Parallel Real-Time Rendering using Heterogeneous PC Clusters

Tim Süß

Zusammenfassung

3-D-Szenen aus CAD-Systemen besitzen zumeist eine hohe geometrische Komplexität. Es gibt eine Reihe von Konzepten, welche sich mit der Schwierigkeit auseinandersetzen, solche Szenen in Echtzeit zu rendern; beispielsweise daten- und rechenparallele Techniken oder Out-of-Core Rendering Mechanismen. Diese Dissertationsschrift behandelt die Nutzung von heterogenen PC-Clustern zur parallelen Bildberechnung von hochkomplexen Szenen. Für drei unterschiedliche Szenarien wurden jeweils verschiedene Verfahren entwickelt bei deren Berechnungen wenige Highend-Rechner von vielen schwachen Rechnern unterstützt werden. Zunächst sind dies statische Szenen, die vollständig in den Hauptspeicher eines Rechners geladen werden können. Im zweiten Szenario werden statische Szenen betrachtet, deren Komplexität den Speicher eines einzelnen Rechners übersteigt. Zuletzt werden Szenen betrachtet, die neben statischen Objekten auch dynamische Objekte beinhalten.

Abstract

Often 3D scenes created with CAD applications have a high geometric complexity. There are several concepts (like out-of-core rendering, levels of detail, parallel rendering) to render these complex scenes in real-time. This dissertation focuses on the usage of heterogeneous PC clusters for parallel real-time rendering of highly complex scenes. For three different scene types specific rendering approaches were developed, where a small group of high-end computers is supported by a large number of weaker PC cluster nodes. The first scene type consists of static scenes that can be stored completely in a single computer's main memory, while the scenes of the second type exceed this memory limitations. The scenes of the last type contain not only static but also dynamic objects.

HEINZ NIXDORF INSTITUT
Universität Paderborn

Parallel Real-Time Rendering using Heterogeneous PC Clusters

Dissertation

by

Tim Süß

Heinz Nixdorf Institute and Department of Computer Science
University of Paderborn
November 2011

Reviewers:

- Prof. Dr. Friedhelm Meyer auf der Heide, University of Paderborn
- Prof. Dr. André Brinkmann, University of Mainz

For Anja and Noel

Contents

1 Introduction

Complex polygonal 3D models may consist of hundreds of millions of triangles and require multiple gigabytes of memory. Rendering such *Massive Models* in real-time is one of the most challenging problems in modern computer graphics [KBF05]. A user should be able to navigate through a scene or models interactively while at least six to ten frames per second are computed. The parallelization of the rendering process is a common approach to face this problem [KDG+08]. Many real-time rendering algorithms can improve performance by distributing the load among multiple computers. For each frame that is displayed an image for the current camera position must be rendered. To produce images of polygonal 3D models, usually their geometric primitives are sent through a rendering pipeline (see Figure 1.1), where they are transformed into pixels [AMHH08]. The parallelization of such real-time, *pipeline rendering* algorithms is done rarely because PC clusters completely equipped with modern graphic adapters are still rare. Usually, PC clusters are intended to be used for other applications, such as scientific computations. For example, the PADS and TeraPort PC clusters at the University of Chicago do not provide any OpenGL accelerating hardware nor does the JUGENE supercomputer of the Jülich Supercomputing Center. Some systems are equipped with weak graphics adapters, like the Paderborn Center of Parallel Computing's (PC^2) Arminius PC cluster. On the other hand, many compute centers offer a *small* group of so-called *visualization nodes* that are equipped with high end graphics hardware. The performance improvements and the GPGPU programmability of modern graphic adapters make these components more and more valuable for PC clusters. However, this kind of PC clusters can hardly be used for standard parallel pipeline rendering techniques as proposed by Molnar et al. [MCEF94, MCEF08]. Typically, the performance of these methods depend on the slowest node. Due to these reasons, we put the focus of this thesis on the development of new parallel pipeline rendering algorithms for such heterogeneous PC clusters. We require that these heterogeneous systems include a small group of powerful visualization nodes and a large group of weaker *back-end nodes*. While the visualization nodes should be equipped with high end graphics adapters, the back-end nodes require only weak graphics performance. The back-end nodes should be equipped with common hardware and a network must connect the different nodes.

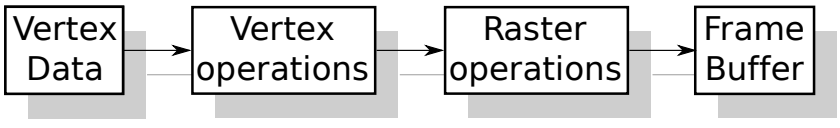


Figure 1.1: Simplified representation of a rendering pipeline.

The objective is to render complex 3D scenes in real-time using such heterogeneous environments. Thus, the following questions are addressed in this thesis:

- How can we utilize heterogeneous PC clusters for parallel pipeline rendering?
- How can the powerful, high end nodes benefit from the additional computational power of the weaker nodes?
- How can we cope with or even exploit asynchronous communication?
- How can the load be distributed fairly among the different nodes?

Different scene classes require different rendering algorithms. Many *levels of detail* systems require objects (for example chairs, balls or even planes) that consist of connected components (often, these methods are unsuitable for triangles in arbitrary order, usually referred to as *polygon-soups*) [LWC+02], and the rendering algorithm introduced by Chamberlain et al. is best applied on spacious scenes that provide an even distribution of their geometry [CDL+96]. Several *occlusion culling* systems are only suitable for scenes consisting of architectural models [AMHH08]. These systems discard objects, which are covered by other objects. Scenes that exceed primary memory (in our case the main and video memory) often must be processed differently than scenes that fit within it. Real-time rendering is challenging even for large static scenes; including dynamic objects further increases the difficulty.

We developed different pipeline rendering algorithms and data structures for three different scenarios. Each of our developed parallel rendering algorithms is suitable for one of these. Below we denote pipeline rendering simply by "rendering", unless otherwise stated.

Scenario I Static scenes (scene's objects never change their position) that fit into the primary memory of a visualization node and have sparse occlusion (objects are rarely covered completely). The challenge of rendering such scenes is the large number of objects participating in an image, because it is rare that objects occlude others completely.

Scenario II Static scenes that do not fit into the primary memory of a visualization node. Rendering images of such scenes with a suitable frame rate requires fast access to data items which are not stored in the primary memory and a fast detection of the visible items.

Scenario III Large dynamic scenes (scene's objects can change their position). In such scenes objects' visibility can change constantly. Furthermore, additional computations are necessary for the moving objects, which influence the rendering speed.

We developed several data structures, rendering algorithms, and object simplifications to render such scenes on heterogeneous PC clusters. We also tested different hardware configurations to accelerate the parallel rendering process of dynamic scenes. Below we will describe briefly our approaches for each scenario. Additionally, we state the issues, which must be addressed to realize these approaches.

Method for Scenario I

For the first scenario we introduce *reliefboards* [SJF10]. A reliefboard is an approximation of a complex scene object. These approximations are generated asynchronously on the weak back-end nodes while the displayed images are rendered on a single powerful visualization node. Since the reliefboard's computation takes several seconds on the weak back-end nodes, the reliefboard must be suitable for multiple camera positions and for many frames. Reliefboards appear like a scene's original objects for many camera positions, and provide other positive properties. If the image errors that occur through camera movements exceed a threshold, the reliefboard must be exchanged with a new one. In this scenario the following problems have been solved:

1. Computation of simplifications that are suitable for many camera positions.
2. Determination of objects that can be replaced by reliefboards.
3. Even distribution of the computational load.
4. Reduction of the payload that is sent across the network.

To 1: Reliefboards make it possible to render complex scenes with high frame rates with acceptable image errors.
To 2: We employed a clustering algorithm to determine objects which are suitable for replacement by reliefboards.
To 3: For the load balancing we modified a data management protocol, the so-called c-collision protocol, for our asynchronous communication scheme.
To 4: Reliefboards have little memory requirements in comparison to the original geometry that leads to small messages sent across the network.

Furthermore, using our developed object simplifications reduces the image noise and frame rate's fluctuations during the walk-through. In contrast to other parallel rendering systems, our approach scales in the image quality rather than scaling in the frame rate if the number of back-end nodes is increased. Our rendering system do not wait until the back-end nodes have finished their tasks. If the number of back-end nodes is increased, we can decrease the number of approximations each of these nodes has to process or the number of objects it has to test. Thus, we reduce the time spent between the initiation and response of a job, whereby the required results of these jobs arrive earlier.

Methods for Scenario II

For the second scenario we introduce two different parallel out-of-core rendering systems [SWF10a, SWF10b, SKJ+11] and one sequential rendering technique [SKJF11]. The rendering method of all rendering systems is approximative, which leads to pixel errors in the final image. In the parallel out-of-core systems, the weak back-end nodes serve as secondary memory of the visualization nodes. The complete scene is distributed among these weak nodes and stored in their primary memory, to allow for fast data access. When scene objects are requested, the back-end nodes test the visibility of these objects instead of sending them blindly.

The parallel out-of-core rendering system that we will introduce first uses a version of the *c*-load-collision protocol to balance rendering load and nodes' contention. The back-end nodes perform visibility tests, while they have only access to a subset of all objects and global, but aged, distance information of the other objects. Due to the aged distance information the back-end nodes cannot guarantee to determine all visible objects until the information are updated. The positive tested objects are sent to the visualization node, where they are rendered and displayed.

The sequential rendering system and its data structure is the groundwork for the second parallel out-of-core render system. Our developed *hull tree* is a spatial, hierarchical data structure. It covers scene's objects more tightly than other commonly used data structures. Additionally, we store for each object an approximation to improve and accelerate the visibility test. Our associated approximative rendering algorithm exploits this structure.

The second parallel out-of-core rendering system combines the hull tree with another spatial data structure, the so-called *randomized sample tree*, to improve its properties for parallel rendering. Each back-end node stores a small subset of the original objects and approximations for the other objects. The back-end nodes perform visibility tests with this mix of originals and approximations which are organized in a hull tree.

In these parallel rendering systems the following challenges arise:

1. Distribution of the objects to achieve fast and balanced data access.

2. Providing information to perform suitable visibility tests.

3. Even distribution of the computational load and the data load.

4. Reducing the number of objects sent across the network.

To 1: Due to a randomized distribution we achieve good load balancing.
To 2: We achieve suitable visibility tests if we use global, but aged, distance information of scene's objects. The hull tree in combination with the used approximations reduces the complexity of the visibility tests.
To 3: The data management protocol of our first parallel out-of-core rendering system achieves a good balancing of the load. Due to the hull tree's combination with a randomized sample tree, sending the visible objects across the network is distributed among multiple frames.
To 4: Testing objects visibility a priori on the back-end nodes instead sending them blindly reduces the network load. Thus, we could reduce the network requirements for the second parallel out-of-core rendering system.

Like the reliefboard approach the parallel out-of-core rendering systems scales in the image quality if the number of back-end nodes is increased. Furthermore, an increased number of back-end nodes reduces the delay between a request initiation and its answer.

Method to Scenario III

In the third scenario the focus is put on large dynamic scenes [DFH+08, SFH+08, SFH+09]. In this scenario we separate the dynamic and static objects. In this way it is possible to apply the techniques for static scenes and process the dynamic objects separately. The computations of objects' movements and their interactions is mostly isolated from the rendering of the static parts. We render the different parts on different nodes, and combine the resulting images into a final image. Combining the images is a time consuming operation. For this reason we searched for alternative techniques to accelerate the conflation. Here, hardware and software based solutions were developed [SSPP09, SSPP11]. For our developed systems for dynamic scenes the following questions must be answered:

1. How do we combine the different images?
2. What kind of hardware can be used to accelerate the combining of the different images?

To 1: The developed rendering approach allows to visualize multiple dynamic scenes simultaneously in only one displaying window. We combined the images of the static and the different dynamic scene parts by comparing the depth values of the different corresponding pixels.
To 2: To accelerate this merging process we tested different techniques. We utilized different CPU features as well as GPU features and an FPGA.

Structure of this thesis

In this thesis we focus on the utilization of PC clusters that consist of few visualization nodes equipped with high performance hardware and a large amount of back-end nodes with poor graphics performance for parallel pipeline rendering.

In Chapter 2, we introduce the relevance of this work's topic followed by an overview of work related to this thesis. In Chapter 3, we will introduce preliminaries and definitions used in this work. Afterwards we will present our different rendering algorithms as well as techniques for load balancing, data management, and visibility testing for heterogeneous PC clusters. These techniques provide object-approximations, data structures, and different algorithms to accelerate the rendering of massive models.

