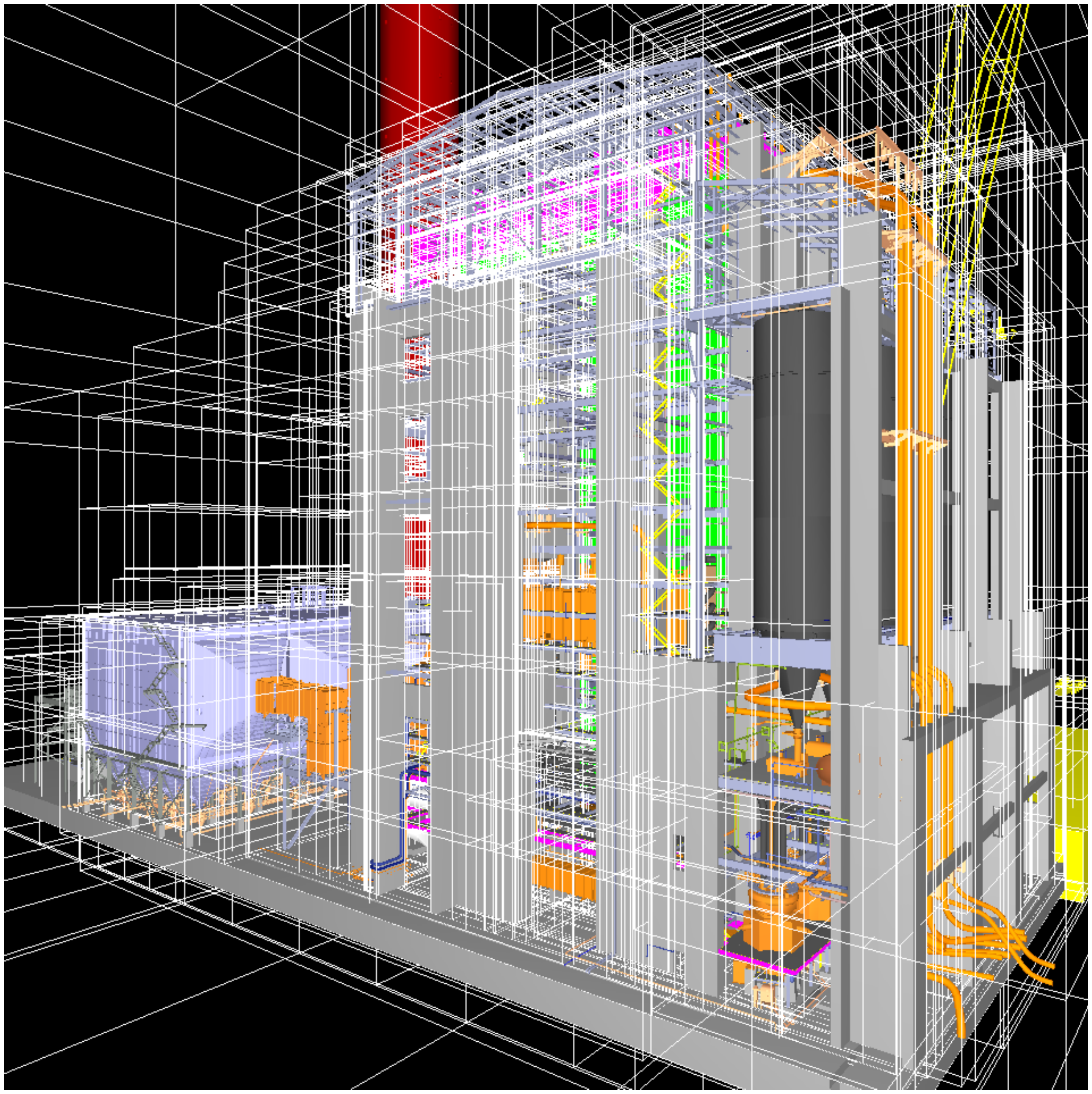
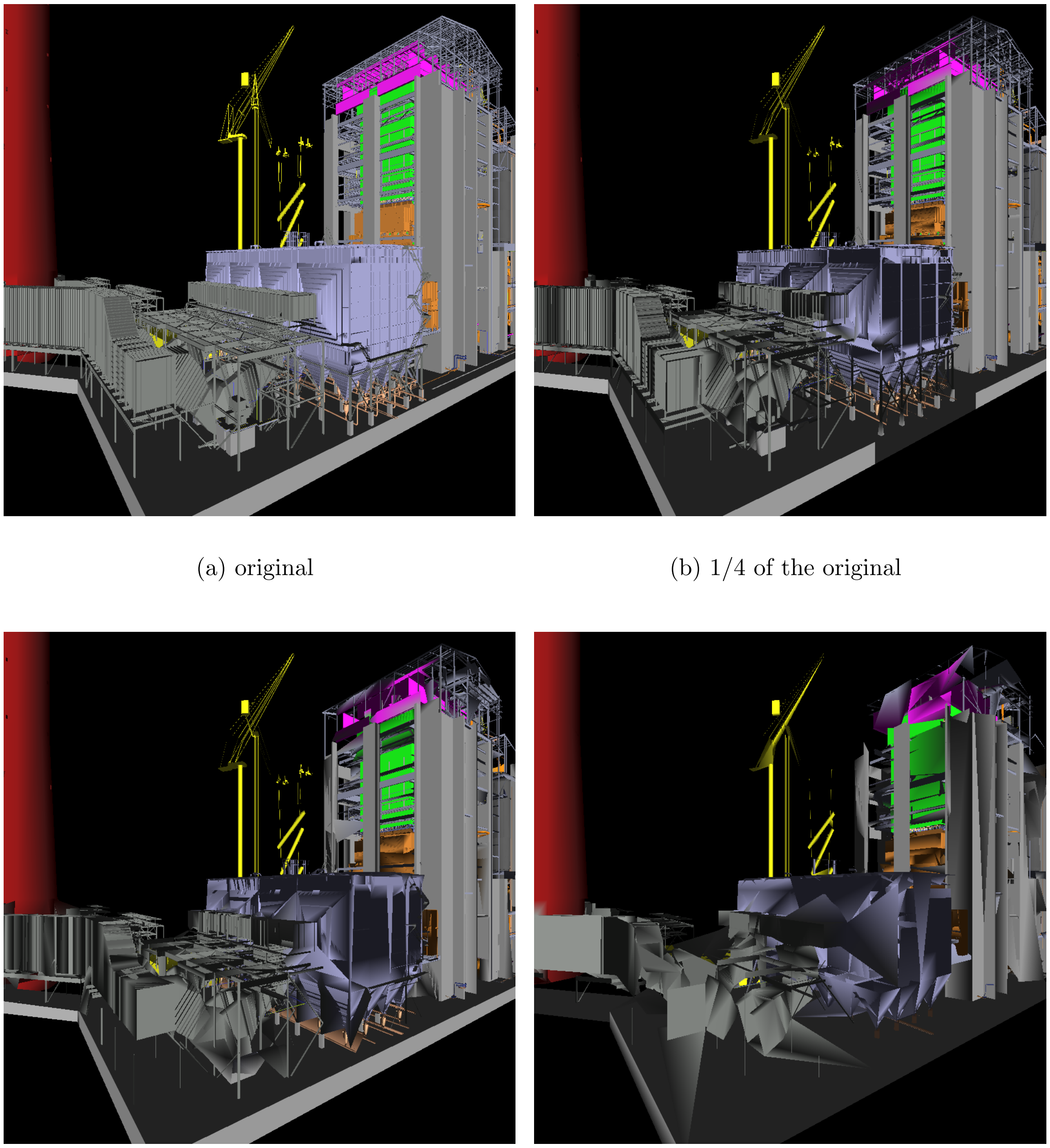


Figure 3.12: The UNC power plant model [147] with the octree superimposed.

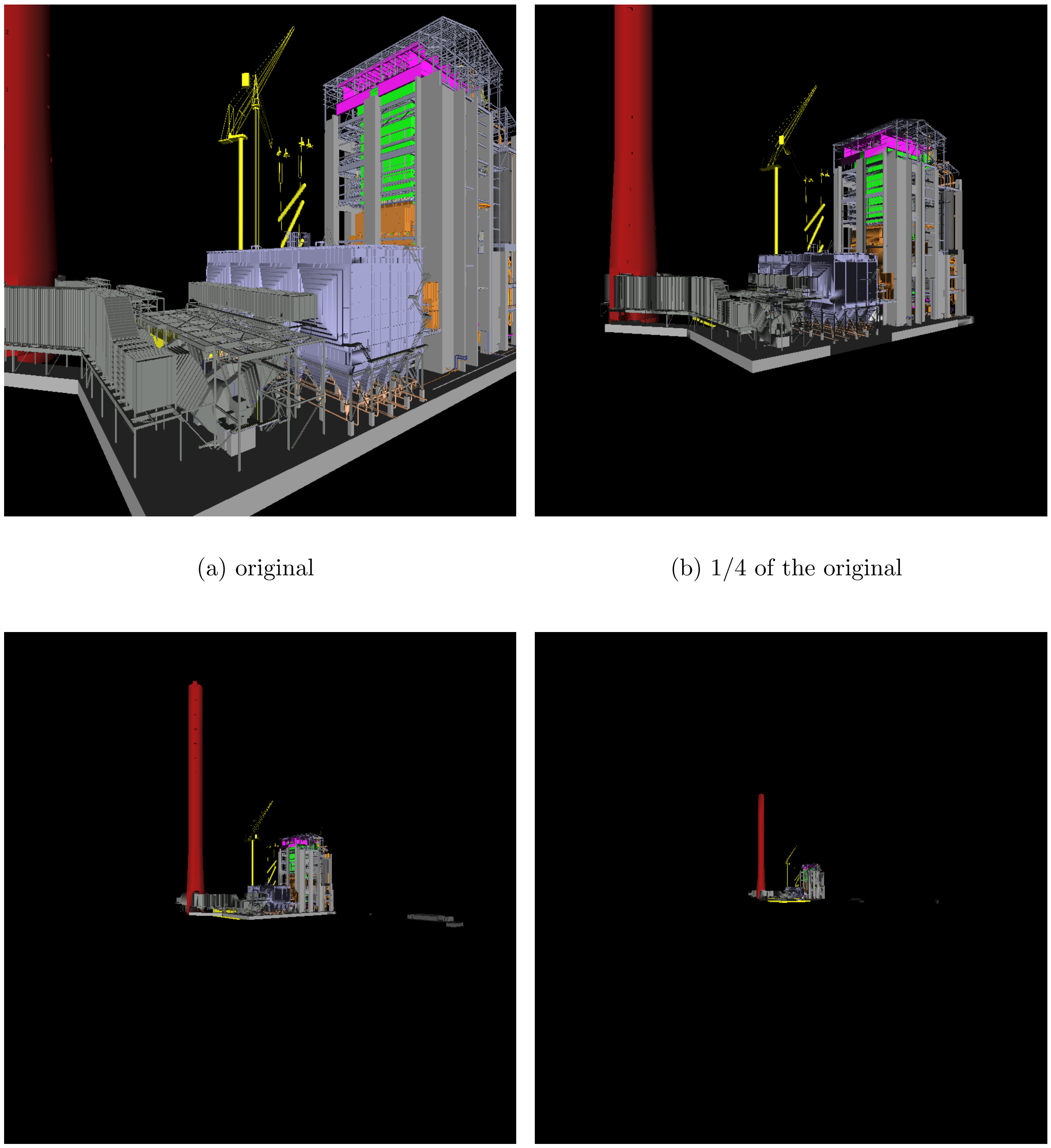

(a) original

(b) 1/4 of the original

(c) 1/16 of the original

(d) 1/64 of the original

Figure 3.13: Closeup view of several levels of detail of the power plant.


(a) original

(b) 1/4 of the original

(c) 1/16 of the original

(d) 1/64 of the original

Figure 3.14: Regular view of several levels of detail of the powerplant.

3.4.2 LLNL Isosurface Results

Building the Octree

To build the octree for the LLNL isosurface, we used the limit of 480,000 vertices per leaf. The original dataset has 473 million triangles, and its size (after adding colors) is 9.8 GB. The resulting octree has 561 million triangles (because of replication of triangles that intersect multiple nodes), and its size is 10 GB. The octree has 7,393 nodes, 6,469 leaves, and the maximum depth is 5. The construction of the octree took 1 hour and 24 minutes. The size of the hierarchy structure file for this octree is 1.3 MB. Figure 3.15 shows a screenshot of the structure of this massive octree.

Computing Visibility Coefficients

To compute the visibility coefficients for the LLNL isosurface, we used the same approach we used for the power plant model. The total running time to compute the visibility coefficients was 25 minutes and 46 seconds. The total size of the visibility coefficients was 303 KB.

Creating Levels of Detail

To create the levels of detail for the LLNL isosurface, we created at most 4 approximations of the original data starting from a grid of 128 voxels per axis. Each approximation had at most 1/4 of the data of the previous approximation. The total running time to create the levels of detail was 1 hour and 16 minutes, and the total size of the approximations was 2.3 GB.

Figure 3.16 shows closeup views of several levels of detail of the LLNL isosurface dataset. Figure 3.17 shows those same levels of detail using regular views. Once again, the quality of the approximations produced by vertex clustering seems good enough for our purposes.