Chapter 4 introduces the reliefboard technique where an object-approximation is created on-the-fly on the weak back-end-nodes. The technique presented in Chapter 5 handles the usability of an adjusted data management protocol for parallel out-of-core rendering. To accelerate visibility tests, we developed a culling technique using the hull tree, presented in Chapter 6. We integrated this technique with its associated data structure in a parallel out-of-core rendering system in Chapter 7. For dynamic scenes we evaluated, for one technique, the benefit of FPGAs for parallel rendering in heterogeneous PC clusters, in Chapter 8. The thesis ends with a conclusion and an outlook to future work.

Personal Publications

[DFH+08] Wilhelm Dangelmaier, Matthias Fischer, Daniel Huber, Christoph Laroque, and Tim Süß. Aggregated 3d-visualization of a distributed simulation experiment of a queuing system. In S. J. Mason, R. Hill, L. Moench, and O. Rose, editors, *Proceedings of the Winter Simulation Conference*, WSC´ 08, pages 2012 – 2020. IEEE, Omnipress, 2008.

[DS10] Dominic Dumrauf and Tim Süß. On the complexity of local search for weighted standard set problems. In *Proceedings of the 6th Conference on Computability in Europe*, pages 132–140, 30 June - 4 July 2010.

[SFH+08] Tim Süß, Matthias Fischer, Daniel Huber, Christoph Laroque, and Wilhelm Dangelmaier. A system for aggregated visualization of multiple parallel discrete event simulations. In *Proceedings of the International Symposium on Advances in Parallel and Distributed Computing Techniques*, APDCT '08, pages 587–593. IEEE, IEEE Computer Society Press, December 2008.

[SFH+09] Tim Süß, Matthias Fischer, Daniel Huber, Christoph Laroque, and Wilhelm Dangelmaier. Ein System zur aggregierten Visualisierung verteilter Materialflusssimulationen. In Jürgen Gausemeier and Michael Grafe, editors, *Augmented & Virtual Reality in der Produktentstehung*, volume 252, pages 111–126. Heinz Nixdorf Institut, Universität Paderborn, May 2009.

[SJF10] Tim Süß, Claudius Jähn, and Matthias Fischer. Asynchronous parallel reliefboard computation for scene object approximation. In *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization*, EGPGV '10, pages 43–51, Norrköping, Sweden, May 2010. Eurographics Association, Eurographics Association.

[SKJ+11] Tim Süß, Clemens Koch, Claudius Jähn, Matthias Fischer, and Friedhelm Meyer auf der Heide. Ein paralleles Out-of-Core Renderingsystem für Standard-Rechnernetze. In Jürgen Gausemeier, Michael Grafe, and Friedhelm Meyer auf der Heide, editors, *Augmented & Virtual Reality in der Produktentstehung*, volume 295 of *HNI-Verlagsschriftenreihe, Paderborn*, pages 185–197. Heinz Nixdorf Institut, Universität Paderborn, May 2011.

[SKJF11] Tim Süß, Clemens Koch, Claudius Jähn, and Matthias Fischer. Approximative occlusion culling using the hull tree. In *Proceedings of the Graphics Interface 2011*, pages 79–86. Canadian Human-Computer Communications Society, May 2011.

[SSPP09] Tobias Schumacher, Tim Süß, Christian Plessl, and Marco Platzner. Communication performance characterization for reconfigurable accelerator design on the XD1000. In *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 9 - 11 December 2009.

[SSPP11] Tobias Schumacher, Tim Süß, Christian Plessl, and Marco Platzner. Fpga acceleration of communication-bound streaming applications: Architecture modeling and a 3d image compositing case study. *International Journal of Reconfigurable Computing*, 2011:1–11, 2011. Article ID 760954.

[SWF10a] Tim Süß, Timo Wiesemann, and Matthias Fischer. Evaluation of a c-load-collision-protocol for load-balancing in interactive environments. In *Proceedings of the 5th IEEE International Conference on Networking, Architecture, and Storage*, pages 448 – 456. IEEE Computer Society, IEEE Press, 15 - 17 July 2010.

[SWF10b] Tim Süß, Timo Wiesemann, and Matthias Fischer. Gewichtetes c-Collision-Protokoll zur Balancierung eines parallelen Out-of-Core-Renderingsystems. In Jürgen Gausemeier and Michael Grafe, editors, *Augmented & Virtual Reality in der Produktentstehung*, HNI-Verlagsschriftenreihe, Paderborn, pages 39–52. Universität Paderborn, HNI Verlagsschriftenreihe, Paderborn, 2010.

2 Rendering

Computer graphics are becoming more and more important in various areas of our life. Architects, designers, physicians and many more gain information from the ability to create realistic images of their objects of interest. One area in the huge field of computer-graphics is the real-time rendering of three-dimensional scenes: the aim of 3D real-time rendering is to produce images of a given scene with at least six frames per second [AMHH08]. Thus, a user is able to navigate through the scene without losing his orientation. Usually these scenes are composed of two-dimensional polygons. The process of transforming these polygon data sets into 2D raster images is called rendering. At *pipeline rendering* this process is supported by specialized hardware of common graphics adapters (typically programmed with *OpenGL* or *DirectX*^{TM}). In order to improve the realistic appearance of images produced by such scenes, often the scene objects' polygon count is increased. Thus the virtual objects become more detailed. Incrementing the polygon count also results in more precise object surfaces which allows for more accurate measurements. This is highly relevant for simulations of industrial complexes, production environments, machinery, etc. where users want to check, for example, whether one gear fits to another or if they can be replaced without removing any other parts of the machinery. To increase realism even more, the different objects and the scene itself include several attributes like materials, shaders, textures, or light sources. There is a great need to render scenes that consist of hundreds of millions or more triangles [KDG^{+}08], usually qualified as massive models, in real-time. If the geometry in such scenes is never modified, they are referred to as *static scenes*. Otherwise they are called *dynamic scenes*. The complexity of such massive models can increase so much that a single computer is unable to render them in real-time, even if the scenes are static. When a scene contains dynamic objects, which change their position, orientation or appearance during runtime, the problem becomes even worse.

2.1 PC Clusters for Parallel Rendering

A common approach to computing images of such massive models is to use PC clusters, distributing the rendering load among multiple computers. PC clusters offer a lot of computational power and memory, and in many cases fast networks to exchange data between different nodes. If the PC cluster nodes are equipped with graphics adapters, it is possible to distribute the rendering among these cards, instead using only one. Here for example, each graphics adapter could render a different subimage which are finally combined to a complete one [SFL01, SFLS00, MCEF08]. However, the usage of many PCs instead of only one for rendering introduces new problems. For example, to achieve

good performance the computational load should be distributed evenly among the cluster nodes. This introduces a need for load balancing algorithms. Another problem passed by the parallelization of the rendering process is the data organization. Fast data access and a good data distribution is necessary to render massive models. If the data amount is so large that it does not fit into the *primary memory* of a single PC, data access or data distribution methods are also required. In this thesis the primary memory of a computer refers to its processor memory, its main memory, and its video memory. Slower storage elements used (for example, local hard disks, network attached storage systems) are referred to as secondary memory.

Apart from algorithmic problems, hardware related challenges also occur. While upgrading a single computer can be done with little effort, the renewal of a whole PC cluster rises bigger problems. Beside the acquisition costs for the computer hardware, other costs arise. In many cases, advanced hardware consumes more power which leads to the requirement of a better power supply. The increasing power requirements usually result in a higher heat production, which, in turn, requires an improved cooling system. This then leads to an even higher power consumption. PC clusters' energy consumption and temperature control challenge their providers: For example *Green-IT* refers to the concept of reducing compute centers' energy consumption, while increasing their computational power. Another problem when upgrading PC clusters is hardware's short improvement circles. Usually, the influences of new components on the rest of the system cannot be tested sufficiently. So the upgrade of single hardware components of all PC cluster nodes can lead to malfunctions and the instability of the entire system. One method to combat these problems and to raise computational power is to allow heterogeneous PC clusters, where different nodes are equipped with different hardware components. The nodes within such systems are equipped with different hardware, which is specialized for different tasks. While upgrading an entire PC cluster with state of the art components is a big challenge, a few cluster nodes can be upgraded with significantly less effort. The different components of these nodes (CPUs, memory, graphics cards, etc.) can be replaced in shorter periods of time. While the configuration of most cluster nodes is unmodified, a minor group of dedicated nodes can be kept up-to-date. If these nodes are organized beside the main system, their influence on the stability of the main system is marginal. These nodes can be equipped with *high end* hardware that offers all features that a user or developer would like to use.

2.2 Overview of Related Work

This section introduces work related to the algorithms presented in this thesis. Here, concepts and techniques are introduced that build the basics for all parts of this thesis. We will give a brief overview of parallel rendering in §2.2.1 and out-of-core rendering in §2.2.2. Additionally, we will introduce different techniques for occlusion culling in §2.2.3 and mesh replacement in §2.2.4 that are related to methods we have developed for our rendering methods.

2.2.1 Parallel Rendering

Parallel rendering has been categorized into three basic approaches by Molnar et al. [MCEF94, MCEF08]: sort-first, sort-middle, and sort-last. These parallelization approaches specify in which stage of the rendering pipeline the geometric primitives are distributed among PC cluster nodes (see Figure 1.1).

Sort-first

In the sort-first approach, the display is subdivided into multiple *tiles*. Usually the geometric primitives in each of these tiles are rendered by a different processor. Before rendering, it is estimated in which tile each primitive must be displayed. In other words, the primitives are assigned/sorted to the different nodes before they are sent into the rendering pipeline. When each node has finalized its partial image, the different tiles are composed to the final image (see Figure 2.1, left).

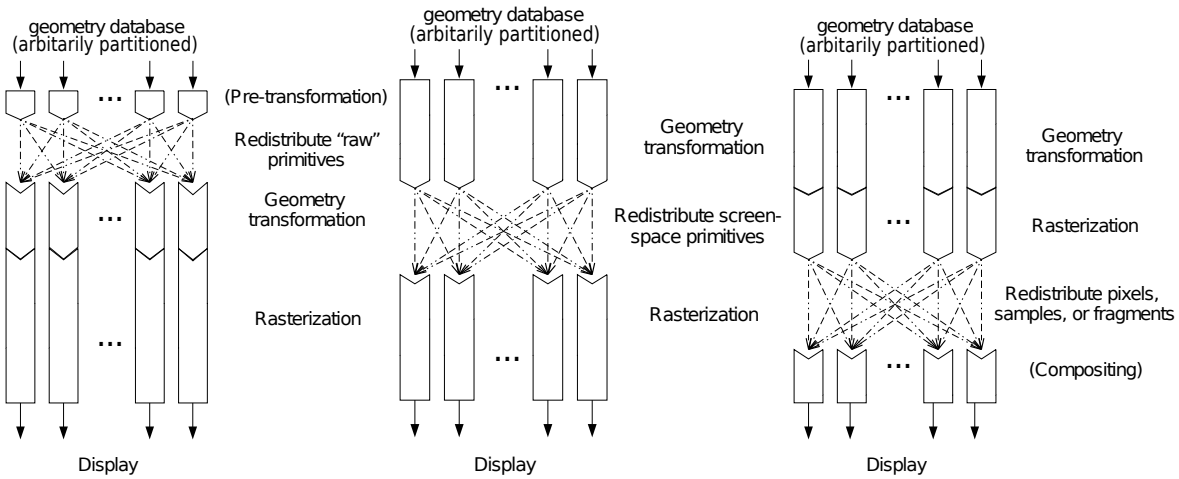


Figure 2.1: In sort-first rendering the geometric primitives are redistributed during the geometry transformation. In sort-middle, the screen-space elements are redistributed in between geometry transformation and rasterization. In sort-last, the different picture elements are redistributed after rasterization and composed afterwards.

The drawback of this method is that usually the different primitives cannot be assigned to the rendering nodes uniquely. Primitives can overlap into multiple tiles, which influence the scalability of that approach negatively.

Sort-middle

In sort-middle, the primitives' arrangement is made in the middle of the rendering pipeline, in between geometry transformation and rasterization. Similar to the sort-first approach, the display is subdivided into tiles. Additionally, the geometric primitives are distributed arbitrarily exclusively to the nodes in a preprocessing step. This distribution

does not change during runtime. Each node performs the geometry transformation for its assigned geometric primitives and determines its corresponding tiles. The transformed display coordinates are transferred to the responsible nodes, where the primitives' rasterization are performed. When all the primitives have been transformed, the nodes' partial images are composed to a final image (see Figure 2.1, middle).

In the sort-middle approach the primitives must be assigned to the different rendering nodes while passing the rendering pipeline. Typical common computer hardware does not provide an interface for interfering with the rendering process at this stage, without causing high delays. Thus, this parallel rendering approach is impracticable for parallelization on PC clusters. However, most graphics adapters internally use the sort-middle approach in their rendering pipeline [CDR02].