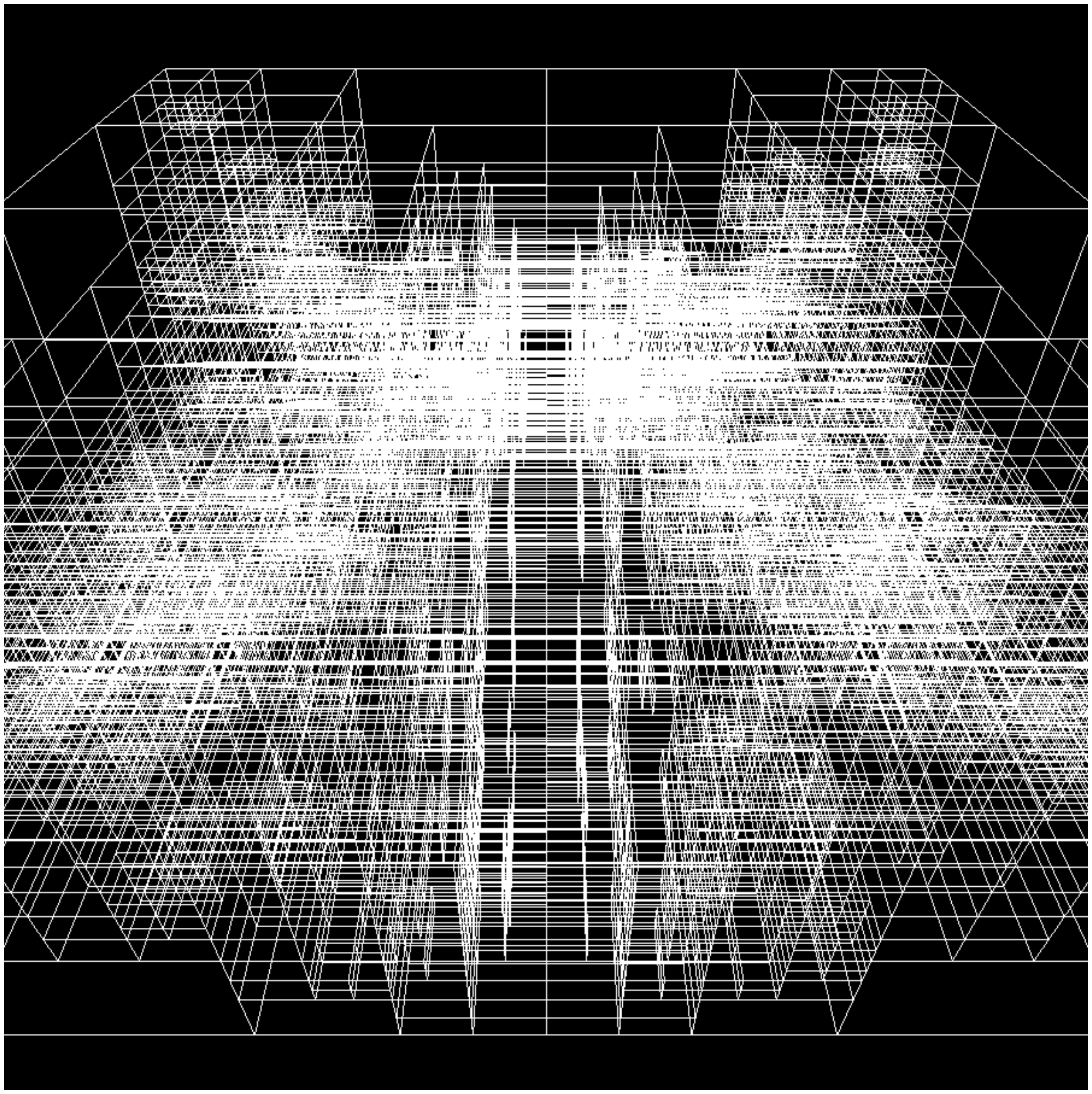
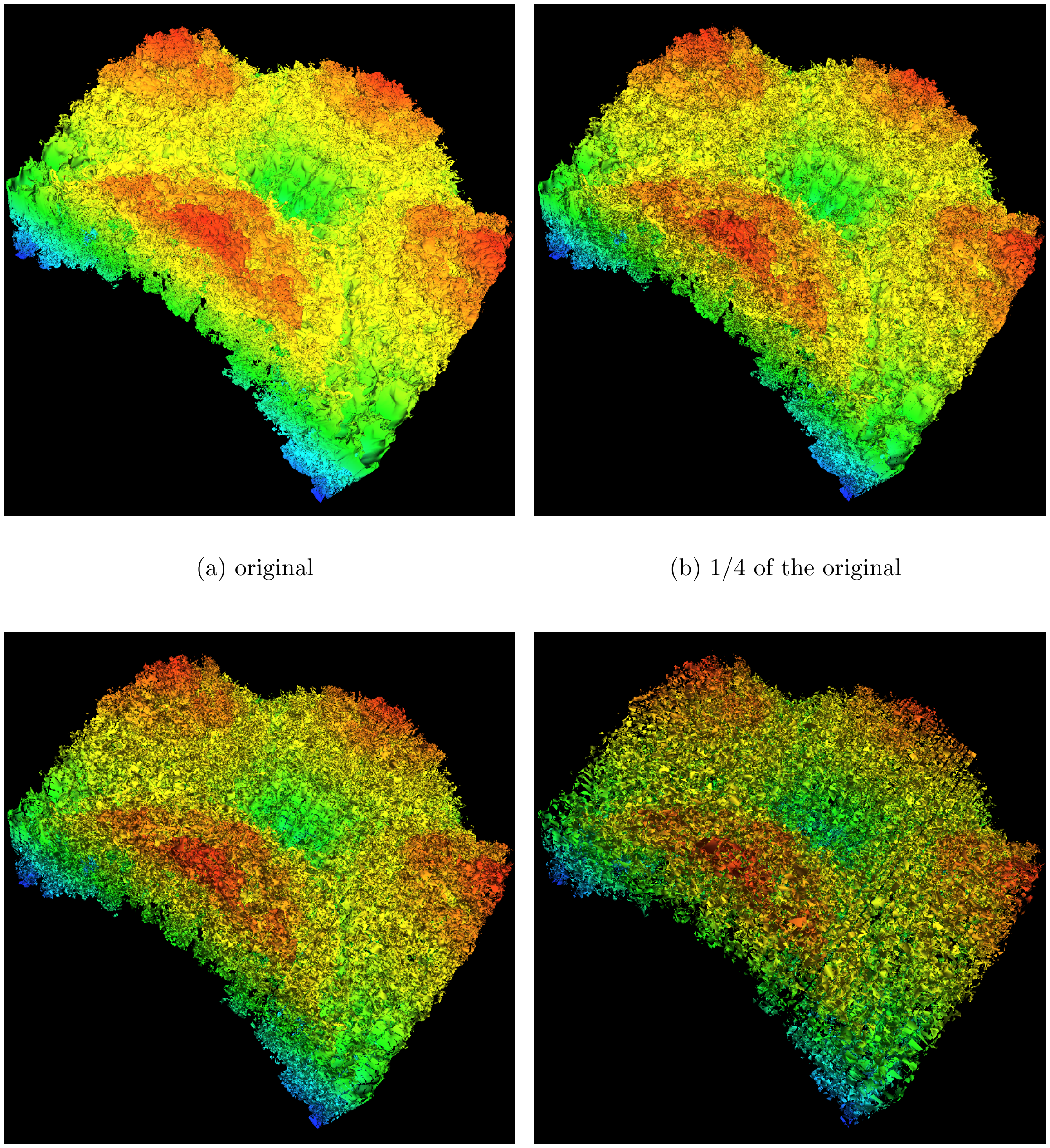


Figure 3.15: The structure of the octree for the LLNL isosurface.



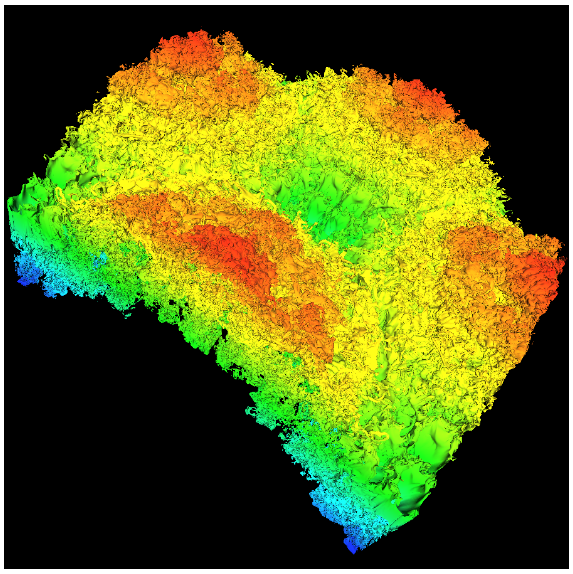
(a) original

(b) 1/4 of the original

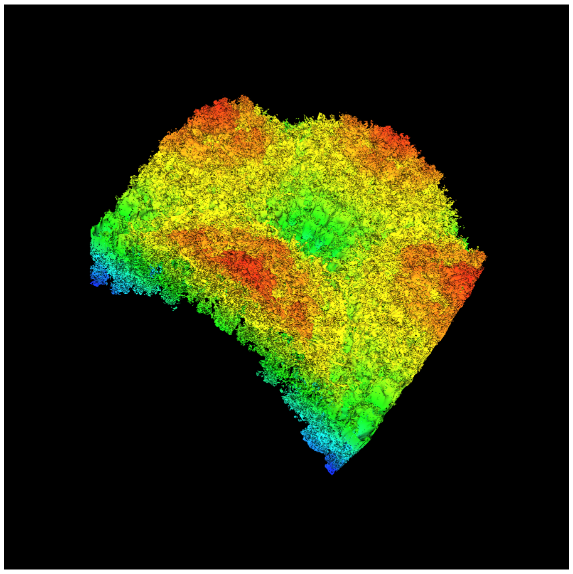
(c) 1/16 of the original

(d) 1/64 of the original

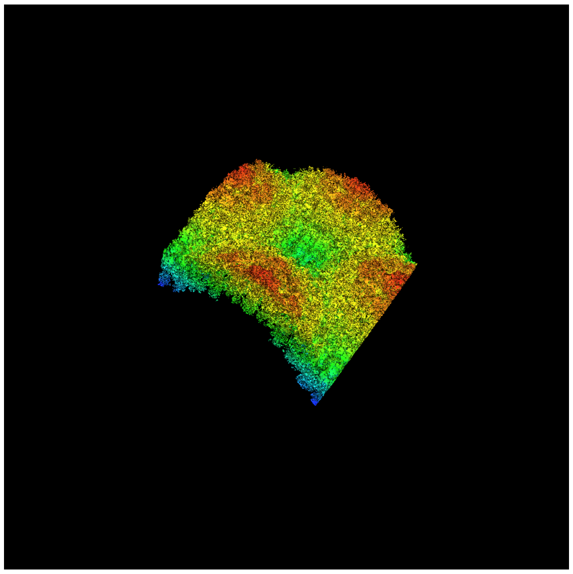
Figure 3.16: Closeup view of several levels of detail of the LLNL isosurface.



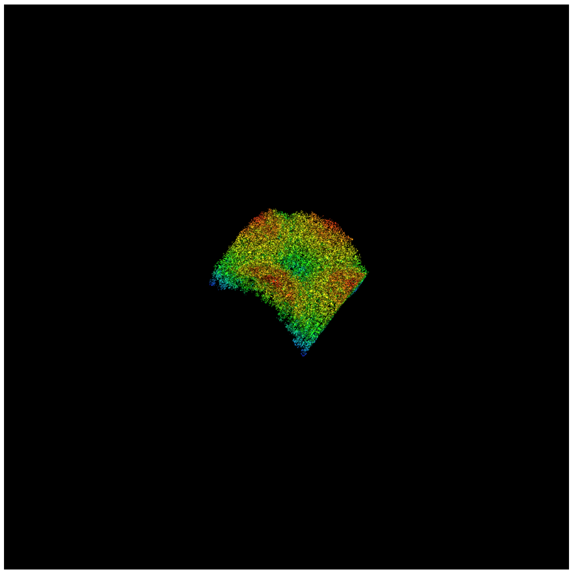
(a) original



(b) 1/4 of the original



(c) 1/16 of the original



(d) 1/64 of the original

Figure 3.17: Regular view of several levels of detail of the LLNL isosurface.

3.4.3 Summary of the Preprocessing Results

Our system was able to spatialize the UNC power plant model in about 6 minutes. The system was also able to spatialize the LLNL isosurface dataset, which is 20 times larger than the main memory of our test machine, in about 1.5 hours, which is fast enough to be practical. The computation time and storage requirements for the visibility coefficients were negligible. The computation time and storage requirements for the levels of detail were low, and the quality of the approximations produced by vertex clustering was good enough for an interactive previewer.

The best numbers we know for automatic, out-of-core spatialization of the power plant model are from Wald et al. [146]: roughly 30 minutes. The actual number they published in 2001 is 2.5 hours, but we are estimating that our test machine is roughly 5 times faster than the one they used then. Thus, keeping in mind that our data structures are different (we use an octree, and they use a BSP tree), it is perhaps fair to say that our spatialization algorithm is 5 times faster than theirs.

For the LLNL isosurface, we are unaware of any out-of-core preprocessing numbers on low-end PCs. Lindstrom [82] reports 2 hours and 40 minutes for an isosurface about half the size of the one we used. We cannot make a direct comparison, however, because his algorithm is more sophisticated (it creates view-dependent LODs, as opposed to static LODs), and his test machine was a high-end SGI Onyx2.

Chapter 4

Out-Of-Core Rendering

This chapter presents our approach to render datasets larger than main memory. So far we have shown how to break the dataset into manageable pieces, and how to precompute visibility information and levels of detail for each piece. Now we show how to render the pieces at runtime. We start with an overview or the rendering approach, followed by a review of the PLP and cPLP visibility algorithms. Then, we describe our extensions to PLP and cPLP, and present our cache management techniques, including our novel prefetching algorithm.

4.1 Overview of the Rendering Approach

We named our rendering system iWalk. Figure 4.1 shows a diagram of iWalk's rendering approach. The user interface (a) keeps track of the position, orientation, and field-of-view of the user's camera. For each new set of camera parameters, the system computes the visible set — the set of octree nodes that the user sees. According to the user's choice, the system can compute an approximate visible set (b), or a conservative visible set (c). To compute an approximate visible set, iWalk uses the prioritized-layered projection (PLP) algorithm [73]. To compute a conservative visible set, iWalk uses cPLP [74], a conservative extension of PLP. (We will review PLP

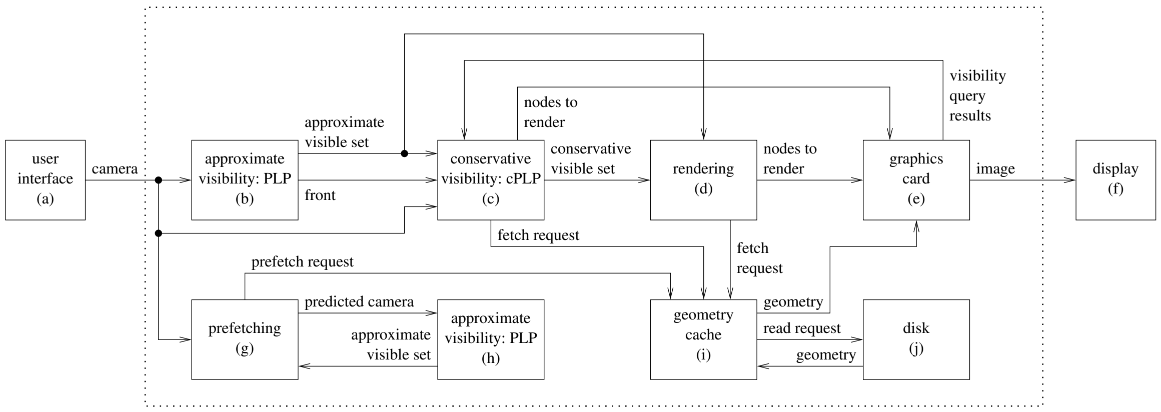


Figure 4.1: The multi-threaded out-of-core rendering approach of the iWalk system. For each new camera (a), the system finds the set of visible nodes using either approximate visibility (b), or conservative visibility (c). For each visible node, the rendering thread (d) sends a fetch request to the geometry cache (i), and then sends the node to the graphics card (e). The prefetching thread (g) predicts future cameras, estimates the nodes that the user would see then (h), and sends prefetch requests to the geometry cache (i).

and cPLP shortly.) For each node in the visible set, the rendering thread (d) sends a fetch request to the geometry cache (i), which will read the node from disk (j) into memory. The rendering thread then sends the node to the graphics card (e) for display (f). To avoid bursts of disk operations, the prefetching thread (g) predicts where the user's camera is likely to be in the next few frames. For each predicted camera, the prefetching thread uses PLP (h) to estimate the visible set, and then sends prefetch requests to the geometry cache (i).

4.2 Review of the PLP and cPLP Algorithms

To better understand the rendering approach, we need to review the visibility algorithms that iWalk uses. When iWalk is running in approximate mode, the rendering thread uses the prioritized-layered projection (PLP) algorithm [73]. In conservative mode, the rendering thread uses the cPLP algorithm [74]. In either mode, the prefetching thread uses PLP.

PLP is an approximate, from-point visibility algorithm that may be thought of as a set of modifications to the traditional hierarchical view frustum culling algorithm [23]. First, instead of traversing the model hierarchy in a predefined order, PLP keeps the hierarchy leaf nodes in a priority queue called the *front*, and traverses the nodes from highest to lowest priority. The front is initialized with the leaf closest to the viewpoint. When PLP visits a node, it adds it to the *visible set*, removes it from the front, and adds the unvisited neighbors of the node to the front. Second, instead of traversing the entire hierarchy, PLP works on a budget, stopping the traversal after a certain number of primitives have been added to the visible set. Finally, PLP requires each node to know not only its children, but also all of its neighbors.

An implementation of PLP may be simple or sophisticated, depending on the heuristic to assign priorities to each node. Several heuristics precompute for each node an opacity value between 0.0 and 1.0 that estimates how likely it is for the node to occlude an object behind it. At run time, the priority of a node is found by initializing it to 1.0, and attenuating it based on the opacity of the nodes found along the traversal path to the node (Figure 4.2). In the next section we describe how we use the precomputed view-dependent visibility coefficients as opacity values.

In addition to being time-critical, another key feature of PLP that iWalk exploits is that PLP can generate an approximate visible set based on just the information stored in the hierarchy structure file created at preprocessing time (Figure 3.1). In other words, PLP can estimate the visible set *without* access to the actual scene geometry, thus allowing us to keep invisible geometry on disk.

PLP does not guarantee image quality, and some frames may show objectionable artifacts. To avoid this problem, the system may use cPLP [74], a conservative extension of PLP that guarantees 100% accurate images. However, cPLP cannot find the visible set from the HS file only, and needs to read the geometry of all potentially visible nodes. The additional disk operations may make cPLP much slower than PLP.

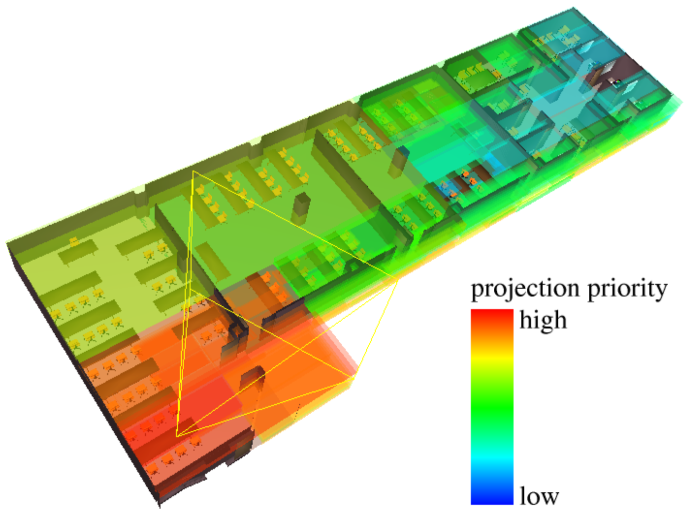


Figure 4.2: A section of the Soda Hall model. At runtime, the iWalk system uses the prioritized-layered projection (PLP) algorithm to estimate the nodes potentially visible from the current view frustum (outlined in yellow). The color of each node indicates the projection priority of the node. Model courtesy of UC Berkeley.

4.3 Extensions to PLP and cPLP

In this section we present our extensions to the PLP and cPLP. We first show how to improve the accuracy of the approximate visible set returned by PLP. Then we show how to exploit new OpenGL extensions to improve the running time of cPLP.

4.3.1 Improving the Accuracy of PLP

In their original paper, Klosowski and Silva [73] computed the opacity of an octree node based on the number of primitives inside the node. One problem with this heuristic is that the number of primitives may not correlate well with visibility. A node with many small triangles clustered together may be less likely to occlude other nodes than a node with a single large triangle.

A better way to estimate the opacity of an octree node is to use the ratio of the projected area of the geometry inside the node relative to the projected area of the node's bounding box. We use the term visibility coefficient to refer to this ratio. Of course, the visibility coefficients depend on the current viewing direction. Computing these coefficients for each viewing direction at runtime would be too expensive, and it would prevent us from achieving interactive frame rates. To avoid this problem, we pick a number of sample viewing directions (typically 20), and precompute the coefficients for these directions (Chapter 3). At runtime, we determine the sample direction that is closest to the current viewing direction, and use the coefficient precomputed for that direction to approximate the opacity of an octree node.

Instead of using a single view-dependent sample to approximate the opacity of a node, we could interpolate between a certain number of closest sample viewing directions. In their image-based rendering system, Debevec et al. [37] use the three closest sample viewing directions to find the weights to blend precomputed images. We experimented with this idea, but found that the additional running time cost was not worth the marginal gain in accuracy.

To further improve PLP's accuracy, we also modify the way to compute a node's projection priority (used in the PLP front). Klosowski and Silva [73] compute projection priorities based on the number of primitives in each node, the normal of the face shared by two nodes, and a penalty factor for adjacencies that are not star-shaped. The number of primitives may not correlate well with visibility, and taking into account shared faces and star-shaped adjacencies creates special cases.

Our approach to compute the node's projection priority is based on sparse ray tracing. We trace a certain number of rays (typically 0.1% of the total number of pixels) from the viewpoint to the scene. Each ray has a contribution value initialized to 1. When a ray hits a node, we assign the ray contribution to the node's projection priority. If multiple rays hit a node, we average their contributions. After a ray passes through a node, we attenuate the ray's contribution by a factor based on the opacity of the node and the distance traveled by the ray inside the node. We terminate a ray if its contribution falls below a certain threshold (typically 0.01). The projection priority of a node not hit by any ray is 0.

Figure 4.3 illustrates how the use of precomputed visibility coefficients and runtime sparse ray tracing improves the computation of projection priorities. The figure shows a section of the Soda Hall model seen by the user's view frustum outlined in yellow. The color of each octree node encodes the projection priority of the node using the same scale used in Figure 4.2. Figure 4.3a shows the priorities computed by the original heuristic. Notice how the priorities decrease smoothly from node to node. Figure 4.3b shows the rays traced from the user's point of view. Figure 4.3c shows the priorities computed by the improved heuristic, which are more accurate. Notice the sharp decreases in priorities from visible nodes to occluded nodes.

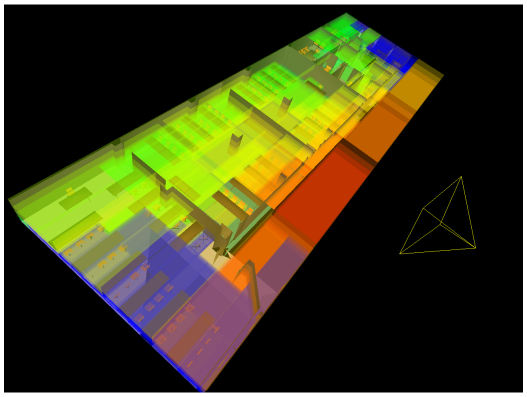
The improved visibility heuristic helps the system in many ways:

Better images in approximate mode: If the system is running in approximate mode, the images generated using the improved heuristic will be more accurate than the images generated using the original heuristic.

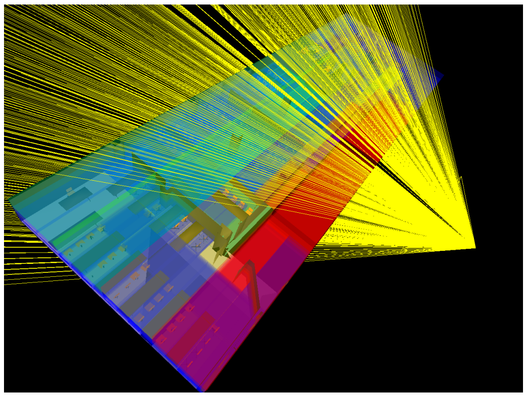
Better frame rates in conservation mode: If the system is running in conservative mode, frame rates will tend to improve, because the initial guess of the visible set will be more accurate, and cPLP will need fewer operations to compute a conservative visible set.

Better prefetching: Using the improved heuristic, the prefetching thread will have a better guess of what nodes to bring from disk into memory, which reduces cache pollution, and avoids stalls due to cache misses.

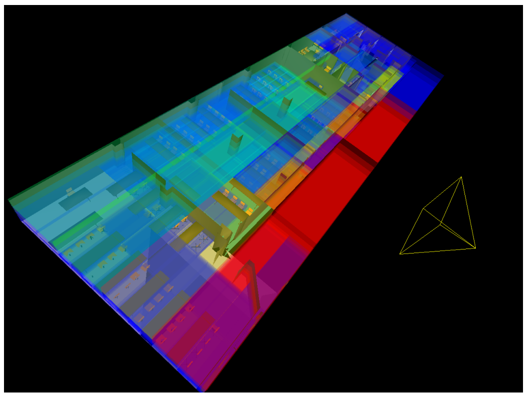
Better LOD selection: Our system uses the estimate of the visibility of an octree node as a hint for what level of detail to use for the node. A better visibility estimate allows us to use lower levels of detail for nodes that are likely to be occluded, which in turn improves cache and disk bandwidth usage.



(a) priorities using original heuristic



(b) using ray tracing to improve heuristic



(c) priorities using improved heuristic

Figure 4.3: Improving the accuracy of PLP.

4.3.2 Improving the Running Time of cPLP

The cPLP algorithm [74] augments the approximate visible set found by PLP to make it a conservative one. The basic idea is to keep projecting visible nodes and adding their potentially visible neighbors (that have not been visited yet) to the front until the front is empty. Klosowski and Silva prove that when the front is empty, all the potentially visible nodes have been found.

Klosowski and Silva also show how to implement cPLP using image-space visibility queries. One approach that they present (and that we implement in our system) is using an item-buffer. First, the geometry of the visible set found so far is rendered on the Z-buffer. Then, the bounding boxes of the nodes currently in the front are rendered on the color buffer with a color that encodes the node number. To determine the visible nodes, the color buffer is read back and searched for node numbers. This approach is portable to any system that supports OpenGL, but reading back the color buffer is still a slow operation on current graphics cards.

Another approach presented by Klosowski and Silva (and reimplemented by us) is to use the HP occlusion test [124]. The HP test allows us to send a piece of geometry to the graphics pipeline, and then ask the graphics card whether or not that geometry was visible. The HP test is typically much faster than reading back the color buffer. Unfortunately, the HP test only allows us to have one occlusion query at a time, and the result of the query is a single boolean value.

Recently, NVIDIA solved the limitations of the HP occlusion test, and gave us the capability we need to implement cPLP very efficiently. The newer NVIDIA graphics cards have an occlusion query extension [110] that allows us to ask about the visibility of *multiple* pieces of geometry in parallel. In addition, the result of each query is not just a boolean flag, but a *count* of the number of visible pixels for the corresponding geometry. The NVIDIA occlusion queries also run faster than the HP tests, because they avoid pipeline stalls by running multiple queries in parallel.

Our implementation of cPLP using the NVIDIA occlusion query extension is roughly 3 times faster than the implementation that reads back the color buffer, and roughly 50% faster than then implementation using HP tests. Because we assume that the dataset is static, we can create an occlusion query per octree node when the program starts, and delete the occlusion queries when the program exits. For best performance, instead of repeatedly issuing a visibility query and then getting its result, we issue multiple visible queries, and later get their results. This decoupling hides the latency of the visibility tests performed by the graphics card.

Another use of the NVIDIA occlusion query extension is in LOD selection. The count of visible pixels returned by the occlusion query of a node gives us a hint of what level of detail to use for the node. We could also get this hint from the implementation using an item-buffer, but not from the implementation using the HP test.

4.4 The Geometry Cache

To render a model larger than main memory, the iWalk system keeps on disk an octree-based representation for the model (Figure 3.1), and loads on demand the contents of the octree nodes that the user sees. Because nodes that are visible in one frame tend to be visible in the next frame (frame-to-frame coherence), iWalk tries to reduce the number of disk operations by maintaining a geometry cache (Figure 4.1i) with the contents of the most recently used nodes.

As the user walks through a model, the conservative visibility thread (Figure 4.1c) and the rendering thread (Figure 4.1d) send fetch requests to the geometry cache (Figure 4.1i). A fetch request contains the identification of an octree node whose contents will be rendered. The geometry cache puts the fetch requests in a queue, and a set of fetch threads process the requests. (Butenhof [19] uses the term *work queue* to refer to a set of threads that accept work requests from a common queue, processing them potentially in parallel.) Each fetch thread pops a request from the fetch queue, and checks whether the contents of the requested node is in memory (a hit) or not (a miss). In the case of a miss, the fetch thread allocates memory for the contents of the requested node, and reads it from disk (Figure 4.1j). If the cache is full, the least recently used nodes are evicted from memory. Finally, the fetch thread puts the requested node in a queue for nodes that are ready to be rendered.

Since the cost of disk read operations is high, most systems try to overlap these operations with other computations by running several processes on a multiprocessor machine [5, 48, 53], or on a network of machines [146, 153]. Along these same lines, our system uses threads on a single processor machine to overlap disk operations with visibility computations and rendering.

The user can configure the number of threads that process the requests in the fetch queue. One advantage of using multiple fetch threads is that it avoids stalls in the rendering pipeline: if a fetch thread processes a miss, that thread will wait until the requested node is read from disk, but the fetch threads that process hits will put the requested nodes in the ready queue, keeping the graphics card busy. Another advantage of using multiple fetch threads is that it gives the operating system kernel a chance to better schedule the read operations when there are concurrent misses.

The geometry cache uses a locking mechanism to prevent multiple threads from modifying or deleting the same node at the same time. The locking mechanism is similar to the one used by the UNIX operating system in its buffer cache [10]. The main difference is that the UNIX buffer cache uses multiple processes for parallelism and signals for synchronization, and we use threads and condition variables [19]. Another difference is that the UNIX buffer cache uses buffers of fixed size, and we use buffers of variable size.

4.5 The From-Point Prefetching Method

The idea behind prefetching is to predict a set of nodes that the user is likely to see next, and bring them to memory ahead of time. Ideally, by the time the user sees those nodes, they will be already in the cache, and the frame rates will not be affected by the disk latency. Systems researchers have studied prefetching strategies for decades [54, 108], and many rendering systems [5, 48, 50, 143] have used prefetching successfully. To our knowledge, all previous prefetching methods that employ occlusion culling have been based on from-region visibility algorithms, and were designed to run on multiprocessor machines. Our prefetching method works with from-point visibility algorithms, and runs as a separate thread in a uniprocessor machine.

Our prefetching method exploits the fact that PLP can very quickly compute an approximate visible set. Given the current camera (Figure 4.1a), the prefetching thread (Figure 4.1g) predicts the next camera position by simply extrapolating the current position and the camera's linear and angular speeds. More sophisticated prediction schemes could consider accelerations and several prior camera locations. For each predicted camera, the prefetching thread uses PLP (Figure 4.1h) to determine which nodes the predicted camera is likely to see. For each node likely to be visible, the prefetching thread sends a prefetch request to the geometry cache (Figure 4.1i). The geometry cache puts the prefetch requests in a queue and a set of prefetch threads process the requests. If there are no fetch requests pending, and if the maximum amount of geometry that can be prefetched per frame has not been reached, a prefetch thread will pop a request from the prefetch queue, and read the requested node from disk (if necessary) (Figure 4.1j).

Figures 4.4–4.6 show the pseudo-code for the main routines run by the threads in the cache. When a client makes a fetch request, a thread executes the fetch routine (and similarly for a prefetch request). When the client is done using that node, it must call the release routine. These routines have to be very careful about sharing

```
fetch(node, ready_queue)
{
    lock cache;
    while (node is busy) {
        wait until node is free;
    }
    mark node as busy;
    if (node is valid) {
        miss = false;
        update node position;
    } else {
        miss = true;
        allocate memory;
    }
    unlock cache;

    if (miss) {
        read node;
    }

    lock cache;
    if (miss) {
        add node to cache;
    }
    if (no fetches pending) {
        broadcast no fetches pending;
    }
    unlock cache;
    add node to ready_queue;
}
```

Figure 4.4: Pseudo-code for the fetch routine.

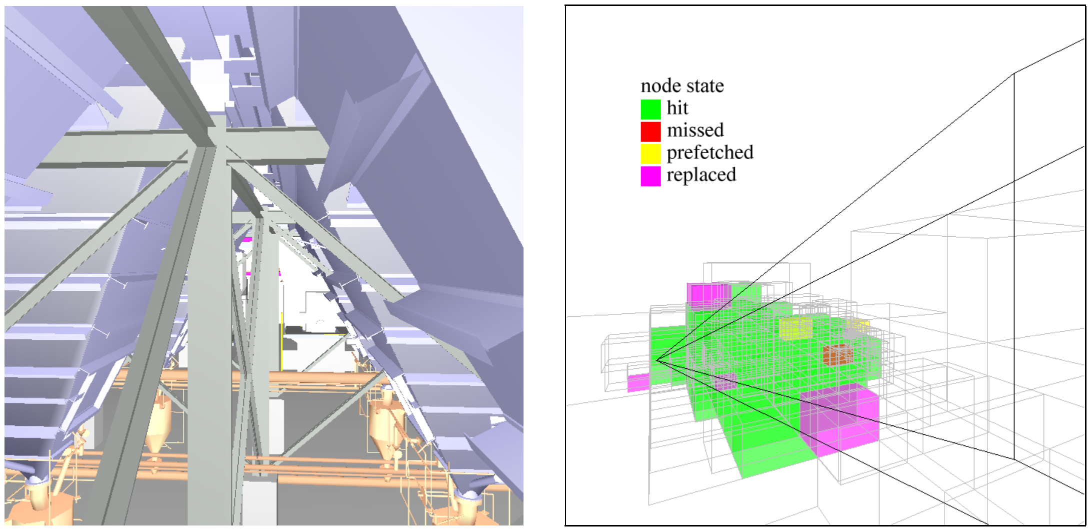
```
prefetch(node, ready_queue)
{
    lock cache;
    while (there are fetch requests pending) {
        wait until no fetch requests pending;
    }
    while (node is busy) {
        wait until node is free;
    }
    mark node as busy;
    if ((node is valid)
        || (reached max prefetch amount per frame)
        || (reached max prefetch request age)) {
        can_read = false;
    } else {
        can_read = true;
        allocate memory;
    }
    unlock cache;

    if (can_read) {
        read node;
        lock cache;
        add node to cache;
        unlock cache;
    }
    add node to ready_queue;
}
```

Figure 4.5: Pseudo-code for the prefetch routine.

```
release(node)
{
    lock cache;
    mark node as free;
    if (node is valid) {
        broadcast memory available;
    }
    broadcast node is free;
    unlock cache;
}
```

Figure 4.6: Pseudo-code for the release routine.