Sort-last

In the sort-last approach the arrangement is made after the geometry primitives have been sent through the rendering-pipeline. Before rendering, the different primitives are assigned to different nodes. For each frame, the nodes render their subset of primitives into separate frame and depth buffers. Afterwards all nodes read back both buffers into their main memory. The partial images are combined into a single image by comparing the depth values of the different depth buffers. For the final image, the pixel from the frame buffer whose depth value is the nearest to the view plane is chosen (see Figure 2.1, right).

Load balancing strategies

To accelerate the sort-first rendering process, the computational load must be distributed evenly among the cluster nodes. It is easy to produce situations where most processors are idle while only a single node has to render the complete scene. Hence, load balancing techniques are required. Usually the load is balanced by adapting rendering nodes' tile sizes [SZF+99, ACCC04, OU06, RRR06, Paj08] using heuristics, for example.

Using the sort-last approach, the composition of the different partial images requires a significant amount of time. While the rendering load usually decreases for a rising number of cluster nodes, the merging time increases. To distribute the load for composition evenly the *binary-swap* algorithm and its improvements can be utilized [MPHK94, MPHK08, TIH03, EP08]. Additional run-length encoding of the images reduce the load of the network and accelerate the image composition even more [HWC10, KPH+10].

Hybrid techniques

Sort-first and sort-last bring their own advantages and disadvantages. Using sort-first, the different partial images can be easily composed, while many elements are rendered redundantly in multiple tiles. Sort-last renders each element at most once, but the composition of the partial images needs much time. Both approaches can be combined to reduce the drawbacks [SFL01, SFLS00].

Hardware acceleration

Special purpose hardware has also been developed for parallel rendering [Whi92]. As defined in Flynn's taxonomy, there are *Single Instruction, Multiple Data (SIMD)* as well as *Multiple Instruction, Multiple Data (MIMD)* approaches. In SIMD system a single instruction stream is performed in parallel on multiple data streams. In contrast, a MIMD system performs a different instruction stream for each data stream. There are complete hardware systems to render 3D scenes fast [MBDM97, WLLB97, Ada05], in the form of special purpose components (for example the Voodoo^{TM}graphics accelerator) that can be applied to off the shelf computers [DY08, MOM^{+}01, SEP^{+}01].

Many modern graphics adapters provide techniques to combine the computational power of multiple graphics adapters [Res09, AMHH08]. There are different solutions such as SLI or Crossfire where homogeneous graphics adapters are combined. Other techniques, like LucudLogix *HydraLogix Engine* allow the usage of heterogeneous adapters [Lud10].

Next to Molnar's parallelization approaches there are further methods to render 3D scenes in parallel. One of these methods is often offered by the previously mentioned multiple GPU solutions and is referred to as *alternate frame*. Here, multiple graphics adapters or rendering nodes are used to compute consecutive images. Each frame is assigned to a render unit in a round-robin manner. While a GPU is displaying its frame, the following GPUs use this time to render their frame. The problem here is the method's scalability. When the number of render units increases, the latencies of the rendering system also increases. Another problem of this approach is the possible dependency of a frame on the previous one. In this case the required data must be send from one graphics adapter to another over the bus.

2.2.2 Out-of-core Rendering

Out-of-core rendering methods are necessary if a 3D scene is too large to fit into the primary memory of a single computer. In this case, secondary memory is used to store invisible scene parts. In general, the hard drive is used for this purpose. In comparison to the primary memory this storage has massive capacities, but at the costs of much greater latency. Thus, it is necessary to store all objects that contribute to the final image in local main memory to render the images with good performance.

Aliga et al. presented an interactive massive model rendering system to process scenes that exceed primary memory [ACW^{+}99]. Their approach combines many different rendering techniques to handle such large scenes (for example, hierarchical data structures, levels of detail, visibility culling, etc.). This was the first rendering system that was able to process scenes with more than ten million triangles [CKS03]. They employed a *from-region pre-fetching*, to load required data from the secondary memory.

To render models that exceed the primary memory pre-fetching and caching mechanisms are required. Extending the viewing frustum is a simple technique to load objects (see Figure 2.2), which are potentially visible in one of the next frames [VM02]. The iWalk rendering system [CKS02b, CKS03, CKS02a] stores a given 3D scene in an octree [AMHH08]. Using this spatial data structure, selective object loading is possible.

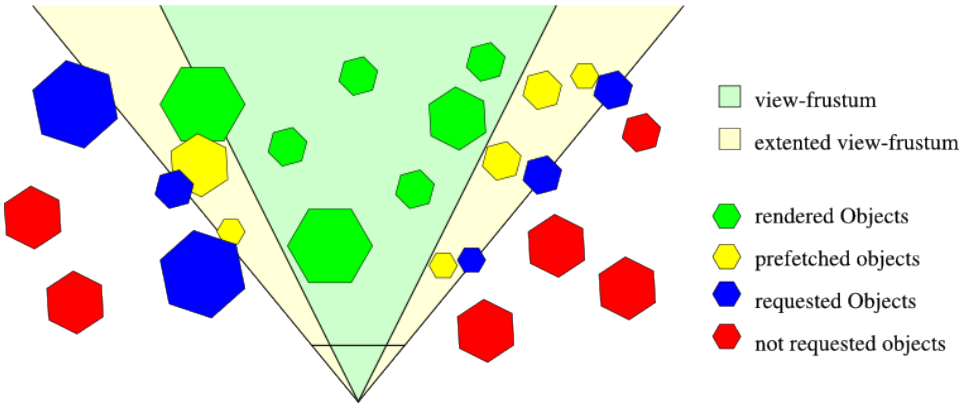


Figure 2.2: Pre-fetching in an out-of-core rendering system.

Sajadi et al. present an out-of-core rendering system that subdivides a given 3D scene using a k-d tree [SHDG+09]. The k-d tree is also used to find the data required by their rendering system. Their system reduces the data and cache-management costs by storing all objects within equal-sized memory blocks. Although this approach uses more memory than the total object size, it prevents the fragmentation of memory.

Goswami et al. present a parallel out-of-core rendering system for very large terrain data sets [GMBP10]. Their system provides a sort-first and a sort-last rendering. For the sort-first portion of the renderer they evaluate the behavior of the system when the display is segmented only by equally sized horizontal or vertical stripes. They show that the load is distributed more evenly if the stripes are orientated vertically. Due to terrain scenes' properties the triangle distribution is usually densest at the horizon, while in the sky the polygon count is usually very low. Using equally sized tiles is reasoned by a reduction of accesses to the data, stored on a network-attached storage.

Brüderlin et al. presented *Interview3D*, an out-of-core rendering system that is able to render CAD scenes of nearly arbitrary size [BHP07]. The system loads only that data into the primary memory that is potentially visible. Thus they use *visibility-guided rendering* which is an output-sensitive approach to reduce the rendering load. Required data is incremental updated from the secondary memory to prevent the renderer from stalling (small details can be delayed).

All rendering nodes of the parallel rendering approaches presented in this section require high capacity graphics adapters to achieve good performance. Typically, the performance of the entire system depends on the slowest node. The described out-of-core rendering approaches utilize the hard drive for storing the data-items. Our methods for primary memory exceeding scenes do not require - unlike other approaches - that the objects are stored in equal-sized blocks. In our methods we use a randomized sample tree for the spatial scene partition, where elements from lower levels in the octree are randomly lifted into higher levels. For this reason, each sample tree cell can have a different number of objects attached to it.

The data transfer rates of hard drives are very low compared to the rates that can be achieved using high speed networks. In other approaches, the data items are loaded

without testing the object for occlusion by other objects. In our rendering system, we use the nodes' RAM as secondary memory instead of hard drives. To ensure fast data access all nodes are connected via an Infiniband network. On the back-end nodes, all requested data items have to pass additional tests before they are transmitted to the rendering node's main memory.

2.2.3 Occlusion Culling

Occlusion culling is a common approach to reduce the number of objects sent into the rendering pipeline. Many graphics adapters provide an interface to retrieve occlusion information directly from the GPU – the so-called *hardware accelerated occlusion queries*. The queries return the number of potentially visible pixels for the queried geometry (i.e. if the query returns zero the geometry is invisible, otherwise it contributes to the final image).

The different algorithms can be categorized into three groups: conservative, approximative, and aggressive culling algorithms. While conservative algorithms usually overestimate the set of visible objects, approximate algorithms can terminate their visibility tests for various reasons, even before they determined all visible objects. Aggressive algorithms only determine objects that are visible, and in addition discard objects even if they are definitely visible.

All three approaches can be used with *from-cell* and *from-camera position* occlusion culling. Algorithms related to the first group determine sets of visible objects within a region in a preprocessing step. The visibility within a cell can be determined conservatively [TS91, NBG02] or approximatively [DDTP00, NB04, Lai05, LSCO03]. Other algorithms determine in a preprocessing, which regions are invisible for a given cell [SDDS00]. It is not necessary to determine the visible objects in advanced. There are methods that compute the from-cell visibility during runtime [RP05, MBWW07, BMW$^+$09]. When the camera is placed within a region, only objects from pre-determined set of potentially visible objects are processed. These approaches exploit spatial coherency.

From-camera culling algorithms are typically processed at runtime. Many of these runtime occlusion culling algorithms profit from the hardware accelerated visibility tests. Exploiting spatial and time coherency can reduce the number of performed tests. The conservative techniques determine all currently visible objects [HSLM02, BWPP04, GBK06, SBS06]. To accelerate the visibility tests object approximations can be used [DMS01, SKJF11]. Other techniques uses multiple graphics adapters to determine visible objects [GSYM03, SWF10a, SKJ$^+$11]. In contrast to these conservative approaches, the approximative techniques do not guarantee that all visible objects are displayed in the final image [KS00, CKS03, GBSF05, GM05]. Some of these approximative methods draw objects until a budget is reached, other techniques determine the visible objects incrementally. Several methods allow for choosing a conservative or approximative culling by parametrization [MBW08].

2.2.4 Mesh Simplification and Impostors

Level of detail and impostor techniques are one of the most common approaches to process massive models. The idea to replace objects by less complex approximations organized in a hierarchy has been introduced by Clark [Cla76]. Polygonal simplification techniques usually require objects of connected components, like chairs, balls, etc. Levels of detail (LOD) can be classified into three groups: discrete, continuous, and view-dependent [Lue01, LWC+02]. Discrete LODs are multiple versions of the same model with variable complexities [SZL92, Tur92, HDD+93, Kle97]. The different versions can be used for different objects-to-camera distances, for example. A continuous LOD has no individual version. Utilizing a data structure, an object's complexity can be refined or coarsened seamlessly [Hop96, SGG+00]. View-dependent LODs are based on continuous LODs that additionally use the current camera position as criteria for the chosen simplifications [LE97, EMB01].

Impostors replace scene parts as well, but they do not manipulate the original objects. Billboards replace scene elements with a texture that is placed on a simple geometric object (e. g. rectangle, cone, cube, etc.). Some techniques replace composited scene parts [SLS+96], others combine multiple billboards to replace a single object like a tree [YSK+02, DDSD03, DN09]. Particle systems, to visualize smoke or water, can also be realized with billboards [USKS06]. Other techniques, like our developed reliefboards, produce simple meshes that are not based on the original geometry to replace objects [ABB+07, GFB10, SJF10].

3 Preliminaries and Definitions

In this chapter we will introduce and define some expressions we require throughout this thesis. Additionally, we describe the PC cluster system used for testing. One of the PC cluster used has been modified over the time, resulting in different settings for the same PC cluster environment.

3.1 Objects, Models, and Scenes

For most of the algorithms we need the central concept of *objects* which are rendered. In this setting, an object is a set of triangles that builds a *semantic group* irrespective of the object's form, complexity, and spatial dimensions. An arbitrary, single polyhedron is defined as *geometric primitive*. Usually these groups of objects are given as input for our rendering algorithms. They are created using CAD applications or 3D scanners. The composition of multiple objects is called a *model* or a *scene*.

If the input is given as set of independent geometric primitives, or if the given objects are not suitable for our rendering algorithm, the input must be reorganized to fit our specific needs. For example, connected components can be found or geometric clustering techniques can be utilized.

To place objects' vertices, we use the Cartesian coordinate system. Every vertex is given as a triple $(x,y,z) \in \mathbb{R}^3$, representing a point in the three dimensional Euclidean space. We cannot use elements of the set of real numbers for our computations because of hardware's limitations. The computations are limited by hardware supported floating point numbers. Nevertheless, in this thesis we will use $\mathbb{R}$ for the used floating point numbers, unless otherwise stated. The coordinate system used in this thesis is right handed [Wat99] (see Figure 3.1). A box containing an object, whose sides are parallel/orthogonal to coordinate system axis is defined as *axis aligned bounding box*, short *AABB*.