(a) user's view (b) cache view

Figure 4.7: A sample frame inside the power plant model. (a) The image that the user sees. (b) The state of the nodes in the geometry cache.

the cache data structures. To guarantee mutual exclusion, there is a lock to access the cache, and each node has a flag indicating whether it is free or busy. This scheme is similar to the one used in the UNIX buffer cache [10]. Figure 4.7a shows the user's view of the UNC power plant model [147] during a walkthrough session, and Figure 4.7b shows the state of the octree nodes in the geometry cache.

Unlike our from-point prefetching method, from-region prefetching methods decompose the model into cells, and precompute for each cell the geometry that the user would see from any point in the cell. At runtime, from-region methods guess in which cell the user will be next, and load the geometry visible from that cell ahead of time. Our from-point prefetching method has several advantages over from-region prefetching methods. First, from-region methods typically require long preprocessing times (tens of hours), while our from-point method requires little preprocessing (a few minutes). Second, the set of nodes visible from a single point is typically much smaller than the set of nodes visible from any point in a region. Thus, our from-point prefetching method avoids unnecessary disk operations, and has a better chance than a from-region method of prefetching nodes that actually will be visible soon. Third, some from-region methods require that cells coincide with axis-aligned polygons in the model. Our from-point method imposes no restriction on the model's geometry. Finally, the nodes visible from a cell may be very different from the nodes visible from a neighbor of that cell. Thus, a from-region method may cause bursts of disk activity when the user crosses cell boundaries, while a from-point method better exploits frame-to-frame coherence.

4.6 Experimental Results

In this section we report on the performance of our system at runtime. Our main goal was to verify whether we could achieve interactive frame rates and acceptable image quality. Another goal was to study how the many configuration parameters of the system interact, and how they affect the performance perceived by the user. More specifically, there were several questions we wanted tp answer. What is the effect of multi-threading and prefetching on frame rates? What is the impact of frame-to-frame coherence on frame rates? How much better is the approximate visible set computed by PLP when using sparse ray tracing and visibility coefficients?

For the runtime tests, we used the same datasets we used for the preprocessing tests in Chapter 3. Our test machine was different, however. For the runtime tests we used a 2.8 GHz Pentium IV computer with 512 MB of main memory, a 35 GB SCSI disk, and a NVIDIA Quadro 980 XGL graphics card. This machine is slightly better than the machine we used for the preprocessing tests, but it is still an inexpensive PC. This machine also ran Red Hat Linux 8.0.

The user can configure many parameters in our system, including geometry cache size, number of fetch threads, number of prefetch threads, maximum amount of prefetched geometry per frame, primitive budget for approximate visibility, target frame rate, and image resolution. These parameters depend mainly on the triangle throughput of the graphics card and the disk bandwidth. For our test machine, we found that the following configuration worked well: 256 MB of geometry cache, 8 fetch threads, 1 prefetch thread, a maximum of 2 MB of prefetched geometry per frame, a budget of 280,000 triangles per frame for approximate visibility, a target frame rate of 10 fps, and image resolution of 1024×768.

4.6.1 UNC Power Plant Results

To analyze the overall performance of our system, we measured the frame rates achieved when walking through the power plant model along several predefined paths (which enabled repeatable conditions for our experiments). Note that our algorithms made no assumptions on the paths being known beforehand; complete camera interactivity is always available to the user. The first path used has 36,432 viewpoints, visits almost every part of the model, and requires fetching a total of 900 MB of data from disk. Using the above configuration, our system rendered the frames along that path in 74 minutes. Only 95 frames (0.26%) caused the system to achieve less than 1 fps. The mean frame rate was 9.2 fps, and the median frame rate was 9.3 fps.

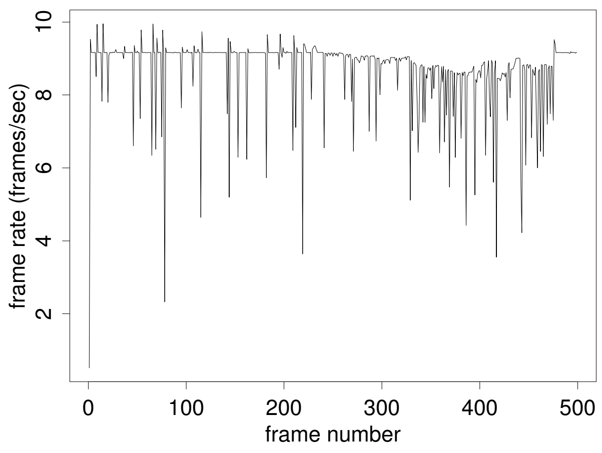
To analyze the detailed performance of our system, it is easier to use shorter paths. For this purpose, we used a 500-frame path which required 210 MB of data to be read from disk. If fetched independently, the maximum amount of memory necessary to render any given frame in approximate mode would be 16 MB.

To study how multiple threads improve the frame rates, we ran tests using three different configurations. The first configuration is entirely sequential: a single thread is responsible for computing visibility, performing disk operations, and rendering. The second configuration adds asynchronous fetching to the first configuration, allowing up to 8 fetch threads. The third configuration adds an extra thread for speculative prefetching to the second configuration, allowing up to 2 MB of geometry to be prefetched per frame. Figure 4.8 shows the frame rates achieved by these three configurations for the 500-frame path. For the purely sequential configuration, we see many downward spikes that correspond to abrupt drops in frame rates, which are caused by the latency of the disk operations, and spoil the user's experience. The first spike happens because the cache is initially empty. When we add asynchronous fetching, many of the downward spikes disappear, but too many still remain. The user's experience is much better, but the frame rate drops are still disturbing. When we add speculative prefetching, all significant downward spikes disappear, and the user experience is smooth. Note that the gain in interactivity comes entirely from overlapping the independent operations. The three configurations achieve exactly the same image accuracy (Figure 4.9).

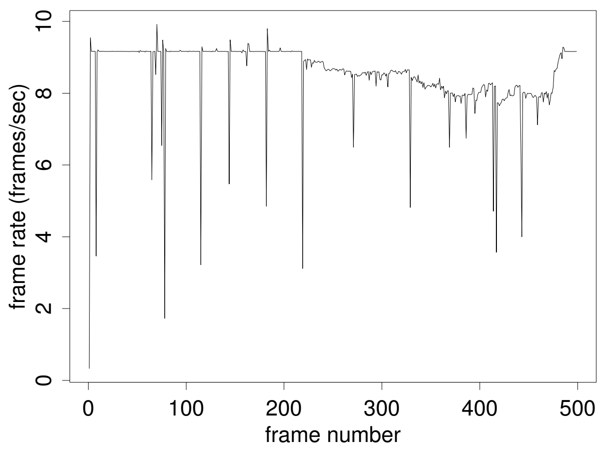
Figure 4.10 shows why prefetching improves the frame rates. The charts compare the amount of geometry that the system reads from disk per frame for the second and third configurations described above. Prefetching greatly reduces the need to fetch large amounts of geometry in a single frame, and thus helps the system to maintain higher and smoother frame rates.

Figure 4.11 shows that the user speed is another important parameter in the system, and has to be adjusted to the disk bandwidth. When the user speed increases, the changes in the visible set are larger. In other words, as the frame-to-frame coherence decreases, the amount of data the system needs to read per frame increases. Thus, caching and prefetching are more effective if the user moves at speeds compatible with the disk bandwidth. The figure also indicates that higher disk bandwidth should improve frame rates.

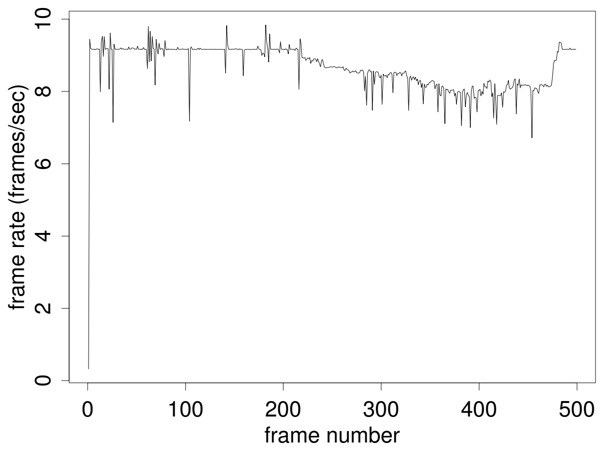
The frame rates reported above were obtained when the system was running in approximate mode and with LOD management turned off. In conservative mode, to obtain similar frame rates we need to turn on LOD management, otherwise the frame rates are not interactive. When using LODs, the frame rates in conservative



(a) sequential fetching and rendering



(b) concurrent fetching and rendering



(c) concurrent fetching, rendering, and prefetching

Figure 4.8: Using multiple threads to improve frame rates.

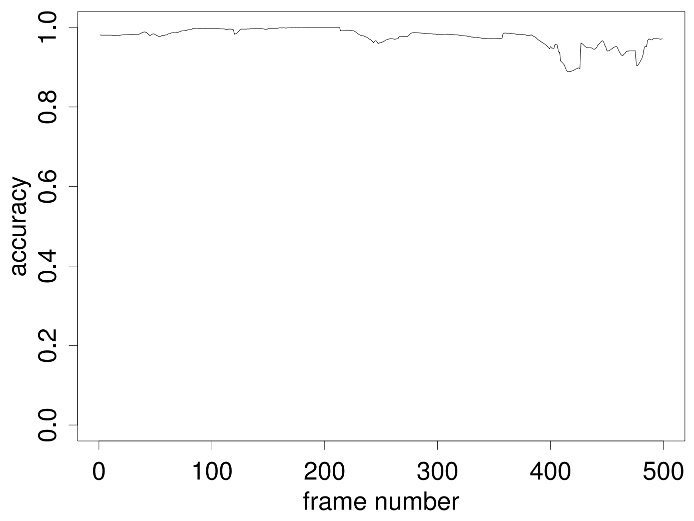
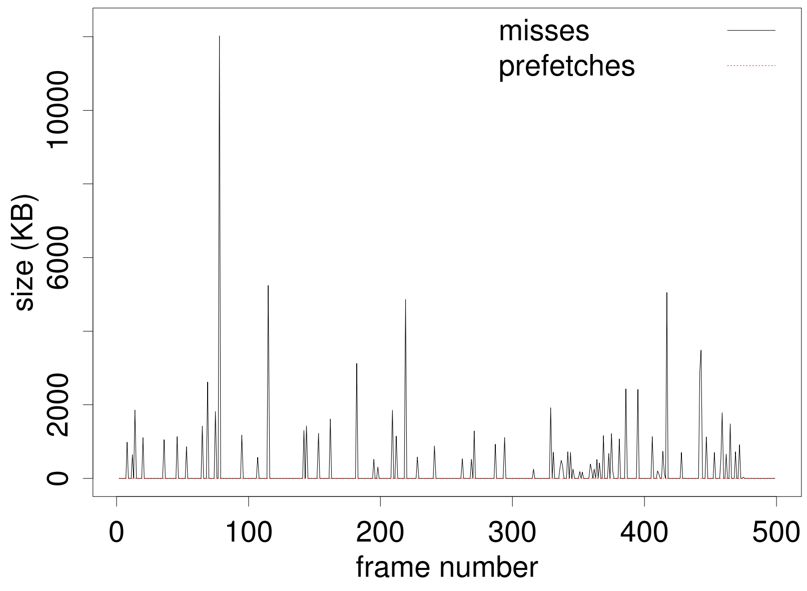


Figure 4.9: Image accuracy for a 500-frame walkthrough of the power plant model when using approximate visibility. The vertical axis represents the fraction of correct pixels in the approximate images in comparison to the conservative images. The minimum accuracy was 89%, and the median accuracy was 98%.

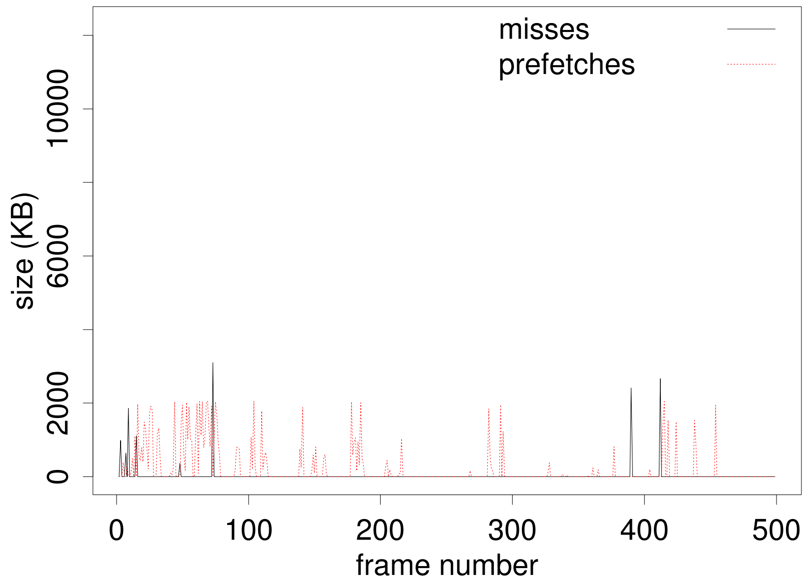
mode are almost the same as the frame rates in approximate mode. The difference between these configurations is the quality of the images generated. The combination of conservative visibility with LOD management tends to produce better final images than approximate visibility, especially for exterior views of the model.

When using approximate visibility without LODs, the nodes deemed to be visible are rendered in full resolution, and the nodes deemed to be over the primitive budget are not rendered at all. The final image has areas with no error and areas with large error. In contrast, when using conservative visibility with LODs, few areas have no error, but no area has large error. The total number of wrong pixels may be similar between these two approaches, but the images produced by the second approach are more pleasing to the user.

Figure 4.12 shows an interior view of the power plant model. Because this model has very high depth complexity, the visible set is very small when the user is inside the model. In this case, the original PLP heuristic is enough to produce very accurate images. On the other hand, Figure 4.13 shows that for exterior views the improved heuristic produces approximate images much closer to the conservative images.

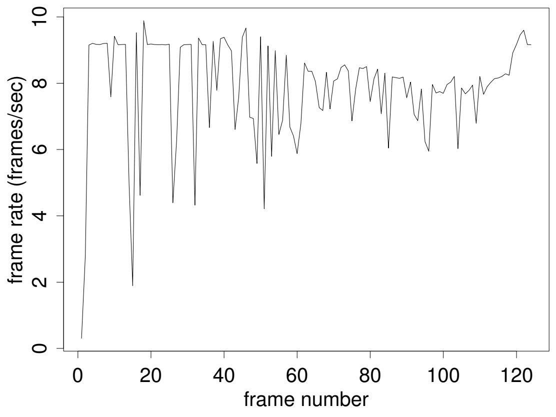


(a) without prefetching

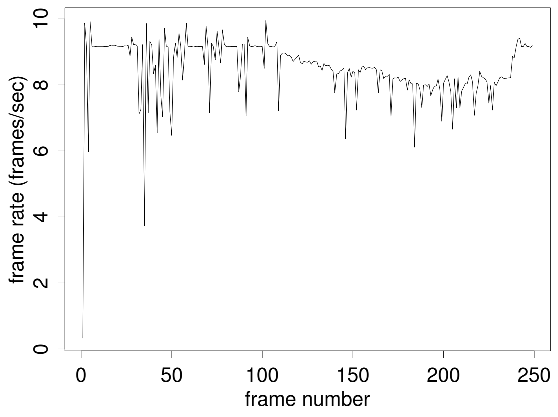


(b) with prefetching

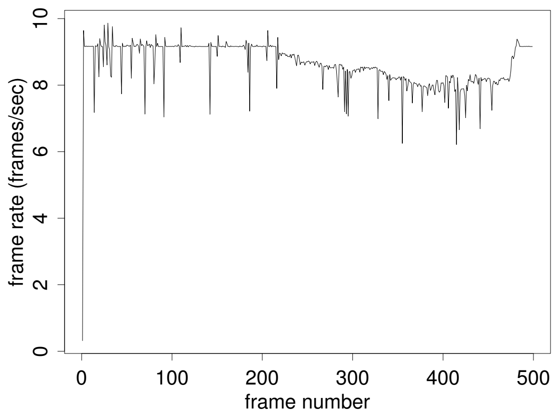
Figure 4.10: Using prefetching to amortize the cost of disk operations. We measured the amount of geometry fetched per frame without prefetching (a) and with prefetching (b). Prefetching amortizes the cost of bursts of disk operations over frames with few disk operations, thus eliminating or alleviating most frame rate drops. The system was configured to prefetch at most 2 MB per frame.