3.2 PC Cluster

In the following section we will formally introduce the terms and definition related to a PC cluster, followed by the detailed PC cluster configuration used in the tests.

Definitions

In this thesis we use the terms processor, rendering node, and cluster node synonymously as atomic computation units of a PC cluster. Each processing element in a PC cluster

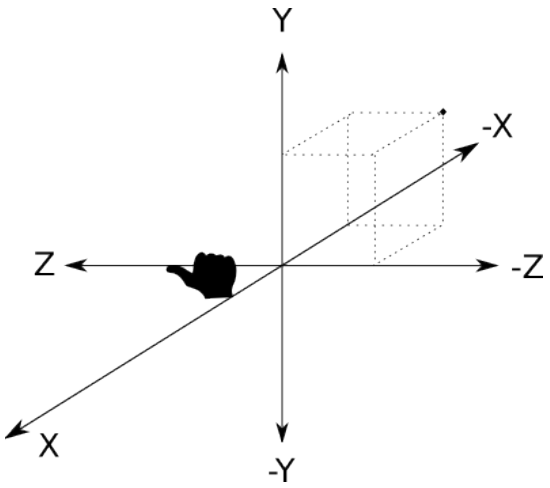


Figure 3.1: Right-handed coordinate system with a single vertex. Virtually, when a hand grabs the z-axis from the top and the arm is placed on the positive x-axis, the thumb points along the positive z-axis.

consisting of N nodes is identifiable via a unique id $PE_i \in \{PE_0, PE_1, \ldots, PE_{N-1}\}$. The used PC cluster is a distributed memory MIMD system, as defined in Flynn's taxonomy [Fly72], which means that the different nodes have no access to a shared memory space [Pac96]. For this restriction, each piece of information or data item must be sent and received over a network. The different algorithms developed for this thesis use message passing for communication. The MIMD system is programmed via *SPMD* (single-program, multiple-data). That means that each node executes the same program, but can process different data.

Configuration

The PC cluster we used for the evaluation was the Paderborn Center of Parallel Computing's (PC^2) Arminius cluster. Over time the cluster was modified so we have two different configurations: a large and a small configuration.

In the large configuration, the strongest visualization nodes provide an NVidia Geforce 9800 GX2 graphics adapter with 512 MiB memory, 16× PCI-e 1.0, two AMD Opteron 250 processors and 4 GiB RAM. The installed 64-bit OS was Fedora Linux 9. In this configuration 200 back-end nodes were available. Each back-end node provides an NVidia Quadro NVS280 graphics adapter with 64 MiB memory, PCI-e 1.0 (single lane), two Intel Xeon 3.2 GHz processors, and 4 GiB RAM. The installed 64-bit OS was Red Hat Enterprise Linux AS, release 4. All nodes are connected via Infiniband (using 16× PCI-e, about 1.8 GiB/sec at full-duplex transmission).

In the small configuration, neither the hardware configuration of the visualization node nor the hardware configuration of the back-end nodes has been changed, but the number of available back-end nodes is decreased to 16. The operating system was changed to the 64-bit version of Cent-OS 5.5.

Scenario I:

Static and Sparse Occluded Scenes Fitting into Processor's Primary Memory

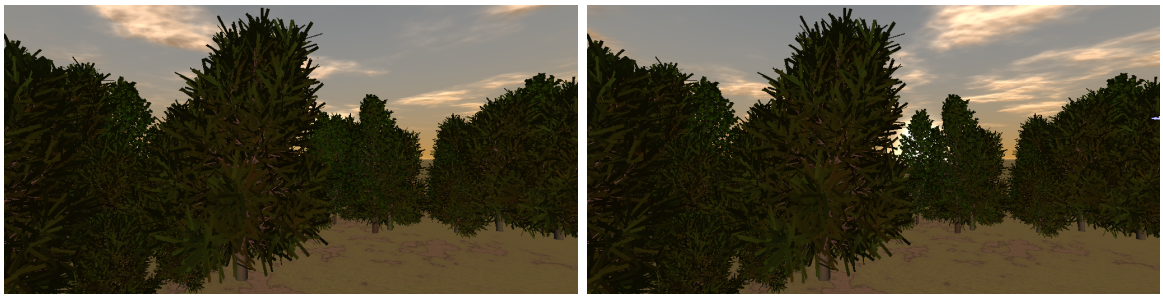


Figure 3.2: If trees are discarded too early it is possible to see the sunset behind them. This strong change in the illumination can influence shaders negatively.

Here, we will describe the challenges which inspired the development of *reliefboards*. Real-time rendering of large scenes is a challenge, especially if the objects' complete, mutual occlusion is sparse. In such scenes, camera positions exist where all objects must be rendered to produce a correct image. Examples for such scenes are forests where it is possible to see through gaps between tree's leaves. Here it is difficult to decide which objects can be discarded and which objects appear in the final image. This is reasoned by small gaps between leafs, branches, trunks, etc. where an observer is able to look through (see Figure 3.2). The discarding of objects which allows the background to be seen can lead to disturbing image errors. This can occur, for example, when shaders are used which use information of previous frames to determine the illumination.

Additionally, rendering highly tessellated scenes like woods can produce many aliasing artifacts. This is because of the small projection sizes of distant objects and the difficulty of determining the pixels' color (see Figure 3.3). A pixel's appearance depends on the primitives which are projected on its center or multiple other sample points included in this pixel [Ros05, AMHH08]. If the projection size of the primitives in a display region is smaller than a pixel, small changes in the camera position can result in noise.

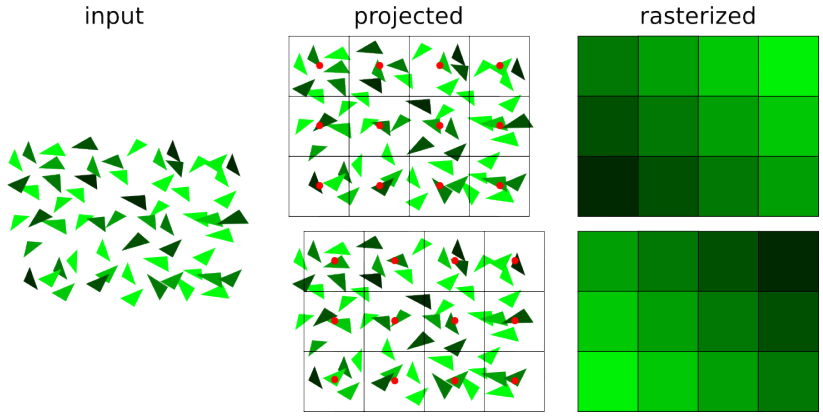


Figure 3.3: Minor changes in camera position can result in significant image changes. If these changes appear permanently during walk-through, it is called noise. Such effects can easily be produced by small structures in a forest scene.

Image noise during a walk-through can disturb users and reduce the overall image quality. There are many techniques to avoid this problem, however these techniques often need significant additional computation time.

One common approach to face this problem is the use of textures that are placed on planes for distant objects. These *billboards* are simple and achieve good results, but they do not cover other objects correctly. Additionally, if only one billboard is used per object the parallax effect is missing, where elements change their distance in screen space if the camera is moved sideways (see Figure 3.4).

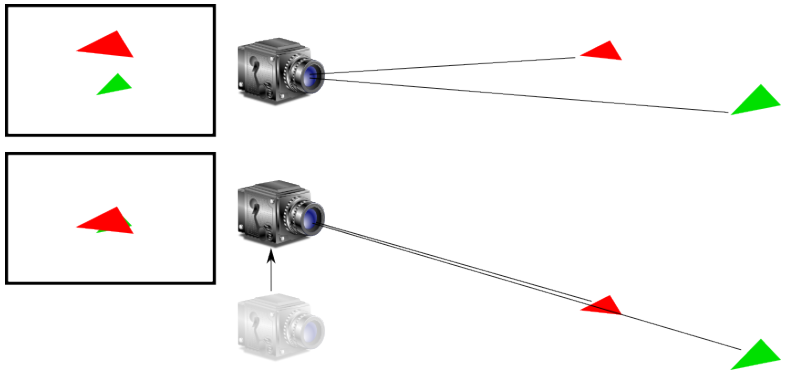


Figure 3.4: Schematic representation of the parallax effect. Due to the motion parallel to the viewing plane, objects change their distances.

To process scenes as described before and to avoid the occurring problems, we developed an object simplification, the so-called *reliefboards*. We developed a parallel rendering system that uses these approximations. In contrast to other parallel rendering systems, the presented system scales in image quality instead of frame rate. For load balancing, we used a randomized data management protocol. To achieve an evenly distribution of the load we distribute the objects randomly and redundant among the PC cluster's back-end nodes. This distribution yields good results and is easy to compute. A deterministic computation of the object distribution is hard. With the needed properties this problem is similar to the *set-packing* optimization problem. We analyzed if a polynomial-time local search algorithms could achieve suitable results. We have shown that the localized version of set-packing is already $PLS_{complete}$ for a two-differ-neighborhood [DS10]. That means, starting with an feasible solution we try to optimize the packing by a sequence of real improvements, in which we exchange at most two elements in each step. The *PLS* completeness of set-packing shows that the length of this sequence could be exponential, until we reach a local optimal solution.

4 Reliefboards

Often complex scenes are composed of many individual objects. Each of these objects can have a moderate complexity but the composition of all of the objects is too complex to be rendered by a single computer, even if they fit into its main memory. Some examples for this are trees or skylines. When a user observes the scene through an aerial perspective, objects barely occlude each other. In worst case, all objects must be rendered.

To render such scenes in real-time we developed *reliefboards* (see Figure 4.1). This approximation technique is similar to regular billboards, but they have improved properties (e.g., they support the parallax effect). To generate reliefboards in our parallel rendering system the PC cluster's back-end nodes do not even need graphics adapters, a software rendering system can be used. Our requisite to the back-end nodes is that the sum of their memory is large enough to store the scene's objects redundantly [SJF10].

4.1 Overview and Summary of Results

Our system architecture uses a single visualization node equipped with a powerful graphics adapter and a PC cluster consisting of a large number of back-end nodes not necessarily offering accelerated 3D graphic performance. The displayed images of the scene are only rendered on the visualization node. We assume that a 3D scene consists of a certain number of atomic objects. The visualization node renders a subset of the objects with their original geometry. A large number of elementary objects are replaced by reliefboards (see Figure 4.1). A reliefboard is a mesh consisting of colored vertexes placed

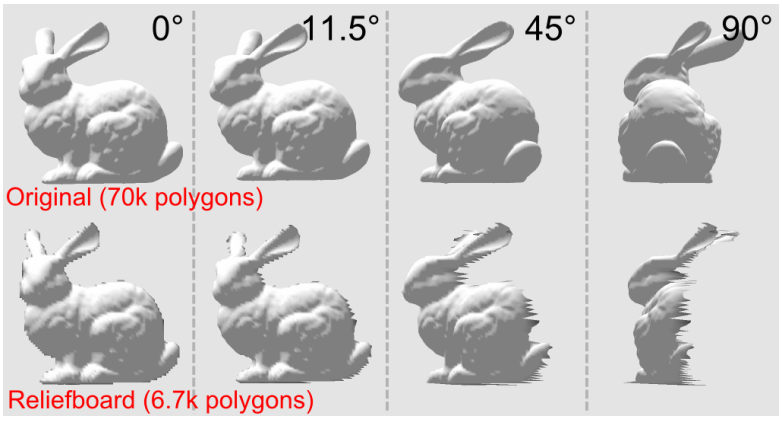


Figure 4.1: Approximating objects by reproducing their surface by reliefboards.

on a regular grid and shifted along the grid's normal vector. Reliefboards approximate the original objects and their quality depends on the viewer's position. Therefore the reliefboards have to be updated periodically. This is done by the PC cluster's nodes, which compute the reliefboards asynchronously.