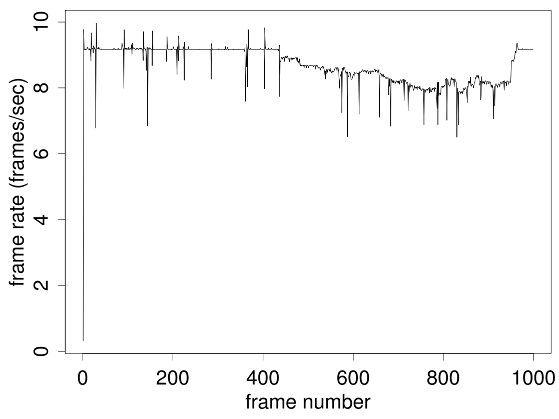
(a) very high user speed



(b) high user speed

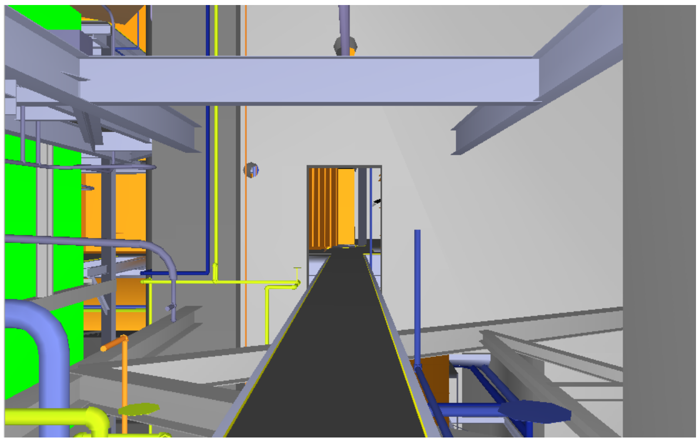


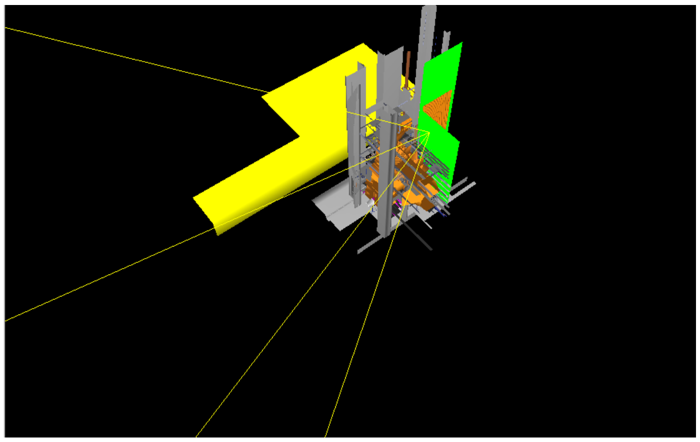
(c) normal user speed



(d) low user speed

Figure 4.11: Adjusting the user speed to the disk bandwidth. We measured the frame rates along a camera path inside the power plant model for different user speeds (or equivalently, for different number of frames in the path). If the user moves too fast, the frame rates are not smooth. The faster the user moves, the larger the changes in occlusion, and therefore the larger the number of disk operations.


(a) user's view


(b) bird's eye view

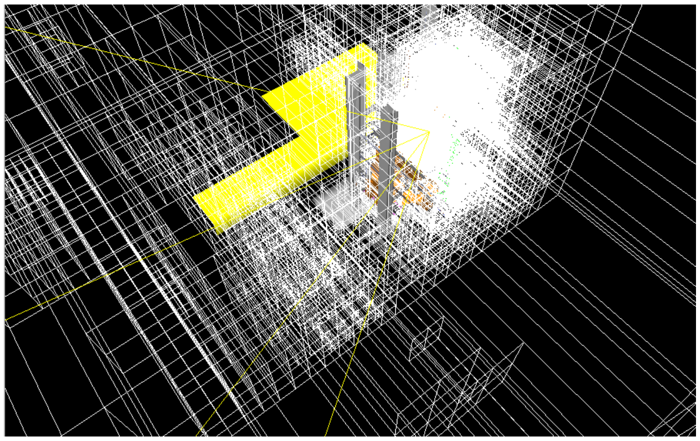
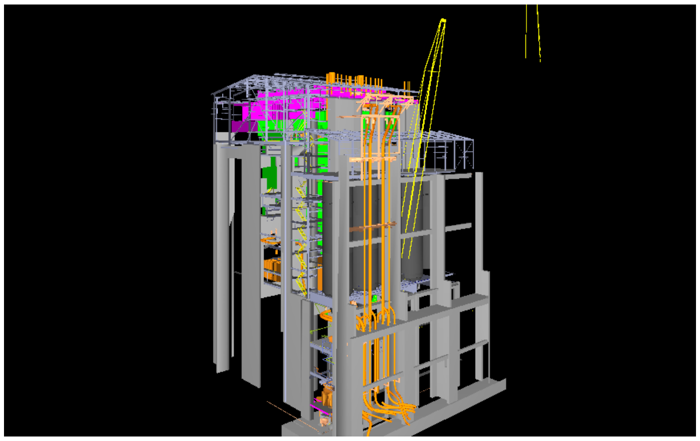
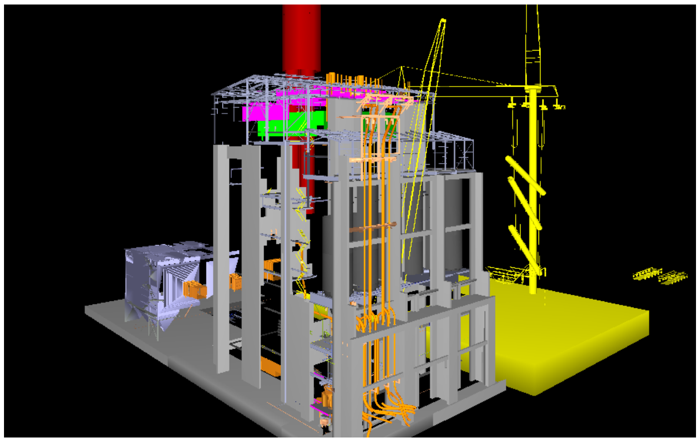

(c) bird's eye view with octree

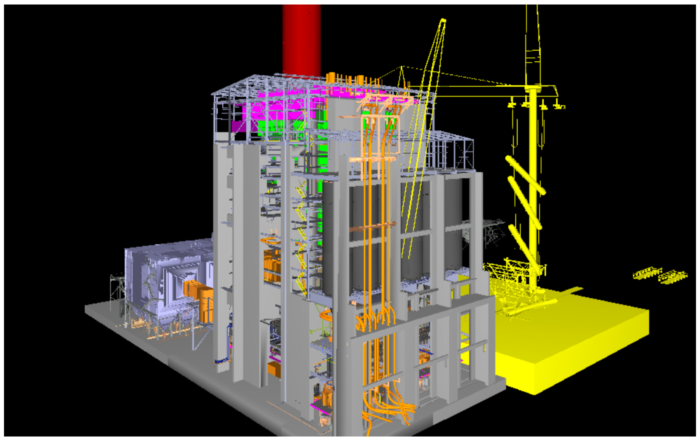
Figure 4.12: Interior view of the power plant model.



(a) using original approximate visibility



(b) using improved approximate visibility



(c) using conservative visibility

Figure 4.13: Exterior view of the power plant model.

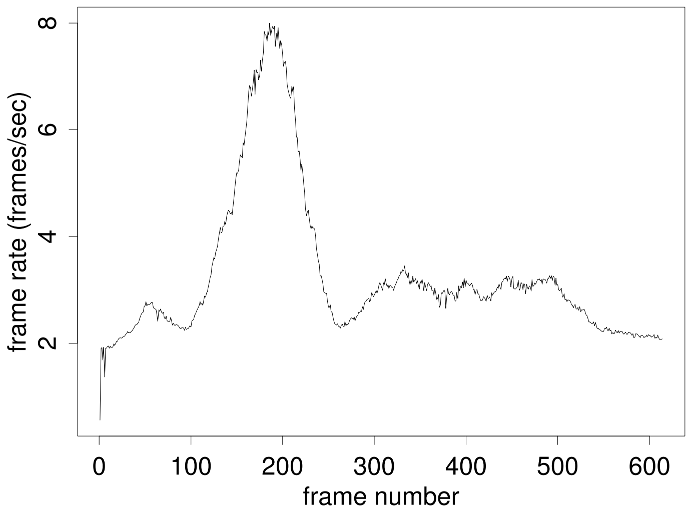


Figure 4.14: Frame rates for the LLNL isosurface dataset.

4.6.2 LLNL Isosurface Results

For the LLNL isosurface dataset, we measured the frame rates achieved by our system using a 615-frame camera path. This path was recorded in a session in which the user starts by inspecting the entire model from the outside, rotating it around. The user then moves close to a particular area of the surface, and finally moves back to see the entire model again.

Because of the huge size of this model, approximate visibility alone, i.e., without LOD management, is not accurate for outside views. When we combine approximate visibility with LOD management, the images are more accurate, but still far from correct. If we use conservative visibility alone, the frame rates are too low (up to several minutes per frame). The only configuration able to handle this model at interactive frame rates and acceptable image quality is the combination of conservative visibility and LOD management.

Figure 4.14 shows the frame rates achieved using conservative visibility combined with LOD management. The overall mean frame rate was 3 fps. The frame rates when the user gets closer to a particular area are higher because a large part of the model is not visible then.

4.6.3 Summary of Rendering Results

Using an inexpensive PC, our system was able to render both the UNC power plant and the LLNL isosurface at interactive frame rates and acceptable image quality. The use of multiple threads for asynchronous fetching and prefetching greatly improves the frame rates, but the performance of the system is heavily dependent on frame-to-frame coherence. The use of sparse ray tracing and visibility coefficients significantly increases the accuracy of the approximate visible set estimated by PLP. With better visibility estimation, the system delivers better images when running in approximate mode and better frame rates in conservative mode. The system brings to memory data that is more likely to be visible, and has a better hint (than distance or projected area) for LOD selection.

For the UNC power plant model, the best previously published out-of-core rendering results are from the ray tracing system of Wald et al. [146]. Our system achieves higher frame rates than theirs, but their system delivers better image quality.

For the LLNL dataset, the best out-of-core rendering results are from Lindstrom [82]. Despite not using occlusion culling, his system is able to deliver frame rates similar to the ones achieved by our system on similar hardware.

Chapter 5

Out-Of-Core Parallel Rendering

Chapter 4 described the out-of-core rendering approach of the iWalk system. Although iWalk is able to handle models larger than main memory, it only produces low-resolution (1024×768) images at interactive frame rates. This chapter describes a parallel system that uses iWalk as a building block, and delivers high-resolution (4096×3072) images at the same frame rates or faster.

5.1 Choosing the Hardware

A traditional approach to parallel rendering has been to use a high-end parallel machine. More recently, with the explosive growth in power of inexpensive graphics cards for PCs, and the availability of high-speed networks, using a cluster of PCs for parallel rendering has become an attractive alternative, for many reasons [80, 116]:

Lower cost A cluster of commodity PCs, each costing a few thousand dollars, typically has a better price/performance ratio than a high-end, highly-specialized supercomputer that may cost up to millions of dollars.

Technology tracking High-volume off-the-shelf parts typically improve at faster rates than special-purpose hardware. We can upgrade a cluster of PCs much more frequently than a high-end system, as new inexpensive PC graphics cards become available every 6-12 months.

Modularity and flexibility We can easily add or remove machines from the cluster, and even mix machines of different kinds. We can also use the cluster for tasks other than rendering.

Scalable capacity The aggregate computing, storage, and bandwidth capacity of a PC cluster grows linearly with the number of machines in the cluster.

Thus we have chosen to use a cluster of PCs to drive a multi-projector tiled display to create high resolution images.

5.2 Choosing the Parallelization Strategy

As we have discussed in Chapter 2, there are three categories of parallelization strategies: sort-first, sort-middle, and sort-last [94]. Sort-first approaches divide the screen into tiles, and assign each tile to a different processor, which is responsible for all of the rendering in its tile. Sort-middle approaches assign an arbitrary subset of primitives to each geometry processor, and a tile of the screen to each rasterizer. A geometry processor transforms and lights its primitives, and then sends them to the appropriate rasterizers. Sort-last approaches assign an arbitrary subset of the primitives to each renderer. A renderer computes pixel values for its subset, and then transfer these pixels to compositing processors.

Given our goal and constraints, we have chosen a sort-first approach for two main reasons. First, sort-first processors implement the entire pipeline for a portion of the screen [94], which is exactly the case for which PC graphics cards are optimized. And second, interactive applications tend to exhibit high frame-to-frame coherence, which sort-first approaches exploit well.

We rejected sort-middle approaches because they require a tight integration between the geometry processing and rasterization stages, which is only available on high-end graphics machines [4, 47, 96]. On PC graphics cards, there is no fast access to the results of the geometry processing [116].

We rejected sort-last approaches for two main reasons. First, sort-last approaches require very high bandwidth for pixel compositing [94]. For example, suppose each tile of our screen has 1280×1024 pixels, and that we store 7 bytes per pixel (4 for color and 3 for depth). If our target frame rate is 10 frames per second, each rendering server would need 87.5 MB/s of network bandwidth just to transfer pixels. Some researchers have addressed this problem by designing specialized hardware for pixel compositing [92, 135], but these machines are expensive. The other reason why we rejected sort-last approaches is that they would prevent us from implementing occlusion culling based on image-space queries.

5.3 The Parallel Rendering System

To implement a sort-first approach, the main challenge is to handle the redistribution step [100]. During the geometry processing, after a pre-transformation step determines into which screen tiles each primitive falls, the primitives must become available in main memory at the renderers responsible for those tiles. To get around the redistribution step, some systems simply replicate in main memory the entire model on each renderer. This approach, of course, does not scale with respect to model size. More sophisticated systems replicate the model only on a subset of the renderers [115]. Our system keeps a hierarchical partitioning of the model on disk, and each renderer loads the visible parts of the model into its memory cache on demand. Since the disk where we keep the model may be a shared network disk or a local disk, this approach imposes virtually no limit on the model size.

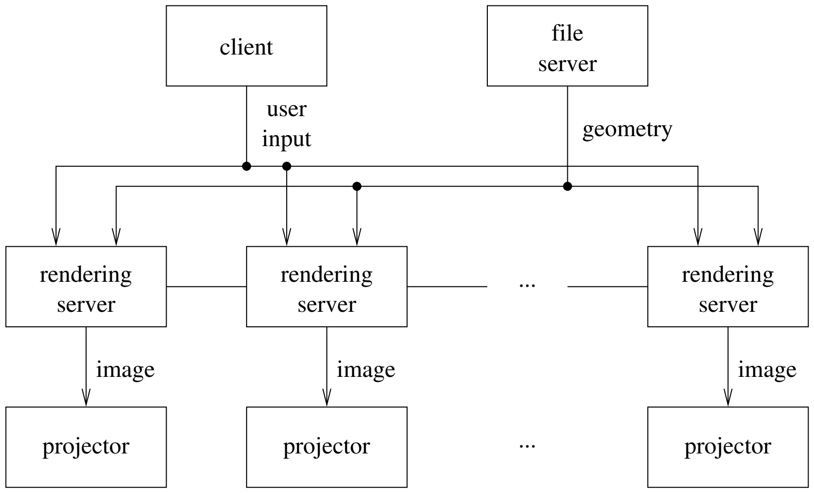


Figure 5.1: The out-of-core sort-first architecture.