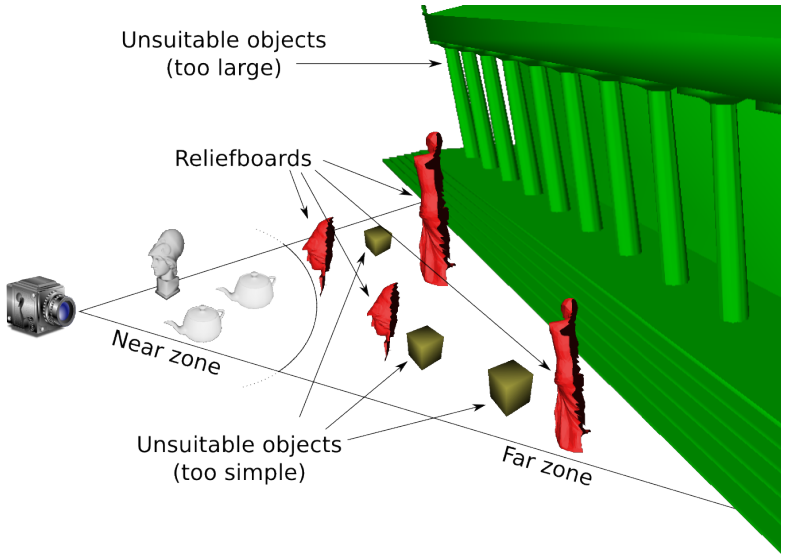


Figure 4.2: Objects which are near the viewer, too large or too simple are rendered with the original polygons. Objects which are far away and suitable are removed and rendered with reliefboards.

Rendering cycle: The visualization node's rendering cycle consists of three steps. First, all *unsuitable objects* are rendered. Unsuitable objects are rendered with original geometry because their diameter is too large or the number of triangles is too low and thus a replacement with a reliefboard would not gain any speedup (see Section 4.5). Replacing objects which are too large could lead to significant image errors. Second, all *suitable objects* are rendered in a front-to-back manner. An object is suitable for replacement with a reliefboard if the number of triangles and the diameter of the object is adequate (see Section 4.5). We call the area where suitable objects are rendered with original triangles the *near zone* (see Figure 4.2). Third, distant, suitable objects behind the near zone are replaced and rendered with a reliefboard. We call this area the *far zone* (see Figure 4.2).

We developed reliefboards and a related parallel rendering algorithm. Reliefboards are easy to compute and yield good results (see Figure 4.3). Using these object impostors allows us to accelerate the rendering process of complex scenes. Additionally, the fluctuations of the required rendering times are reduced by reducing the strong fluctuations in the number of polygons to render.

Before our rendering algorithm starts to render a preprocessing step is necessary (see Figure 4.4). For our parallel rendering method we need suitable objects. These objects

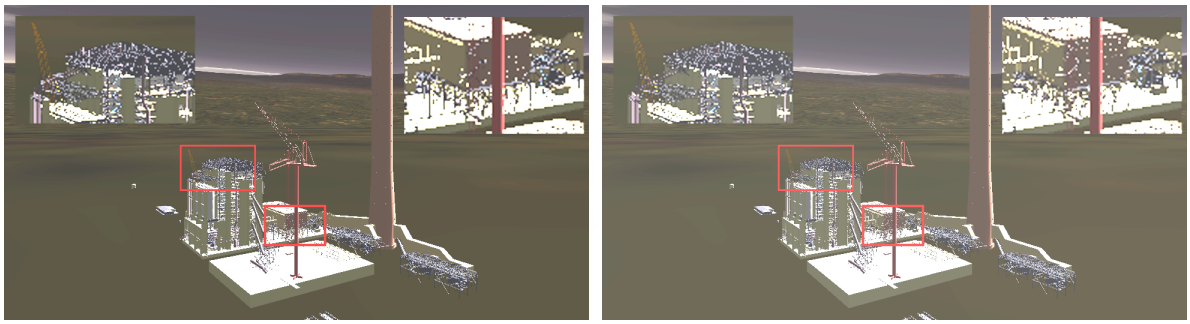


Figure 4.3: Reliefboards applied on the UNC Power Plant scene. The left image shows the scene rendered with the original geometry. For the right image reliefboards were applied. The red frames shows the highlighted areas.

are groups of many triangle that are pooled in a small space. If the given input does not fulfill these properties, we have to compute the required objects during preprocessing. To determine suitable objects we can use an agglomerative clustering algorithm. These objects are needed for the computation of the reliefboards as well as for the load balancing. For this purpose we use a data management protocol. During runtime we balance the rendering load of the different back-end nodes with the c-collision protocol (see Section 5.2 for a brief overview).

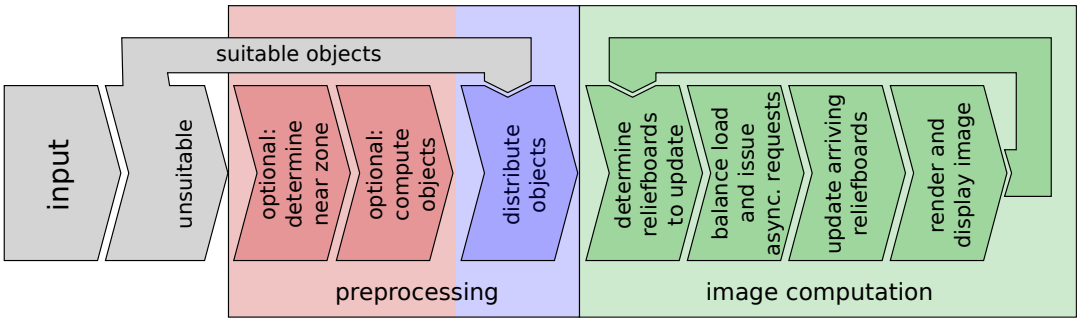


Figure 4.4: Our system transforms unsuitable inputs to objects that meet our needs. These objects are distributed among the back-end nodes. In every loop, reliefboard updates could be requested or received before a new image is rendered.

4.2 Related Work

Here, an overview of related work for mesh replacement and displacement mapping will be presented.

Mesh Replacement substitutes objects' original geometry with a simplified representation, which typically reduces rendering costs (e.g., impostors or LODs). Simple

billboards replace original geometry by single textures placed on a rectangle or another simple geometric object. A disadvantage of this technique is that billboards can intersect at most once (see Figure 4.5). To reduce this negative effect, it is possible to use multiple billboards instead of one [YSK+02, DDSD03].

Jeschke and Wimmer introduced *Textured Depth Meshes* to accelerate the rendering process [JW02]. To create these impostors a new mesh and texture are generated from a voxelizated scene part. These simplifications can be used in fixed view cells as simplifications for complex objects.

Andújar et al. presented omni-directional relief impostors to improve rendering performance [ABB+07]. In a preprocessing step they compute *relief maps* from different directions for a given object. During runtime, depending on the camera position, some of these maps are combined and rendered. Thus, the number of polygons sent into the rendering pipeline is reduced. In contrast to the reliefboards in our rendering system all these impostors are generated in a preprocessing step and not on-the-fly during runtime.

Shade et al. [SLS+96] use textures to reduce scene complexity. They compute a k-d tree that contains the scene's geometry. By utilizing this k-d tree, the cell's geometry can be substituted by a rendered image of it, as seen from the observer's position. The main drawback of this technique is that it is sometimes possible to see through a gap between textures and real geometry. Furthermore, if the position is changed, there is no parallax effect.

LODs, as for example Hoppe's *Mesh optimization* [HDD+93] or his developed *Progressive Meshes* [Hop96] create simplifications based on an object's mesh. Usually, these LODs must be generated during preprocessing and the quality must be checked manually. Reliefboards are computed in parallel, automatically, and on-the-fly. They can be used for groups of different objects and generate a less complex representation for all of them at once. Detailed descriptions of other LOD techniques have been pooled by Luebke et al. [LWC+02].

Displacement mapping is used to add depth to simple surfaces. In contrast to techniques like bump or normal mapping [AMHH08], which only simulate the visual effects of depth (e.g. by lighting or shadowing), displacement mapping changes the geometry of a surface [Coo84]. These techniques support the parallax effect and intersection of objects, such as parallax mapping [KTI+01, Wel04, AMHH08]. Typically, a simple surface (usually a grid) is transformed to a more complex one by using several maps. By using these meshes, correct shadows and valid occlusion can be computed. Furthermore, objects' silhouettes appear realistic [WWT+03]. Due to not respecting the distance between an object and the camera the underlying geometries are often overtessellated, so that a retessellation is needed [GH99, DH00]. Typically, the displacement of the vertices must be computed in every frame. If the number of vertices to displace is high, this technique can influence the rendering time negatively. For this reason, reliefboards do not necessarily displace the vertices in every frame. The vertices of the grid are shifted only once and used for many frames.

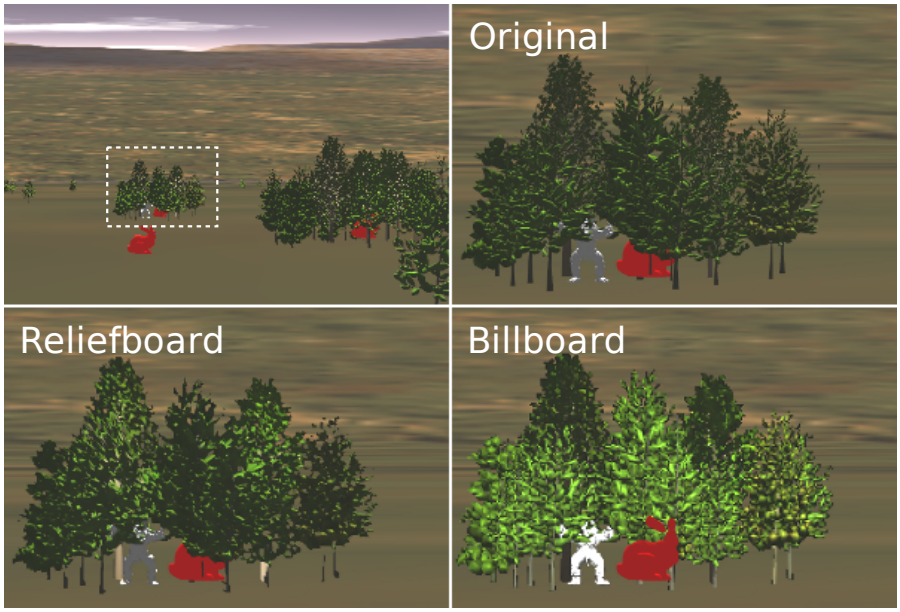


Figure 4.5: The bunny is rendered with original geometry. The lower left picture shows a picture detail using reliefboards. The upper right picture shows the original picture detail and the lower right shows the picture detail using simple billboards. These simple billboards neither support multiple intersections nor correct lighting.

4.3 Distributed Rendering

The parallel rendering system uses two different node types: a single visualization node and a large amount of back-end nodes. The back-end nodes are utilized for the distributed creation of the reliefboards. The visualization node requests and uses the reliefboards to render the actual scene to the screen and provides the interface for the interactive movement of the observer.

Rendering on the visualization node

The rendering of a frame on the visualization node is performed in three steps.

1. In the first step, all unsuitable objects inside the viewing frustum which were identified as too large or not complex enough are rendered to the screen. Their polygon count is thereby accumulated. In neither case would it be practical to replace these objects by reliefboards. If an object is too large, even a slight change of the observer's position is likely to cause an unacceptable change in the perspective. If the object consists of too few polygons, the original object's rendering time could actually be better than the approximation's.

2. In the second step, the other objects are rendered in front-to-back order from the observer, as long as the overall polygon count does not exceed the value of a given

parameter t_{nz}. If the number of objects is small enough, we can sort the objects in every frame using a standard sorting algorithm. Otherwise, one should utilize a spatial data structure like a loose octree for storing the objects and traverse it in front-to-back order. This implicitly results in the subdivision of the frustum into the near zone, where only the original geometry is rendered and the far zone, where reliefboards can be used (compare to Figure 4.2).

3. In the last step, a decision is made for each of the remaining objects in the viewing frustum, whether the original object or the corresponding reliefboard should be rendered. If no reliefboard has yet been received for an object, the original object is rendered and, if no reliefboard request for this object is pending, a new request is issued. Each request consists of an unique identifier for the object, the direction vector from the observer's position to the object's center and the object's projected size in pixels for the reliefboard's resolution. Requests are processed asynchronously and the resulting data arrives after a few frames.

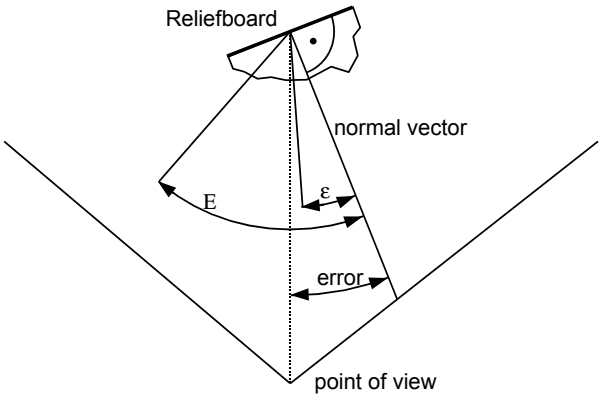


Figure 4.6: When the angle error exceeds ϵ a new reliefboard is requested by the visualization node. If the angle error exceeds E, the visual error is considered too large and the original geometry is rendered.