Figure 5.1 shows a diagram of our system. A client machine is responsible for processing user interface events. For each display tile there is a dedicated rendering server. At each frame, the client sends the current viewing parameters to the rendering servers. Note that the client does almost no work. The rendering servers run the sequential rendering algorithms (from iWalk) that we presented in Chapter 4, with a few modifications that we will discuss below. Each renderer reads the parts of the model it needs from a shared network disk in the file server, and sends the resulting image to one of the display projectors. Optionally, each renderer may read its primitives from a local copy of the model. Note that this copy is on disk, not in main memory. Since disk space is cheap, having a local copy of the model on disk might not hurt the scalability of the system.

Each rendering server is an MPI task and runs basically the same code that iWalk runs, with a few differences. First, since each renderer is responsible for a tile of the display wall, it performs occlusion culling using only the part of view frustum that belongs to it. Second, each renderer receives input events from the client through socket communication, instead of directly from the user. Finally, to synchronize the renderers, we add an MPI barrier at the end of the rendering loop, right before swapping front and back buffers.

We only use MPI to start and synchronize the servers. The client does not need to have an MPI implementation available. The client machine only transmits the current viewing parameters to the rendering servers, and may therefore be as lightweight as a handheld computer. Some systems perform load balancing computations on the client machine, in which case the client may become a bottleneck [118].

Our approach to synchronize the rendering servers is to rely on the MPI barrier, which has a non-trivial latency. An alternative would be using multi-pipe graphics cards with inter-pipe synchronization (genlock). Some new PC graphics cards such as the NVIDIA FX 3000G [102] provide genlock, but their price is still prohibitive.

5.4 Experimental Results

In this section we report the results of the performance and scalability experiments we ran for our parallel rendering system. The main goal of these experiments was to study how the system scales with the number of processors and image resolution. We also wanted to compare the performance of the system when the renderers read data from a shared file server versus from a copy on a local disk. We report results for two different clusters. The first cluster is about three years old, and the second cluster is about one year old.

5.4.1 Results for the Old Cluster

The old cluster consisted of 16 rendering servers and a file server. Each rendering server was a 900 MHz AMD Athlon with 512 MB of main memory, an NVIDIA GeForce2 graphics card, and an IDE hard disk. The file server was a 400 GB disk array composed of eight SCSI disks configured as two 200 GB striped disks. As we have discussed, the client machine does very little processing, and therefore its hardware details are not important. In these tests, the client machine was a 700 MHz

Pentium III. All the machines were connected by switched gigabit ethernet, and ran Red Hat Linux 7.2. The servers ran MPI/Pro 1.6.3 over TCP/IP for synchronization.

For the old cluster, we only ran experiments with the UNC power plant model [147] (Figure 3.7). We ran tests on clusters of 1, 2, 4, 8, and 16 rendering servers. We used a pre-recorded camera path of 500 frames, and for each cluster size we collected statistics for both approximate visibility mode (using PLP) and conservative visibility mode (using cPLP). In both cases, LOD management was turned off. For each cluster size, we first ran the tests reading the model from the file server, and then reading the model from copies on the local disks.

PLP Results

Here we report the results of the experiments we ran in approximate visibility mode, i.e., using PLP to estimate the visible geometry. In typical use, we configure the system according to the triangle throughput of the graphics cards, the bandwidth of the disks, the desired frame rate, and the desired image accuracy. When using a cluster of 16 rendering servers, we usually give each renderer a budget of 70,000 triangles per frame and a geometry cache of 256 MB. This configuration allows us to generate 12-megapixel images of the power plant with a median accuracy of 99.3% at a median frame rate of 10.8 fps. For the scalability analysis that follows, we used instead a total budget of 400K triangles per frame, so that the system would be usable even when configured with only one rendering server.

When we run our system in approximate mode on a single machine, the frame rates depend mostly on the number of triangles rendered and the number of disk accesses; the image resolution has a smaller influence. As we add more machines to the cluster, the total resolution increases, but the resolution of each renderer remains fixed. The total triangle budget per frame for PLP also remains fixed, thus the triangle budget of each renderer decreases.

Ideally, if we doubled the number of machines in the cluster, we would get twice the frame rate and the same image quality. In practice, several factors prevent us from achieving that. First, there is duplication of effort. In sort-first, if a primitive overlaps multiple tiles, it is fetched and rendered multiple times. Since the chances of overlap increase as we add processors, the demands for I/O bandwidth and triangle throughput also increase. There are additional communications costs as well. At the end of each frame, there is an MPI barrier to synchronize all the servers. Finally, the likelihood of load imbalance increases as the number of processors increases, which may have a negative effect on both the frame rate and the image accuracy.

Figure 5.2 shows the frame rates achieved by our system when using PLP, as we vary the cluster size (1, 2, 4, 8, and 16 PCs) and the type of disk (network or local). [1] For these small clusters, the median frame rates (the horizontal lines in the interior of the boxes) improved substantially with the number of PCs. On the other hand, the spread of the frame rates (the height of the boxes) increased. For all configurations, there were very few stalls (the horizontal lines outside the whiskers). A surprising fact is that the disk type has almost no influence on the frame rates. The bandwidth of our network disk, measured using the Bonnie benchmark [17] from a rendering server, is 7.8 MB/s. The similarity between the frame rates for network and local disks indicates that the total bandwidth required by the rendering servers is usually less than the bandwidth of the network disk. We believe the bandwidth requirement is so low because our caching and prefetching schemes are exploiting well the frame-to-frame coherence in our test paths.

[1]How to read the box plots. (From the S-Plus user's guide [89].) The horizontal line in the interior of the box is located at the median of the data, and estimates the center of the distribution for the data. The height of the box is equal to the *interquartile distance*, or IQD, which is the difference between the third quartile of the data and the first quartile, and indicates the spread of the distribution for the data. The whiskers (the dotted lines extending from the top and bottom of the box) extend to the extreme values of the data or a distance of $1.5 \times$ IQD from the center, whichever is less. For data having a Gaussian distribution, approximately 99.3% of the data falls inside the whiskers. Data points that fall outside the whiskers may be outliers and so they are indicated by horizontal lines.

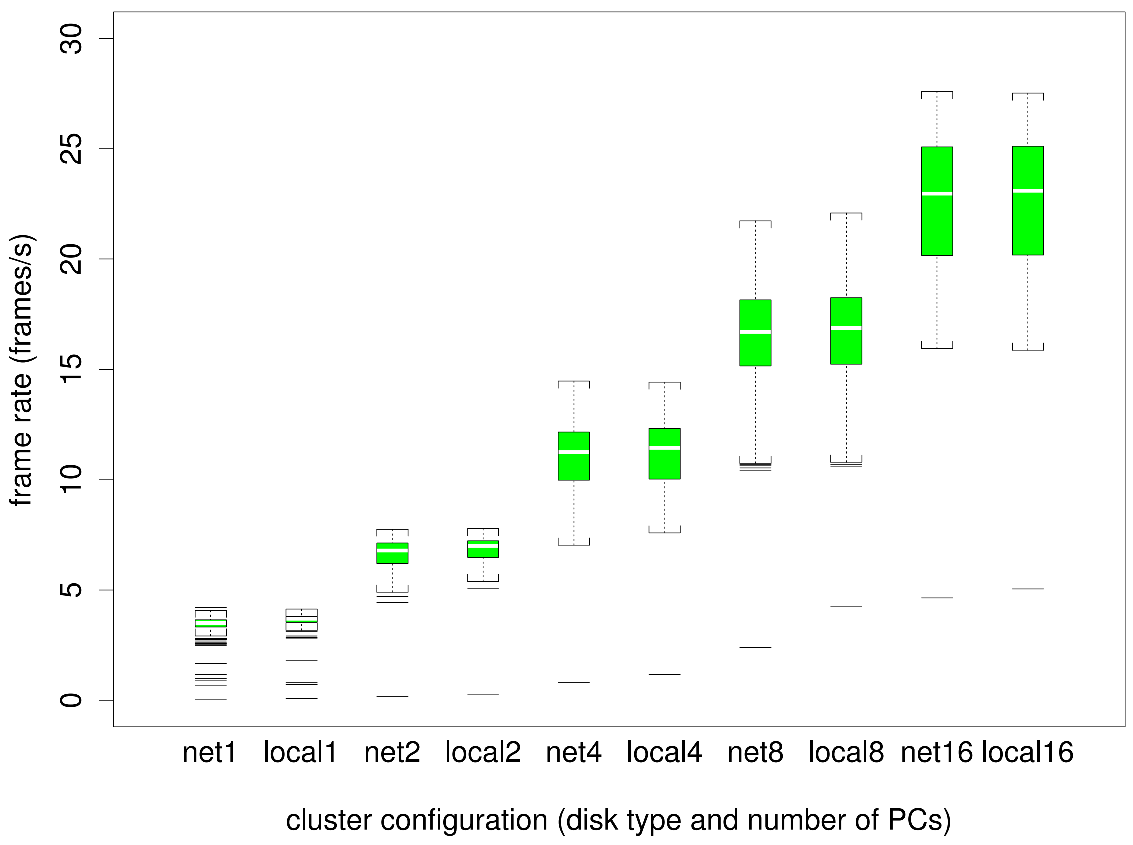


Figure 5.2: Frame rates for PLP in the old cluster as we vary the cluster size and the disk type. We ran tests on clusters of 1, 2, 4, 8, and 16 PCs, for both network and local disks. The median frame rates improve substantially with the number of PCs, and the disk type makes almost no difference.

We measured the accuracy achieved by our system for the tests above by comparing the pixels in the images produced by PLP and the pixels in the correct images. For this particular camera path, which was inside the power plant, in an area with high depth-complexity, PLP estimates the visible set very well. For a single machine, PLP achieves a median accuracy of 99.6%. If the triangles were uniformly distributed across the screen, for a constant total triangle budget B, a cluster with $P > 1$ rendering servers, each of which with a triangle budget of B/P to render its screen tile, would achieve the same accuracy as a single machine. But typically the distribution of the triangles is not uniform, and B/P triangles may be too few for some servers and too many for others. For paths inside the model, this load imbalance is usually small, and the accuracy drops slowly with the cluster size. For the test path, the median accuracy achieved by the cluster with 16 servers was 93%, which is high and typical for paths inside the model. For paths outside the model, the accuracy may be much lower, unless we turn on level-of-detail management.

cPLP Results

Here we report the results of the experiments we ran in conservative visibility mode, i.e., using cPLP to estimate the visible geometry. Conservative visibility introduces another obstacle for ideal scalability. Recall that there is a one-to-one correspondence between servers and projectors. Thus, when we increase the number of servers, although each server becomes responsible for a smaller part of the view frustum, that part will be rendered at higher resolution. As a result, the amount of geometry visible through that part of the view frustum that we need to fetch and render may not decrease. In theory, it could even increase. Since the size of the problem may grow with the cluster size, we cannot expect linear scalability. Turning on level-of-detail management allows us to get almost linear scalability in frame rates, at the cost of some loss in image accuracy.

Figure 5.3 shows the frame rates achieved by our system when using cPLP, as we vary the cluster size and the type of disk. Recall that PLP can estimate a visible set based only on the hierarchy structure file created at preprocessing time, but cPLP needs to read the actual scene geometry. Thus, without LOD management, cPLP needs to perform many more disk accesses than PLP, and the frame rates for cPLP are much lower than those for PLP. In terms of scalability, even though the *maximum* frame rates increase substantially with cluster size, the *median* frame rates remain roughly the same. In terms of disk type, the network disk was able to match once again the performance of the local disks, which indicates that making local copies of the model on each server may be unnecessary.

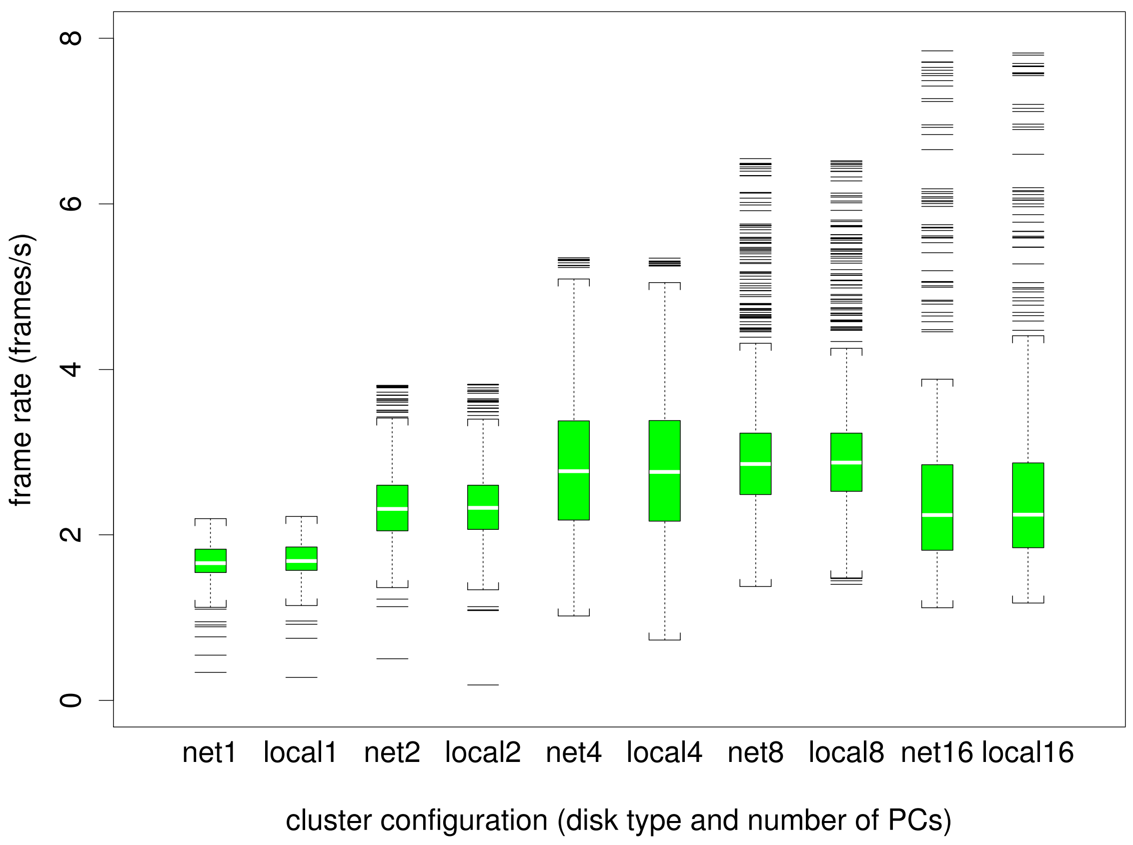


Figure 5.3: Frame rates for cPLP in the old cluster as we vary the cluster size and the disk type. We ran tests on clusters of 1, 2, 4, 8, and 16 PCs, for both network and local disks. The median frame rates stay roughly the same, and the disk type makes almost no difference.

5.4.2 Results for the New Cluster

The new cluster consists of 8 rendering servers and a file server. Each rendering server is a 2.8 GHz Pentium IV computer with 512 MB of main memory, a 35 GB SCSI disk, and a NVIDIA Quadro 980 XGL graphics card. The file server is similar, but in addition it has a 200 GB SCSI disk. The client machine is identical to the rendering servers. The new cluster also uses gigabit ethernet for connectivity. All machines run Red Hat Linux 8.0. The servers use MPICH 1.2.5 for synchronization.

For the new cluster, we ran experiments for both the UNC power plant model [147] (Figure 3.7) and the LLNL isosurface dataset (Figure 3.10). We ran tests on clusters of 1, 2, 4, and 8 rendering servers. All tests for the new cluster used conservative visibility in combination with LOD management. As we did for the old cluster, for each cluster size we ran the tests first reading data from the file server, and then reading data from copies on the local disks.

For both datasets, the mean frame rates improved substantially with the number of PCs, and once again the disk type makes almost no difference. In practice, instead of getting higher mean frame rates with higher variance, we prefer to put a cap on the frame rates (typically 10 fps), and obtain lower variance. For frames that could be rendered faster, the rendering thread waits for the frame time. Meanwhile, the prefetching thread has a better chance to be allowed to bring data from disk into memory, which reduces stalls due to cache misses, and lowers frame rate variance.

When using 8 rendering servers, each rendering a 1280 by 1024 tile, we were able to render the UNC power plant model at 10 frames per second on average, with very little variance. For the LLNL isosurface, we could sustain 4–5 frames per second for exterior views and 8–10 frames per second for interior views.

5.4.3 Summary of Parallel Rendering Results

The sort-first parallel rendering extension of our system allows us to scale the resolution of an application without any loss in performance. On the other hand, unlike Chromium [68], our parallel architecture requires changes in the application source code. The best parallel rendering systems for clusters we know are the ray tracing system of Wald et al. [146] and the sort-last system of Moreland et al. [98]. The system of Wald et al. produces more photorealistic images, while our system delivers higher frame rates and higher image resolution. Our system and the system of Moreland et al. achieve similar results on similar hardware for the LLNL isosurface dataset. Because their system uses a sort-last approach, it scales better than ours with model size. On the other hand, because we use a sort-first approach, we can take advantage of image-space occlusion queries.

Chapter 6

Conclusions

This chapter ends the dissertation. Here we summarize our work, point out our most important research contributions, and discuss possible directions for future work. We end by showing how to obtain the source code for our system.

6.1 Summary

We have presented iWalk, a system for interactive and high-resolution visualization of large datasets on commodity PCs with small memory. To handle datasets larger than main memory, the system uses a new set of out-of-core preprocessing and runtime algorithms. The preprocessing algorithms break the dataset into manageable pieces using an octree, and precompute visibility information and levels of detail for each octree node. The runtime algorithms keep the bulk of the dataset on disk, and bring octree nodes into a memory cache on demand. To achieve interactive and smooth frame rates, the system combines level-of-detail management with occlusion culling, and uses multiple threads to overlap visibility computation, rasterization, fetching, and prefetching. In addition, the system exploits recent OpenGL extensions such as vertex arrays and occlusion queries. To produce high resolution images, a sort-first parallel extension of the system uses a cluster of PCs to drive a multi-tile display.

The out-of-core preprocessing and runtime algorithms are simple, efficient, and make no assumption about the datasets. The combination of these algorithms is a practical and scalable system that allows us to use inexpensive PCs to visualize datasets that until recently would require expensive high-end graphics workstations. By being able to run on inexpensive hardware, our system can help to bring visualization of large datasets to a broader audience.

6.2 Contributions

The main research contributions of our work are:

An out-of-core algorithm to build an octree. Our algorithm is fast and automatic, i.e., it needs no user intervention. In addition, it is incremental, i.e., it allows us to add new data to an existing octree, which is important for some applications (e.g., 3D scanning).

Extensions of the PLP visibility algorithm. Our ray-tracing based heuristic for PLP provides more accurate approximate visible sets at little extra preprocessing and runtime cost. Our hardware-assisted extension of cPLP, combined with level-of-detail management, requires very little preprocessing, and makes conservative occlusion culling practical on commodity hardware.

An out-of-core, from-point prefetching algorithm. Our prefetching algorithm exploits PLP's ability to estimate a visible set without having to read geometry from the disk or use the graphics card. The algorithm runs when the CPU is idle waiting for the disk, the graphics card, or the next frame. Thus, a substantial improvement in frame rates comes at almost no additional preprocessing or runtime costs. We believe our system is the first to employ a prefetching method based on a from-point visibility algorithm.

An out-of-core sort-first parallel rendering architecture. The architecture is a simple and yet effective way for an application to increase the resolution of the output images, and obtain the same or faster frame rates. The architecture keeps the data on disks on the server side, thus avoiding a potential bottleneck on the client, and better utilizing the rendering power of the servers.

A system that integrates these techniques. We prove that our techniques are practical by integrating them in a system that can handle datasets with hundreds of millions of triangles. In addition, we make our system open source, so other researchers can study it, extend it, or compare it with their own systems.

6.3 Future Work

There are many possible avenues for future work. The following list is in increasing order of estimated difficulty to implement:

Add geometry and appearance quantization. The system currently requires 19 bytes per vertex (12 bytes for position, 3 bytes for normal, and 4 bytes for color). We could quantize these geometry and appearance attributes to save storage. This change would reduce the amount of data transfered from the CPU to the graphics card, and would free up space in the cache for more octree nodes.

Eliminate geometry replication. If a triangle intersects multiple octree nodes, the system currently replicates the triangle in all the intersected nodes. Geometry replication can easily double the size of the dataset, especially when the spatialization granularity is fine. Eliminating this problem would be a very welcome change to our system.

Add support for textures. The system currently does not support texture mapping. Adding this feature has the potential to improve the image quality significantly. Although graphics cards have support for texture mapping, they have very small texture memory. Thus, adding this feature should be simple, but may not be trivial,

Finish support for volumes. The system already has some support for volume rendering [70]. By using an out-of-core extension of a well-known cell-projection volume rendering algorithm [150], the system is able to handle arbitrarily large volumes, but the frame rates are not interactive yet.

Add load balancing. The major disadvantage of a sort-first architecture is the potential for load imbalance among the rendering servers if the geometry clusters on regions of the screen. Mueller [99] and Samanta [116, 117, 118] have developed techniques to resize the screen tiles dynamically. Although these dynamic techniques promote better balancing, they do not improve frame rates necessarily, because they create a bottleneck on the client. We would like to compare these techniques with a simple static approach in which tiles are subdivided into a number of sub-tiles equal to the number of rendering servers.

Extract and publish API. We would like to encapsulate our techniques in a library with an application programming interface that could be reused by other systems. Sources of inspiration come from the Gang of Four [51], VTK [123], and Optimizer [126].

Support dynamic scenes. The system assumes that the dataset is static, i.e., the geometry does not change over time. We would like to extend the system to handle dynamic geometry. The general case in which the whole dataset changes may be too difficult. But the special case of localized changes, such as the ones typical of a CAD modeling package, seems much easier to solve, although still very challenging.

Develop an analytic model for the system. There are many internal and external parameters that affect the performance of the system. Internal parameters include the maximum number of vertices per octree node, the size of the geometry cache, the geometry budget for approximate visibility, and the prefetching limit per frame, to name a few. External parameters include the CPU speed, the triangle throughput of the graphics card, and the bandwidths and latencies of the disks and network connections. A model that predicted the system performance given these parameters would be helpful to guide the configuration and optimization of the system.

Optimize system parameters automatically. If we manage to describe the behavior of the system accurately with an analytical model, the next step would be to implement an optimization procedure to find the best parameters for the system without any programmer intervention.

6.4 Speculation

Today's graphics cards are designed to support the Z-buffer algorithm. A revolutionary change in computer graphics will come when ray tracing becomes supported in commodity graphics cards. The simplicity and power of the ray tracing algorithm are just beautiful. Ray tracing has almost "built-in" occlusion culling and level-of-detail, and produces very photorealistic images. In addition, ray tracing is a member of the class of so-called "embarrassingly parallel" algorithms, because the color of each pixel can be computed completely independently of the color of other pixels. If we had a cluster of one million machines, we could allocate a machine to each pixel of our screen. If graphics hardware continues to advance at the current pace, soon enough a graphics card with one million ray processors will be available on game consoles. The recent vertex program extension to OpenGL is a step in the right direction.

6.5 Getting the Source Code

The iWalk system is open source and is part of the GTB suite of graphics tools [25]. From the GTB sourceforge web site, you can download the source code for iWalk and other systems based on GTB. GTB is licensed under the GNU public license.

Bibliography

[1] E. H. Adelson and J. R. Bergen. The plenoptic function and the elements of early vision. In M. Landy and J. A. Movshon, editors, *Computational Models of Visual Processing*, chapter 1. MIT Press, Cambridge, MA, 1991.

[2] P. Agarwal, L. Arge, O. Procopiuc, and J. Vitter. A framework for index bulk loading and dynamization. In *International Colloquium on Automata, Languages, and Programming*, pages 115–127, 2001.

[3] J. M. Airey, J. H. Rohlf, and F. P. Brooks, Jr. Towards image realism with interactive update rates in complex virtual building environments. *1990 ACM Symposium on Interactive 3D Graphics*, pages 41–50, 1990.

[4] K. Akeley. RealityEngine graphics. In *ACM SIGGRAPH 93*, pages 109–116, 1993.

[5] D. Aliaga, J. Cohen, A. Wilson, H. Zhang, C. Erikson, K. Hoff, T. Hudson, W. Stürzlinger, E. Baker, R. Bastos, M. Whitton, F. Brooks, and D. Manocha. MMR: An interactive massive model rendering system using geometric and image-based acceleration. *1999 ACM Symposium on Interactive 3D Graphics*, pages 199–206, 1999.

[6] D. G. Aliaga and A. Lastra. Automatic image placement to provide a guaranteed frame rate. In *ACM SIGGRAPH 99*, pages 307–316, 1999.

[7] C. Andújar, C. Saona-Vázquez, I. Navazo, and P. Brunet. Integrating occlusion culling and levels of detail through hardly-visible sets. *Computer Graphics Forum*, 19(3):499–506, 2000.

[8] L. Arge, K. Hinrichs, J. Vahrenhold, and J. Vitter. Efficient bulk operations on dynamic R-trees. In *Proceedings of Workshop on Algorithm Engineering*, pages 104–128, 1999.

[9] L. S. Avila and W. Schroeder. Interactive visualization of aircraft and power generation engines. In *IEEE Visualization 97*, pages 483–486, 1997.

[10] M. J. Bach. *The Design of the UNIX Operating System*. Prentice Hall, 1986.

[11] D. Bartz, D. Staneker, W. Straßer, B. Cripe, T. Gaskins, K. Orton, M. Carter, A. Johannsen, and J. Trom. Jupiter: A toolkit for interactive large model visualization. In *2001 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 129–134, 2001.

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

[13] F. Bernardini, I. Martin, J. Mittleman, H. Rushmeier, and G. Taubin. Building a digital model of Michelangelo's Florentine Pietà. *IEEE Computer Graphics & Applications*, 22(1):59–67, 2002.

[14] E. W. Bethel, G. Humphreys, B. Paul, and J. D. Brederson. Sort-first, distributed memory parallel visualization and rendering. In *2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 41–50, 2003.

[15] J. F. Blinn. Light reflection functions for simulation of clouds and dusty surfaces. In *ACM SIGGRAPH 82*, pages 21–29, 1992.

[16] P. Borrel, K. Cheng, P. Darmon, P. Kirchner, J. Lipscomb, J. Menon, J. Mittleman, J. Rossignac, B.-O. Schneider, and B. Wolfe. The IBM 3D interaction accelerator (3DIX). Technical report, IBM, 1995.

[17] T. Bray. The Bonnie file system benchmark. http://www.textuality.com/-bonnie.

[18] I. Buck, G. Humphreys, and P. Hanrahan. Tracking graphics state for networked rendering. *2000 Eurographics Workshop on Graphics Hardware*, pages 87–96, 2000.

[19] D. R. Butenhof. *Programming with POSIX Threads*. Addison Wesley, 1997.

[20] J. Chhugani, B. Purnomo, S. Krishnan, J. Cohen, S. Venkatasubramanian, D. Johnson, and S. Kumar. vLOD: A system for high-fidelity walkthroughs of large virtual environments. Manuscript available at http://www.cs.jhu.edu/-˜jatinch/Research, 2003.

[21] Y.-J. Chiang, C. T. Silva, and W. J. Schroeder. Interactive out-of-core isosurface extraction. *IEEE Visualization 98*, pages 167–174, 1998.

[22] P. Cignoni, C. Rocchini, C. Montani, and R. Scopigno. External memory management and simplification of huge meshes. *IEEE Transactions on Visualization and Computer Graphics*, 9(4):525–537, 2003.

[23] J. H. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, Oct. 1976.

[24] D. Cohen-Or, Y. Chrysanthou, C. T. Silva, and F. Durand. A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):412–431, 2003.

[25] W. T. Corrêa. GTB: The graphics toolbox. http://gtb.sourceforge.net/.

[26] W. T. Corrêa, S. Fleishman, and C. T. Silva. Towards point-based acquisition and rendering of large real-world environments. In *XV Brazilian Symposium on Computer Graphics and Image Processing*, pages 59–66, 2002.

[27] W. T. Corrêa, J. T. Klosowski, and C. T. Silva. Fast and simple occlusion culling. In *Game Programming Gems 3*, pages 353–358. Charles River Media, 2002.

[28] W. T. Corrêa, J. T. Klosowski, and C. T. Silva. iWalk: Interactive out-of-core rendering of large models. Technical Report TR-653-02, Princeton University, 2002.

[29] W. T. Corrêa, J. T. Klosowski, and C. T. Silva. Out-of-core sort-first parallel rendering for cluster-based tiled displays. In *2002 Eurographics Workshop on Parallel Graphics and Visualization*, pages 89–96, 2002.

[30] W. T. Corrêa, J. T. Klosowski, and C. T. Silva. Out-of-core sort-first parallel rendering for cluster-based tiled displays. *Parallel Computing*, 29(3):325–338, Mar. 2003.

[31] W. T. Corrêa, J. T. Klosowski, and C. T. Silva. Visibility-based prefetching for interactive out-of-core rendering. In *2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 1–8, 2003.

[32] M. B. Cox and D. Ellsworth. Application-controlled demand paging for out-of-core visualization. *IEEE Visualization 97*, pages 235–244, 1997.

[33] C. Csuri, R. Hackathorn, R. Parent, W. E. Carlson, and M. Howard. Towards an interactive high visual complexity animation system. In *ACM SIGGRAPH 79*, pages 289–299, 1979.

[34] L. Darsa, B. C. Silva, and A. Varshney. Navigating static environments using image-space simplification and morphing. In *1997 ACM Symposium on Interactive 3D Graphics*, pages 25–34, 1997.

[35] P. Debevec and S. Gortler. Image-based modeling and rendering. In *SIGGRAPH 98 Course Notes*. ACM SIGGRAPH, 1998.

[36] P. Debevec, C. Taylor, and J. Malik. Modeling and rendering architecture from photographs: A hybrid geometry- and image-based approach. In *ACM SIGGRAPH 96*, pages 11–20, 1996.

[37] P. E. Debevec, Y. Yu, and G. D. Borshukov. Efficient view-dependent image-based rendering with projective texture-mapping. *1998 Eurographics Rendering Workshop*, pages 105–116, 1998.

[38] X. Decoret, F. Sillion, G. Schaufler, and J. Dorsey. Multi-layered impostors for accelerated rendering. *Computer Graphics Forum*, 18(3):61–73, 1999.

[39] C. DeCoro and R. Pajarola. XFastMesh: Fast view-dependent meshing from external memory. In *IEEE Visualization 2002*, pages 363–370, 2002.

[40] M. A. Duchaineau, M. Wolinsky, D. E. Sigeti, M. C. Miller, C. Aldrich, and M. B. Mineev-Weinstein. ROAMing terrain: Real-time optimally adapting meshes. In *IEEE Visualization 97*, pages 81–88, 1997.

[41] F. Durand, G. Drettakis, J. Thollot, and C. Puech. Conservative visibility preprocessing using extended projections. In *ACM SIGGRAPH 2000*, pages 239–248, 2000.

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

[43] J. El-Sana, N. Sokolovsky, and C. T. Silva. Integrating occlusion culling with view-dependent rendering. In *IEEE Visualization 2001*, pages 371–378, 2001.

[44] J. El-Sana and A. Varshney. Topology simplification for polygonal virtual environments. *IEEE Transaction on Visualization and Computer Graphics*, 4(2):133–144, 1998.

[45] J. El-Sana and A. Varshney. Generalized view-dependent simplification. In *Eurographics 99*, pages 83–94, 1999.

[46] C. Erikson, D. Manocha, and W. V. Baxter III. HLODs for faster display of large static and dynamic environments. In *2001 ACM Symposium on Interactive 3D Graphics*, pages 111–120, 2001.

[47] H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, and L. Israel. Pixel-planes 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories. In *ACM SIGGRAPH 89*, pages 79–88, 1989.

[48] T. A. Funkhouser. Database management for interactive display of large architectural models. *Graphics Interface 96*, pages 1–8, 1996.

[49] T. A. Funkhouser and C. H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *ACM SIGGRAPH 93*, pages 247–254, 1993.

[50] T. A. Funkhouser, C. H. Séquin, and S. J. Teller. Management of large amounts of data in interactive building walkthroughs. *1992 ACM Symposium on Interactive 3D Graphics*, pages 11–20, 1992.

[51] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[52] M. Garland and P. Heckbert. Surface simplification using quadric error metrics. In *ACM SIGGRAPH 97*, pages 209–216, 1997.