If a reliefboard has already been received from the visualization node, the current *angle error* is calculated. The angle error is defined as the angle between the direction vector that was used as parameter for creating the reliefboard and the current direction vector from the observer to the center of the board (see Figure 4.6). This is taken as an indicator for the visual similarity of the reliefboard's projection and the original object. If the error is higher than a fixed parameter ϵ, and no request for this object is pending, then a request for a new reliefboard incorporating the current direction is issued. If the error is unacceptably large (defined by another fixed parameter E), then again the original object is rendered until an updated version is received. Another source for artifacts produced by a reliefboard is the difference in the current projected size to the reliefboard's original resolution. If the observer approaches a low resolution mesh, blocking artifacts occur. If the observer moves away from a high resolution mesh, aliasing artifacts occur. Hence, we use the relative difference in reliefboard's current projected size

and the projected size when generated as an additional indicator to trigger an update request.
If a reliefboard with an acceptable error is available, the corresponding relief mesh is rendered at the original object's position, oriented towards the direction it was created in. This should be the most common case, as a reliefboard can normally be used for many frames.
Also, in every frame a check is performed whether newly created reliefboards have been received from the back-end nodes. If so, the corresponding old ones are replaced. This results in the occurrence of a slight popping effect in the next frame. This effect is more visible as the old and new direction vector differ. Nevertheless, these effects disappear when applying blending. If the new reliefboard replaces an original object, the reliefboard's direction vector differs from the actual direction in which the object was seen last.

Distributed creation and load balancing

One goal of the distributed creation of the reliefboards is to keep the latency for their requests as low as possible. When a request is delayed, it is possible that the out-dated, old version of a reliefboard is used for a longer period resulting in a increased visual error. If the request for an object, which is currently rendered with its original geometry, is delayed the rendering time may increase.

Initially we distribute the data of all objects randomly and redundantly, with a fixed number of replicas over the back-end nodes. In order to assign the requests to the nodes at runtime, we make use of a modified version of the c-collision protocol [Ste96a], which aims at having a low contention of every network node and spreading the requests evenly over the nodes (see Figure 4.7). This protocol is implemented on the visualization node and is executed after each frame if new requests were made.

The assignment of requests is performed in rounds. In every round, a node handles at most c requests. If more than c requests arrives during a round, the node answers none of them. We begin with 1 as value of c. If not all requests can be handled with that value, c is increased until one node can fulfill all its requests. Then c is reset to 1 and the distribution is continued until all requests are assigned to a node. The assignment of a request to a node persists, until the reliefboard is received.

The dynamic handling of the value of c has the disadvantage of not being able to ensure a maximal contention as with the original protocol, but guarantees the handling of all requests, prevents the protocol from deadlocking and allows asynchronous communication.

4.4 Reliefboard Structure and Creation

A reliefboard's foundation is a mesh consisting of colored vertices placed on a regular grid which are shifted along the grid's normal vector (see Figure 4.8). The general orientation of the grid's base plane is orthogonal to the vector from the center of the

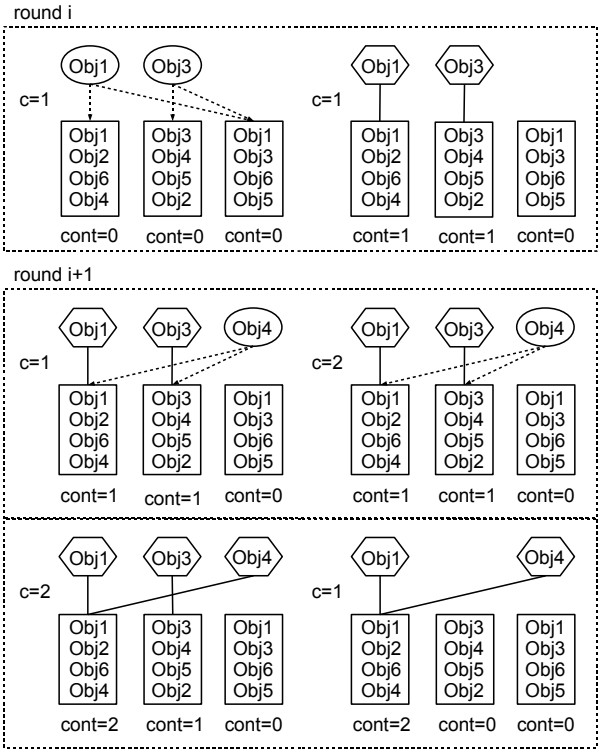


Figure 4.7: Circles represent requests and hexagons represent handled requests. The data items are distributed redundantly and randomly among the different nodes

reliefboard to the observer's position at the time when the reliefboard is created. This absolute orientation stays fixed as long as the reliefboard is valid. It does not change when the observer's position changes, unlike some simple billboards that always face the observer.

The displacement of the vertices on the grid reproduces the surface of the approximated object as seen from observer's original position in orthographic projection (i.e. without perspective distortion). If the position of a vertex does not lie in the area that is covered by the original object's projection onto the reliefboard's plane, this vertex is removed from the mesh. As a result, the reliefboard's silhouette reproduces the original object's silhouette (again, as seen from the observer's original position). The vertex colors and vertex normals are also taken from the projection of the unlighted original object. The combination of unlighted color and the normal vector allows exploiting the normal (possibly dynamic) lightning techniques, used during the reliefboard's rendering on the visualization node. If the perspective distortion or the lightning would be applied during reliefboard's creation these effects would be applied a second time when we render the reliefboards on the visualization node.

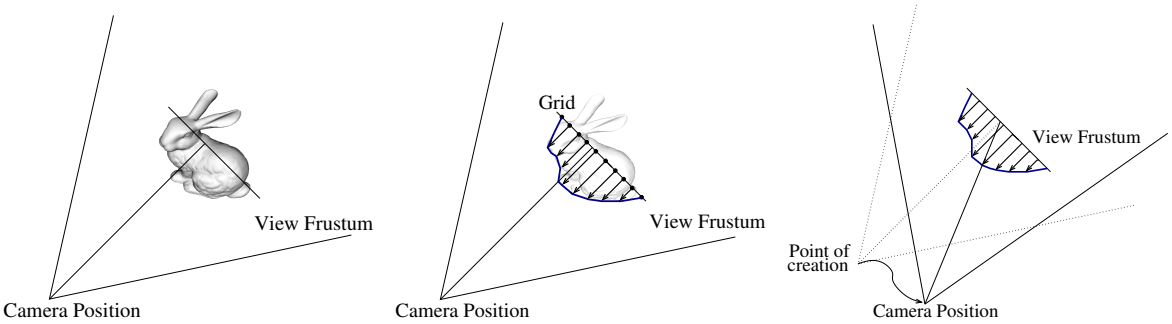


Figure 4.8: Image (a) shows from which position the reliefboard is created. Image (b) visualizes how the vertexes of a grid are shifted. Image (c) shows how the reliefboard holds its original alignment for multiple camera positions.

Creation

The data and parameters necessary for the creation of a reliefboard are:

- The geometry (including colors, normals, etc.) and the position of the original object.
- The directional vector from the observer's position to the original object's center (which also becomes the center of the reliefboard).
- The resolution of the grid. As the reliefboard should reproduce the original object as well as possible, at least at the time it is created, the resolution is set to the current projected size of the object (in pixels). In this way every pixel is initially represented by one colored vertex.

For the computation of a reliefboard, the original object's orthogonal projection needs to fit into an area on the given screen's resolution, the desired side facing towards the camera. Hence the object is initially translated and scaled accordingly (see Figure 4.9).

Then four intermediate, two-dimensional color maps of the given resolution are created by rendering the original object onto the screen (or to an off-screen buffer). As we want to exploit a cluster's nodes (with possibly not up-to-date graphics hardware) for the generation of the reliefboards, we do not use multi target rendering to render to multiple buffers at once. Instead, we use multiple rendering passes to obtain the data for the different target maps separately. Each of the four rendering passes exploits a simple shader to support the efficient processing of the data into the target map:

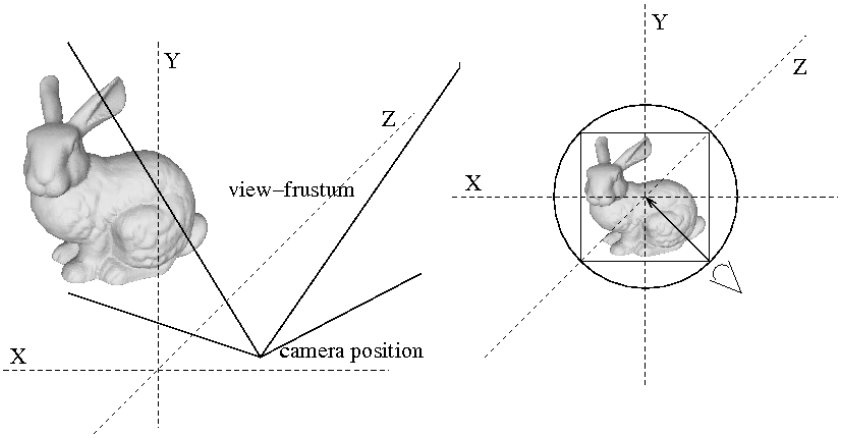


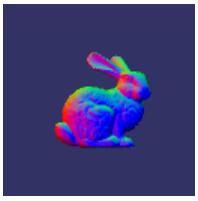
Figure 4.9: To compute the textures for reliefboard generation the camera is placed on the unit-sphere. The spheres center is placed at coordinate systems origin. Each object is scaled and translated such that its AABB fits into the sphere.

The *distance-map* encodes the displacement by extracting the depth information of the pixels. This map is used to shift the vertices of the grid. The higher a pixel's value the more the corresponding vertex is pulled towards the camera position, the lower a value the more the vertex is pushed away.

The *stencil-map* marks which pixels are covered by the projection and which vertices should be removed. The stencil is needed to determine which vertices are needed for the approximation and which have to be discarded.

The *color-map* reproduces the unlighted (but possibly textured) color of the object. Each pixel indicates the unshaded color of the corresponding vertex.

The *normal-map* encodes the surface normals in order to support correct lighting.



Finally, these maps are combined with the grid to form the mesh of the reliefboard at the given position. The created reliefboard replaces the original mesh and does not change its alignment during the walk-through.

4.5 Identifying Objects by Clustering

To achieve good results with reliefboards triangle groups are required instead of single triangles. The triangles in such groups should be pooled as close as possible. Below, these triangle groups are called *objects*. These objects are not limited to semantic groups like chairs. Instead, these objects can consist of arbitrary elements.

For our parallel rendering system we require that, on the one hand, the objects consist of a sufficient amount of triangles and, on the other hand, that the objects have a relatively small diameter. Reliefboards replace the triangles of objects by another groups of triangles, whose complexity has an upper limit. Thus, a sufficient triangle count of the replaced object is necessary to benefit from a replacement. As shown in the previous section, reliefboards are generated from several maps. The maximal resolution of those maps is an input parameter p, which defines the maximal complexity of the reliefboards. This resolution also defines the granularity of the simplifications. Our tests have shown that maps with a side length of 256 pixels yield good results. The smaller the diameter of the replaced object the better reliefboard's quality. If the diameter of an object which is to replace is too large, the generated simplification is to coarse.

For entered input there are two options: the entered objects meets the needs for our rendering system or they do not. In the first case, the entered objects can be used without any additional computation. In the second case, we need to compute suitable objects.

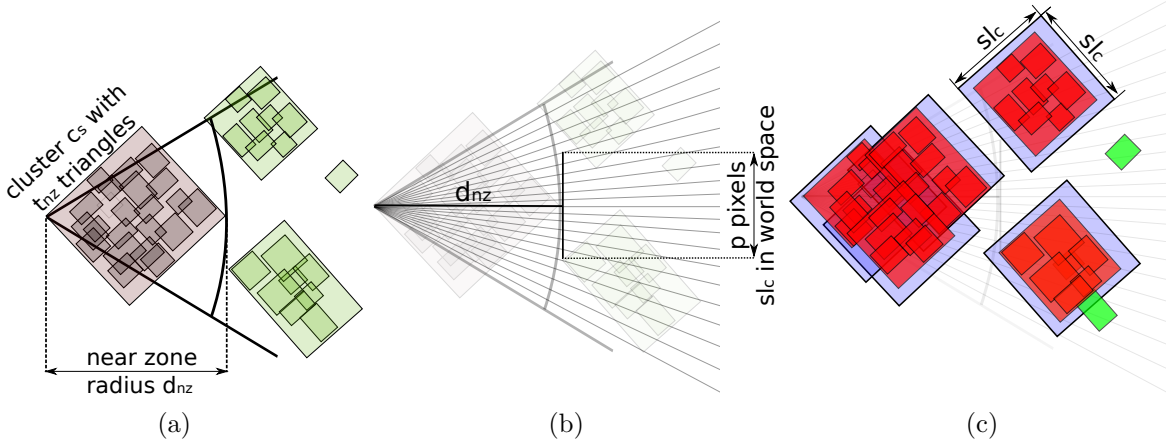


Figure 4.10: At first we determine the expansion of the near zone. Afterwards we calculate the size of suitable objects, followed by the determination of the objects for our rendering algorithm.

For a good image quality we render the objects close the the camera position only with their original geometry. In this near zone we limit the number of rendered triangles of the original objects. Our tests have shown that a third of the maximal amount of triangles that can rendered per frame is a good triangle limit for this zone. Additionally, beyond this near zone the projected size of the replaced objects must not be bigger than the side length of the textures used to generate the reliefboards. Thus, we have to

determine the expansion of the near zone. We compute the smallest cluster of triangles that consist of this triangle limit. The diameter of this cluster defines the size of the near zone (see Figure 4.10a).

Using the expansion of the near zone, display's resolution, and the size of the textures to generate reliefboards allows for determination of the size of our clusters (see Figure 4.10b).

In the last step we compute the final object clusters for our parallel rendering system (see Figure 4.10b). The size of these clusters can be smaller than the calculated maximal size, but they can be larger. Another property is that the clusters can insect each other which does not influence our rendering method.

4.5.1 Clustering Algorithm

Suitable objects are needed for our parallel use of reliefboards. The objects must have a reasonably high triangle count and must be compact in space. If the objects are given in a suitable form, reliefboards can be applied directly. Otherwise, if the objects are unsuitable or the geometry consists of triangles in an arbitrary order, we have to segment the scene into several, new objects. Therefore, we perform a preprocessing stage consisting of three phases. In the last two phases we use a heuristic, based on agglomerative clustering [DE84].

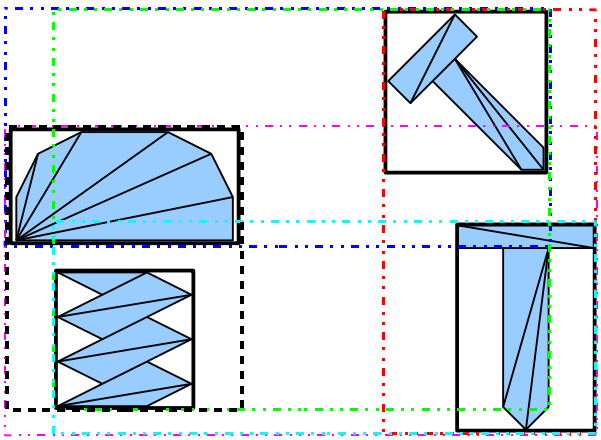


Figure 4.11: Clustering based on the smallest joined bounding box. The two objects in the dashed black bounding box are joined next.

In the preprocessing's first phase we have to process the input. First we remove redundant vertexes and faces from the given input. Afterwards, we determine connected components and their axis aligned bounding boxes. These partial meshes are the *base-clusters* for our agglomerative clustering algorithm.

In the next phase we choose the near zone range d_{nz}. In this zone it is guaranteed that an object's original geometry is always rendered. Using the radius of this zone, we calculate the maximal side length sl_c of the clusters. These clusters can be substituted by reliefboards. For this calculation, we need the maximum amount of triangles t_{nz} that

can be rendered in the near zone, the reliefboards' maximal projected side length in pixels p, and the display resolution r as parameters.

Finally, we determine the required clusters with the previously determined side length sl_c. During rendering, these clusters can be replaced by reliefboards. Below, phase two and three are described in detail.

Phase II: Choosing near zone

In the beginning of this phase we calculate the cluster c_s with t_{nz} triangles which has the "smallest" diameter. To do this we use our agglomerative clustering algorithm as described later in this section. The diameter of c_s is the range of our near zone d_{nz}. Using the determined d_{nz} and the given parameters p and r, we can calculate the needed side length for the clusters sl_c.

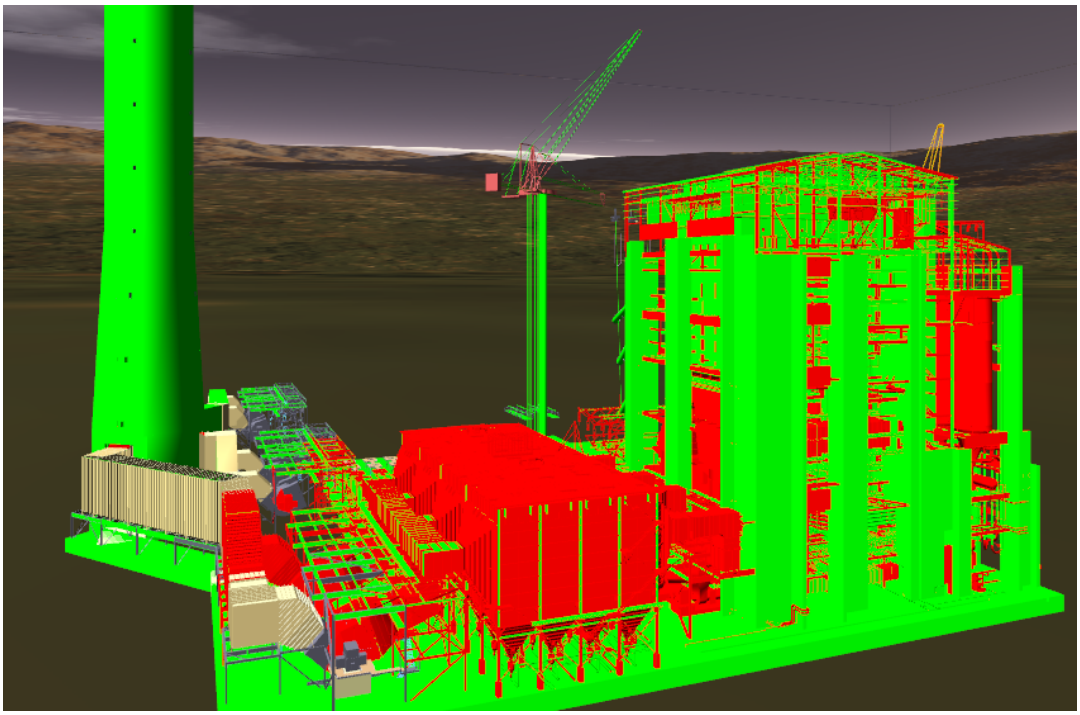


Figure 4.12: Clustering of the UNC Power Plant. Green (brighter) objects exceed the maximal side length of clusters and red (darker) ones are suitable clusters. Different colored parts are clusters, whose complexity is not large enough to replace them with reliefboards.

Phase III: Determining suitable clusters

In the algorithm's third phase we use the base-clusters and sl_c in the agglomerative clustering algorithm. At first we remove all base-clusters whose maximal side length is bigger than sl_c. A replacement of such a base-cluster would result in massive pixel errors because of reliefboard's limited granularity, given as parameter p. Hence these base-clusters should not be substituted by reliefboards (these objects are unsuitable). The

clustering algorithm combines the other clusters as long as their longest side is smaller than sl_c. When a cluster exceeds this limit it is completed and removed from further clustering. The resulting clusters are the objects which can potentially be replaced by reliefboards. In Figure 4.12 these different types of objects are visualized. Green objects are too large to be replaced by reliefboards (they are unsuitable) while red objects are suitable. Objects in Figure 4.12 which are rendered in a different color are also unsuitable because they are too simple to be replaced.

Agglomerative cluster algorithm

Our agglomerative clustering algorithm's inputs are the base-clusters as described before. Naive agglomerative clustering algorithms compute the joined axis aligned bounding box (AABB) for all potential smallest cluster pairs. Here we combine two clusters whose longest side of their combined AABB is minimal with respect to all other cluster pairs (see Figure 4.11). For optimization purposes we do not compute all possible joined AABBs. Instead we compute only a constant number of AABBs for each cluster, we use only those clusters that are nearby as described later in this section. By this we reduce the clustering time from $O(n^3)$ of the naive algorithm to $O(n \log n)$, where n is the number of connected components at the beginning.

To determine the cluster efficiently we use a loose octree [Del00] and a priority queue of cluster pairs ($cluster_a$,$cluster_b$),$a \neq b$. The queue's first element is always that pair with the smallest joined AABB. While building the loose octree, its nodes are subdivided as long as they include more than a defined, constant number of base-clusters. Every loose octree node is identifiable by a unique ID $i \in \mathbb{N}$.

Afterwards we determine for all loose octree nodes $cell_i$ the set of adjacent loose octree cells on the same octree level: $adj_i = \{cell_i,cell_x,cell_y,\ldots,cell_z\},1 \leq |adj_i| \leq 27$. The union of the clusters in the loose octree cells in adj_i is stored in another set uac_i. While the clusters within the different $cell_i$ are disjoint, different uac_i can include equal clusters. For each uac_i we compute the smallest cluster pair scp_i and push this pair into the priority queue.

The following steps are performed to determine a new cluster and to reduce their total number: As defined, both clusters ($cluster_i$,$cluster_j$) of the queue's first element build the smallest cluster. Clustering this pair requires an update of the priority queue and the loose octree. For that we determine the octree nodes $cell_i$ and $cell_j$ that contain the clusters. From these loose octree nodes we get the corresponding sets of adjacent cells adj_i and adj_j. For all octree nodes $cell_k \in adj_i \cup adj_j$, we remove the clusters $cluster_i$,$cluster_j$ from $cell_k$ and from its associated uac_k. Additionally, we remove scp_k from the priority queue.

After all copies of $cluster_i$ and $cluster_j$ are removed from the loose octree we join these clusters to $cluster_n$. The new cluster is inserted into the loose octree at cell $cell_n$. Next, for all octree nodes $cell_k \in adj_i \cup adj_j \cup \{cell_n\}$ we update the uac_k, determine the new scp_k, and push it into the priority queue.

While repeating this algorithm the cardinality of at least one uac_i decreases. If the number of included clusters in a loose octree node is below a threshold, we lift all clusters

from that node $cell_i$ into its parent node and remove its scp_i from the priority queue. Afterwards we update $cell_i$'s parent node and delete $cell_i$ from the tree.

4.6 Evaluation

In this section, we present empirical results showing the practicality and the main characteristics of reliefboards. To evaluate system's scalability we analyze the response time for reliefboard requests and the effects if increase the number of copies of each suitable object. Additionally, we analyze how a larger number of back-end nodes influence the image quality. To gain information about systems rendering performance we compare the rendering times of our system with the rendering times of a system that does not use reliefboards.

Evaluation system

We implemented the presented techniques using the programming language C++ (GCC-3.4.0), OpenGL for rendering and MPI (ScaliMPI-3.12.0-1) for the communication between the nodes. For the evaluation we use the large configuration of the PC^2 Arminius cluster. The frame's resolution is set to $1{,}024 \times 768$ pixels.

Scenes and parameters

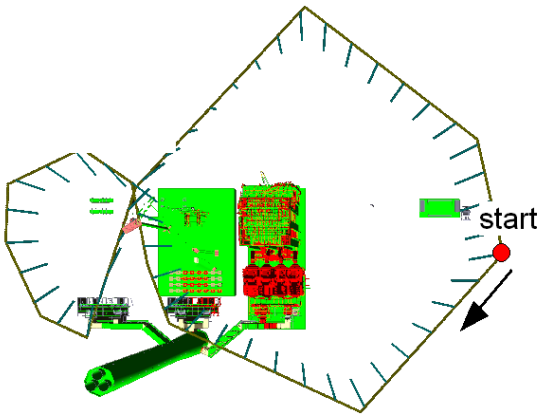


Figure 4.13: Camera path in the UNC Power Plant scene. The path is traversed in 6,000 steps at a uniform speed.

We used two different test scenes: A forest composed of many (not instantiated) models of the Stanford bunny and different kinds of trees ($\approx$ 25 M triangles in 250 total objects see Figure 4.5) and the well known UNC Power Plant model ($\approx$ 12.4 M triangles in 150 total objects). As the forest scene consists of relatively compact objects with almost uniform complexity, it is possible to use these objects directly without any clustering. The power plant's original objects are very complex and lack locality (e.g. long

pipes or a chimney). Hence, to allow the proper application of reliefboards, we defined 150 new objects by clustering the scene (see section 4.5).
The camera path used for the power plant is shown in Figure 4.13. The Forest scene is circled by the camera in 6,000 camera positions.

In our first tests we started an update when an angle error of $\epsilon = 11.5°$ was exceeded or when the projected side length of the reliefboard differed more than 30% from the original. The original geometry was rendered when the angle error exceeded $E = 45°$. The value for t_{nz} is set to 3.5 M triangles.