[53] B. Garlick, D. R. Baum, and J. M. Winget. Interactive viewing of large geometric databases using multiprocessor graphics workstations. In *SIGGRAPH 90 Course Notes (Parallel Algorithms and Architectures for 3D Image Generation)*, pages 239–245. ACM SIGGRAPH, 1990.

[54] B. S. Gindele. Buffer block prefetching method. *IBM Technical Disclosure Bulletin*, 20(2):696–697, 1977.

[55] S. J. Gortler, R. Grzeszczuk, R. Szeliski, and M. F. Cohen. The lumigraph. In *ACM SIGGRAPH 96*, pages 43–54, 1996.

[56] N. Greene, M. Kass, and G. Miller. Hierarchical z-buffer visibility. In *ACM SIGGRAPH 93*, pages 231–238, 1993.

[57] J. Grossman and W. J. Dally. Point sample rendering. In *1998 Eurographics Rendering Workshop*, pages 181–192, 1998.

[58] O. Hall-Holt and S. Rusinkiewicz. Visible zone maintenance for real-time occlusion culling. Manuscript, 2003.

[59] A. Heirich and L. Moll. Scalable distributed visualization using off-the-shelf components. *1999 IEEE Symposium on Parallel Visualization and Graphics*, pages 55–59, 1999.

[60] J. Ho, K.-C. Lee, and D. Kriegman. Compressing large polygonal models. In *IEEE Visualization 2001*, pages 357–362, 2001.

[61] H. Hoppe. Progressive meshes. In *ACM SIGGRAPH 96*, pages 99–108, 1996.

[62] H. Hoppe. View-dependent refinement of progressive meshes. In *ACM SIGGRAPH 97*, pages 189–198, 1997.

[63] H. Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *IEEE Visualization 98*, pages 35–42, 1998.

[64] D. Howe. The free on-line dictionary of computing. http://www.foldoc.org.

[65] G. Humphreys, I. Buck, M. Eldridge, and P. Hanrahan. Distributed rendering for scalable displays. In *2000 IEEE Supercomputing*, 2000.

[66] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan. WireGL: A scalable graphics system for clusters. In *ACM SIGGRAPH 2001*, pages 129–140, 2001.

[67] G. Humphreys and P. Hanrahan. A distributed graphics system for large tiled displays. In *IEEE Visualization 99*, pages 215–223, 1999.

[68] G. Humphreys, M. Houston, Y.-R. Ng, R. Frank, S. Ahern, P. Kirchner, and J. T. Klosowski. Chromium: A stream processing framework for interactive graphics on clusters. In *ACM SIGGRAPH 2002*, pages 693–702, 2002.

[69] M. Isenburg and S. Gumhold. Out-of-core compression for gigantic polygon meshes. In *ACM SIGGRAPH 2003*, pages 935–942, 2003.

[70] W. H. Jiménez, W. T. Corrêa, A. Baptista, and C. Silva. Visualizing spatial and temporal variability in coastal observatories. In *IEEE Visualization 2003*, 2003.

[71] T. L. Kay and J. T. Kajiya. Ray tracing complex scenes. In *ACM SIGGRAPH 86*, pages 269–278, 1986.

[72] A. W. Klein, W. Li, M. M. Kazhdan, W. T. Corrêa, A. Finkelstein, and T. A. Funkhouser. Non-photorealistic virtual environments. In *ACM SIGGRAPH 2000*, pages 527–534, 2000.

[73] J. T. Klosowski and C. T. Silva. The prioritized-layered projection algorithm for visible set estimation. *IEEE Transactions on Visualization and Computer Graphics*, 6(2):108–123, Apr. 2000.

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

[75] T. Kurc, U. Catalyurek, C. Chang, A. Sussman, and J. Saltz. Exploration and visualization of very large datasets with the active data repository. *IEEE Computer Graphics & Applications*, 21(4):24–33, 2001.

[76] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, 1990.

[77] M. Levoy and P. Hanrahan. Light field rendering. In *ACM SIGGRAPH 96*, pages 31–42, 1996.

[78] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, and D. Fulk. The digital Michelangelo project: 3D scanning of large statues. In *ACM SIGGRAPH 2000*, pages 131–144, 2000.

[79] M. Levoy and T. Whitted. The use of points as a display primitive. Technical Report TR 85-022, The University of North Carolina at Chapel Hill, Department of Computer Science, 1985.

[80] K. Li, H. Chen, Y. Chen, D. W. Clark, P. Cook, S. Damianakis, G. Essl, A. Finkelstein, T. Funkhouser, T. Housel, A. Klein, Z. Liu, E. Praun, R. Samanta, B. Shedd, J. P. Singh, G. Tzanetakis, and J. Zheng. Early experiences and challenges in building and using a scalable display wall system. *IEEE Computer Graphics and Applications*, 25(4):671–680, 2000.

[81] P. Lindstrom. Out-of-core simplification of large polygonal models. In *ACM SIGGRAPH 2000*, pages 259–262, 2000.

[82] P. Lindstrom. Out-of-core construction and visualization of multiresolution surfaces. In *2003 ACM SIGGRAPH Symposium on Interactive 3D Graphics*, pages 93–102,239, 2003.

[83] P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hughes, N. Faust, and G. Turner. Real-time, continuous level of detail rendering of height fields. In *ACM SIGGRAPH 96*, pages 109–118, 1996.

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

[85] S. Lombeyda, L. Moll, M. Shand, D. Breen, and A. Heirich. Scalable interactive volume rendering using off-the-shelf components. In *2001 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 115–121, 2001.

[86] D. Luebke and C. Erikson. View-dependent simplification of arbitrary polygonal environments. In *ACM SIGGRAPH 97*, pages 199–208, 1997.

[87] D. Luebke, M. Reddy, J. D. Cohen, A. Varshney, B. Watson, and R. Huebner. *Level of Detail for 3D Graphics.* Morgan Kaufmann, 2002.

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

[89] MathSoft, Data Analysis Products Division. *S-Plus 5 for UNIX User's Guide*, 1998.

[90] S. McMains, J. Hellerstein, and C. Sequin. Out-of-core build of a topological data structure from polygon soup. In *Proceedings of Solid Modeling 2001*, pages 171–182, 2001.

[91] L. McMillan and G. Bishop. Plenoptic modeling: An image-based rendering system. In *ACM SIGGRAPH 95*, pages 39–46, 1995.

[92] L. Moll, A. Heirich, and M. Shand. Sepia: scalable 3D compositing using PCI Pamette. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 146–155, 1999.

[93] T. Möller and E. Haines. *Real-Time Rendering.* A K Peters, 1999.

[94] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32, 1994.

[95] S. Molnar, J. Eyles, and J. Poulton. Pixelflow: High-speed rendering using image composition. In *ACM SIGGRAPH 92*, pages 231–240, 1992.

[96] J. S. Montrym, D. R. Baum, D. L. Dignam, and C. J. Migdal. InfiniteReality: A real-time graphics system. In *ACM SIGGRAPH 97*, pages 293–302, 1997.

[97] K. Moreland and R. Frank. The Lawrence Livermore National Laboratory (LLNL) isosurface dataset. Personal communication. 473 million triangles.

[98] K. Moreland, B. Wylie, and C. Pavlakos. Sort-last parallel rendering for viewing extremely large data sets on tile displays. In *2001 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 85–92, 2001.

[99] C. Mueller. The sort-first rendering architecture for high-performance graphics. *1995 ACM Symposium on Interactive 3D Graphics*, pages 75–84, 1995.

[100] C. Mueller. Hierarchical graphics databases in sort-first. In *1997 IEEE Symposium on Parallel Rendering*, pages 49–58, 1997.

[101] J. Noble and D. B. Charles Weir and. *Small Memory Software: Patterns for Systems with Limited Memory.* Addison-Wesley, 2000.

[102] nVIDIA. The nVIDIA FX 3000G. http://www.nvidia.com/object/vis.html, 2003.

[103] OpenGL.org. OpenGL FAQ: Display lists and vertex arrays. http://-www.opengl.org/developers/faqs/technical/displaylist.htm.

[104] H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels: Surface elements as rendering primitives. In *ACM SIGGRAPH 2000*, pages 335–342, 2000.

[105] M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan. Rendering complex scenes with memory-coherent ray tracing. In *ACM SIGGRAPH 97*, pages 101–108, 1997.

[106] S. Popinet. GTS: The GNU triangulated surface library. Available at http://gts.sourceforge.net/.

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

[108] S. A. Przybylski. *Cache and Memory Hierarchy Design: A Performance-Directed Approach.* Morgan Kaufmann, 1990.

[109] W. T. Reeves. Particle systems — a technique for modeling a class of fuzzy objects. *ACM Transactions on Graphics*, 2(2):91–108, Apr. 1983.

[110] A. Rege. Occlusion extensions. http://developer.nvidia.com/docs/IO/2645/-ATT/GDC2002_occlusion.pdf, 2002.

[111] D. Reiners, G. Voß, J. Behr, M. Roth, and A. Zieringer. The OpenSG scene graph system. http://www.opensg.org/, 1999.

[112] J. Rossignac and P. Borrel. Multi-resolution 3D approximations for rendering complex scenes. In *Geometric Modeling in Computer Graphics*, pages 455–465. Springer Verlag, 1993.

[113] S. Rusinkiewicz and M. Levoy. QSplat: A multiresolution point rendering system for large meshes. In *ACM SIGGRAPH 2000*, pages 343–352, 2000.

[114] S. Rusinkiewicz and M. Levoy. Streaming QSplat: A viewer for networked visualization of large, dense models. In *2001 ACM Symposium on Interactive 3D Graphics*, 2001.

[115] R. Samanta, T. Funkhouser, and K. Li. Parallel rendering with k-way replication. In *2001 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 75–84, 2001.

[116] R. Samanta, T. Funkhouser, K. Li, and J. P. Singh. Hybrid sort-first and sort-last parallel rendering with a cluster of PCs. In *2000 Eurographics Workshop on Graphics Hardware*, pages 97–108, 2000.

[117] R. Samanta, T. Funkhouser, K. Li, and J. P. Singh. Sort-first parallel rendering with a cluster of PCs. In *Sketches and Applications, SIGGRAPH 2000*, page 260, 2000.

[118] R. Samanta, J. Zheng, T. Funkhouser, K. Li, and J. P. Singh. Load balancing for multi-projector rendering systems. In *1999 Eurographics Workshop on Graphics Hardware*, pages 107–116, 1999.

[119] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[120] G. Schaufler, J. Dorsey, X. Decoret, and F. X. Sillion. Conservative volumetric visibility with occluder fusion. In *ACM SIGGRAPH 2000*, pages 229–238, 2000.

[121] G. Schaufler and H. W. Jensen. Ray tracing point sampled geometry. In *2000 Eurographics Rendering Workshop*, pages 319–328, 2000.

[122] B.-O. Schneider, P. Borrel, J. Menon, J. Mittleman, and J. Rossignac. BRUSH as a walkthrough system for architectural models. In *1995 Eurographics Rendering Workshop*, pages 389–399, 1995.

[123] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics.* Prentic Hall, 1996.

[124] K. Severson. VISUALIZE workstation graphics for Windows NT. HP product literature, 1999.

[125] SGI. OpenGL Optimizer Programmer's Guide: An Open API for Large-Model Visualization, 1998. Available online.

[126] SGI. OpenGL optimizer. http://www.sgi.com/software/optimizer, 2003.

[127] SGI. OpenGL performer. http://www.sgi.com/software/performer, 2003.

[128] SGI. The Silicon Graphics Onyx4 UltimateVision. http://www.sgi.com/-visualization/onyx4, 2003.

[129] J. Shade, S. Gortler, L. He, and R. Szeliski. Layered depth images. In *ACM SIGGRAPH 98*, pages 231–242, 1998.

[130] J. Shade, D. Lischinski, D. H. Salesin, T. DeRose, and J. Snyder. Hierarchical image caching for accelerated walkthroughs of complex environments. In *ACM SIGGRAPH 96*, pages 75–82, 1996.

[131] H.-Y. Shum and L.-W. He. Rendering with concentric mosaics. In *ACM SIGGRAPH 99*, pages 299–306, 1999.

[132] F. Sillion, G. Drettakis, and B. Bodelet. Efficient impostor manipulation for real-time visualization of urban scenery. *Computer Graphics Forum*, 16:207–218, 1997.

[133] A. R. Smith. Plants, fractals and formal languages. In *ACM SIGGRAPH 84*, pages 1–10, 1984.

[134] I. Software. Quake 3. http://www.idsoftware.com/games/quake/, 2003.

[135] G. Stoll, M. Eldridge, D. Patterson, A. Webb, S. Berman, R. Levy, C. Caywood, M. Taveira, S. Hunt, and P. Hanrahan. Lightning-2: A high-performance display subsystem for pc clusters. In *ACM SIGGRAPH 2001*, pages 141–148, 2001.

[136] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker. A characterization of ten hidden-surface algorithms. *ACM Computing Surveys*, 6(1):1–55, 1974.

[137] A. S. Tanenbaum and A. S. Woodhull. *Operating Systems: Design and Implementation*. Prentice Hall, 2nd edition, 1997.

[138] S. J. Teller and C. H. Séquin. Visibility preprocessing for interactive walkthroughs. In *ACM SIGGRAPH 91*, pages 61–69, 1991.

[139] TGS. Open inventor 4.0. http://www.tgs.com/pro_div/oiv_main.htm, 2003.

[140] W. C. Thibault and B. F. Naylor. Set operations on polyhedra using binary space partitioning trees. In *ACM SIGGRAPH 87*, pages 153–162, 1987.

[141] S.-K. Ueng, C. Sikorski, and K.-L. Ma. Out-of-core streamline visualization on large unstructured meshes. *IEEE Transactions on Visualization and Computer Graphics*, 3(4):370–380, Oct. 1997.

[142] M. K. Ullner. *Parallel machines for computer graphics.* PhD thesis, California Institute of Technology, Pasadena, California, 1983.

[143] G. Varadhan and D. Manocha. Out-of-core rendering of massive geometrci environments. In *IEEE Visualization 2002*, pages 69–76, 2002.

[144] G. Voß, J. Behr, D. Reiners, and M. Roth. A multi-thread safe foundation for scene graphs and its extension to clusters. In *2002 Eurographics Workshop on Parallel Graphics and Visualization*, pages 33–37, 2002.

[145] I. Wald, C. Benthin, and P. Slusallek. Distributed interactive ray tracing of dynamic scenes. In *2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 77–86, 2003.

[146] I. Wald, P. Slusallek, and C. Benthin. Interactive distributed ray tracing of highly complex models. *Rendering Techniques 2001*, pages 277–288, 2001.

[147] Walkthru Project at UNC Chapel Hill. The power plant model. http://-www.cs.unc.edu/˜geom/Powerplant.

[148] B. Wei, D. W. Clark, E. W. Felten, K. Li, and G. Stoll. Performance issues of a distributed frame buffer on a multicomputer. *1998 Eurographics Workshop on Graphics Hardware*, pages 87–96, 1998.

[149] L. Williams. Pyramidal parametrics. In *ACM SIGGRAPH 83*, pages 1–11, 1983.

[150] P. L. Williams. Visibility-ordering meshed polyhedra. *ACM Transactions on Graphics*, 11(2):103–126, 1992.

[151] P. Wilson, S. Kaplan, and Y. Smaragdakis. The case for compressed caching in virtual memory systems. In *Proceedings of the USENIX Annual Technical Conference*, pages 6–11, 1999.

[152] P. Wonka, M. Wimmer, and D. Schmalstieg. Visibility preprocessing with occluder fusion for urban walkthroughs. In *Rendering Techniques 2000*, pages 71–82, 2000.

[153] P. Wonka, M. Wimmer, and F. Sillion. Instant visibility. *Computer Graphics Forum*, 20(3):411–421, 2001.

[154] M. Woo, J. Neider, T. Davis, and D. Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley, 3rd edition edition, 1999.

[155] J. Xia, J. El-Sana, and A. Varshney. Adaptive real-time level-of-detail-based rendering for polygonal models. *IEEE Transaction on Visualization and Computer Graphics*, 3(2):171–183, 1997.

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

[157] X. Zhang, C. Bajaj, W. Blanke, and D. Fussell. Scalable isosurface visualization of massive datasets on COTS clusters. In *2001 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 51–58, 2001.