The number of replicas for the objects placed on the back-end nodes was set to three, as a compromise between a sufficient distribution of the requests over the nodes and the increased demand for memory for additional copies.

Scalability

In the first scalability test we evaluate the influence of the number of back-end nodes used on the requests' average response time. Due to the need of storing the data redundantly on different nodes we use at least four back-end nodes in all our tests. This is the average time that passes from the initiation of a request until the corresponding reliefboard has been received by the visualization node. This includes the time for completing all prior requests on a node, rendering the original geometry, reading the frame buffer in the desired resolution several times (see section 4.4), and finally composing and sending the created reliefboard to the visualization node.

Our experiments show that increasing the number of nodes decreases this response time, as the average congestion (the average size of the queue of open requests) on every node decreases (see Figure 4.14). If many back-end nodes are available, the response time converges to the time needed for the construction and transmission of a single reliefboard. This implies that, for a given scene and an amount of requests (implied by the speed of the observer's movement), increasing the number of nodes past a certain point does not improve the performance.

The time needed for updating all reliefboards in the scene (e.g. when the observer turns around very quickly) is not much higher than the average time needed for a single request, due to the parallel execution. For the UNC Power Plant scene, the time for a complete update is nearly the same as the average response time; for the forest scene, the time is about twice the average response time.

In our our second scalability test we evaluate the influence on the c-collision protocol when the number of nodes is increased. We use the c-collision protocol to distribute the rendering load among the back-end nodes. Due to the asynchronous communication and the permanent changing situation on the visualization node, requests for new reliefboards can be delayed. Instead of using an unchangeable c in our implementation of the protocol, we increase c to assign requests (described in Section 4.3). The number of back-end nodes and the number of copies should influence the maximum c reached during the walk-through.

For this test we discard the maximum contention of the nodes, because our algorithm computes new reliefboards for all suitable objects at the start of execution. When

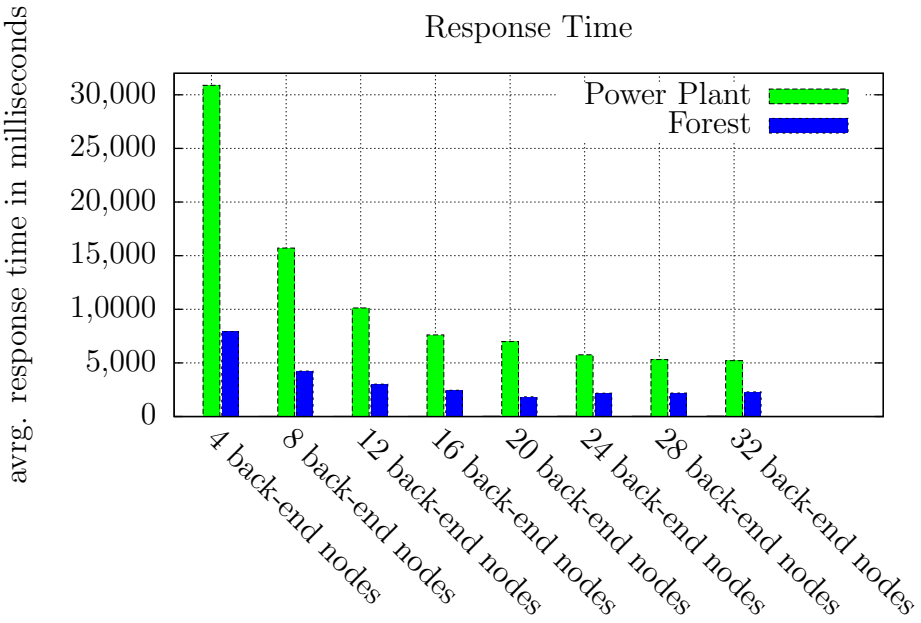


Figure 4.14: Average response times for requested reliefboards. The more nodes, the faster is the response – to the point where all requests can be processed immediately.

reliefboards' initial computation is finalized, and the contention of all back-end nodes is zero, we start the measurement. The resulting maximal c's are displayed in the diagram of Figure 4.15.

The diagram shows that the maximal value of c decreases if the number of back-end is increased. The higher the number of back-end nodes the lower the number of reliefboards each back-end node has to compute. In the UNC Power Plant scene we were not able to measure a significant effect when the number of distributed copies is increased, unlike in the forest scene. Noticeable is the significant increased c when we used four back-end nodes and we distributed each object three times. In this case the c is about 2.35 times larger than that when we distributed only two copies. This is due to the amount of data every back-end node has to store and process. Memory space that is required to store the objects exceed the GPU's available memory on the back-end nodes. Due to this objects must be rendered from nodes' main memory, which is less efficient. Thus processing a request requires more time and more new requests for other objects appear.

If more than four back-end nodes are used, the system takes advantages of the redundant placement. In the forest scene the maximum c reached decreases when the number of copies is increased.

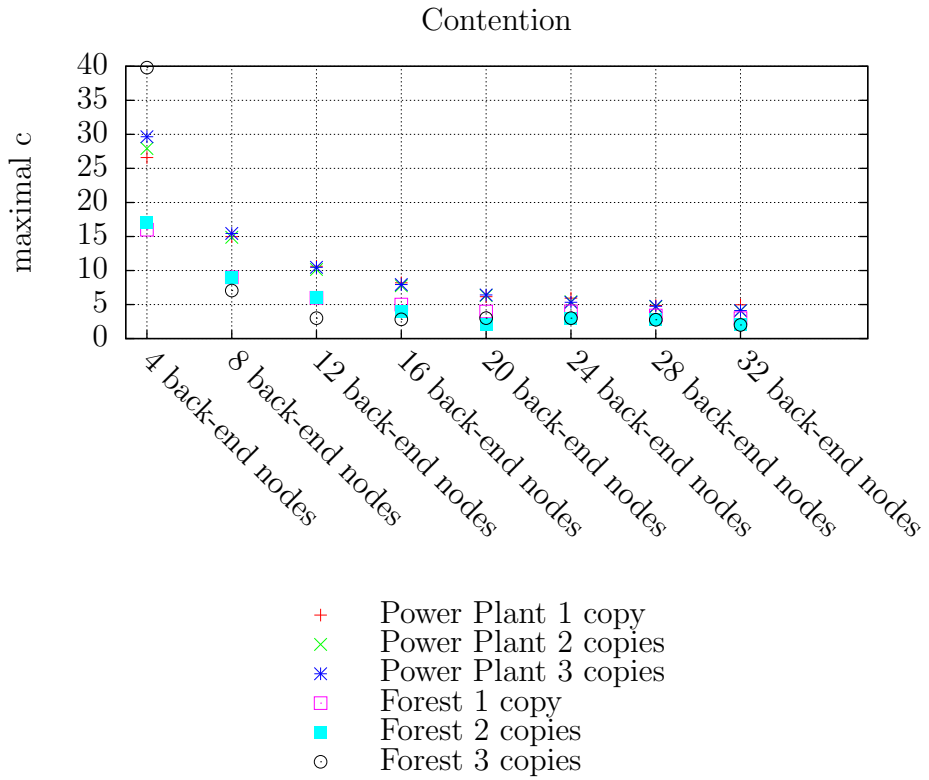


Figure 4.15: Maximal reached contention during the walk-through. With increasing the number of processors, the maximal contention decreases.

Rendering time

One property of our method is that the frame rate is mainly determined by the visualization node's computational power and the properties of the scene (i.e. the number of objects in the viewing frustum, the complexity of the unsuitable objects and the resolution of the reliefboards). The number of back-end nodes only slightly influences the rendering time. Due to an overreaching latency resulting in a high error rate, the original objects are rendered instead of a reliefboard. Figure 4.16 shows that the rendering time using reliefboards can be reduced substantially if many objects are in the frustum.

The average rendering time along the chosen camera path using only frustum culling and the original objects is 43 ms, whereas the average rendering time with applied reliefboards is only about 16 ms, nearly independent from the number of back-end nodes used (the observed difference is below the error threshold of the measurements). Thus, the frame rate for a given scene can be influenced by the hardware configuration of only a single PC, which can be improved by relatively cheap updates without modifying any other node of the cluster.

While testing the Forest scene the frame rate fluctuates widely if only original geometry is rendered. Using reliefboards accelerates the rendering process and reduces these fluctuations.

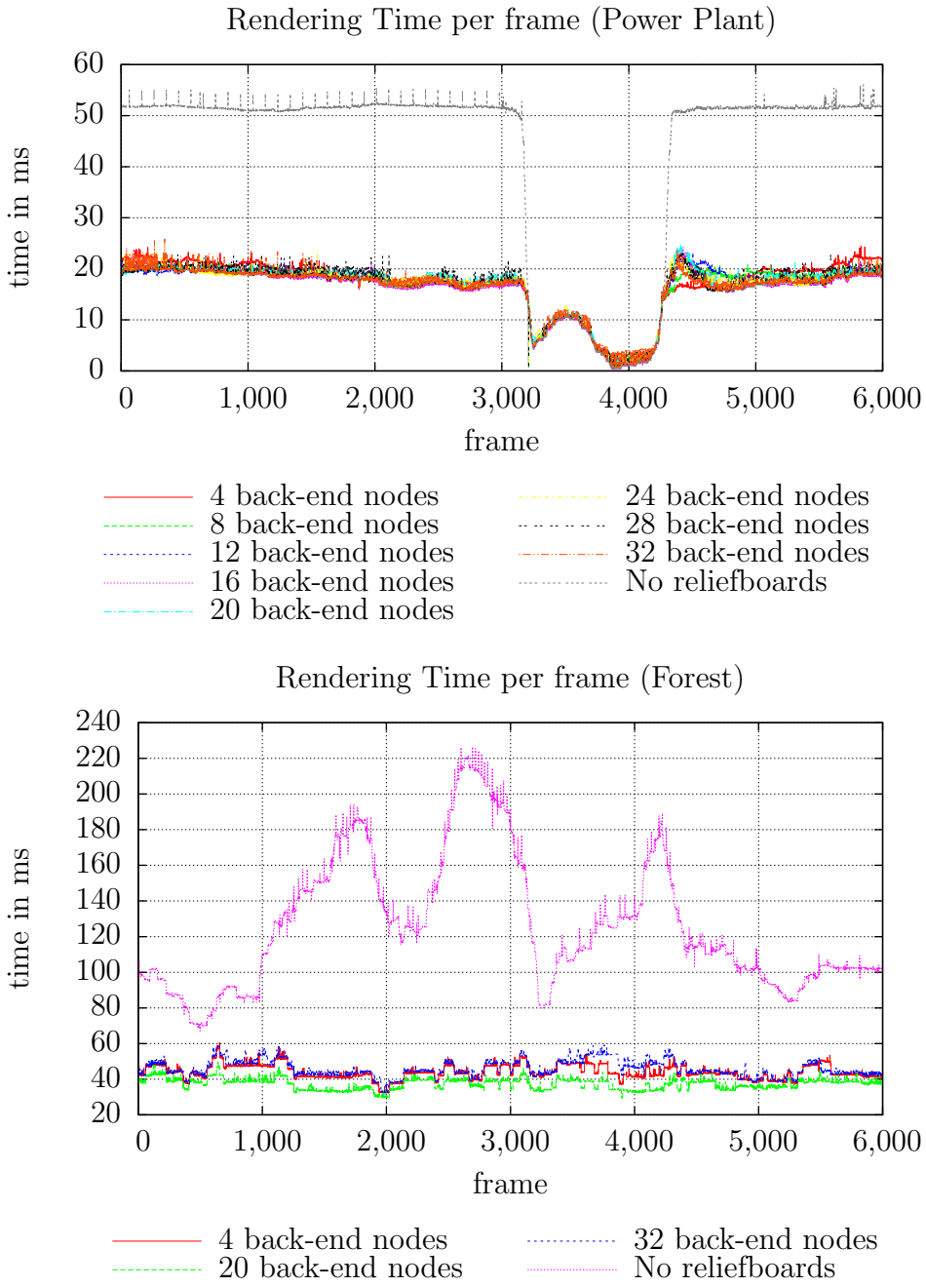


Figure 4.16: The diagrams show the rendering acceleration through reliefboards – top for the UNC Power Plant scene, bottom for the forest scene . The speedup is nearly independent from the number of nodes. The highest lines show the rendering time without reliefboards.

Scene sizes

The complexity of the scene for which the presented method is suitable is determined by several parameters:

- The overall amount of available memory on the back-end nodes needs to be large enough to store the scene redundantly with the chosen number of replicas. In our experimental setting, this was far from becoming a problem.
- The computational power of the graphics adapter of the visualization node mainly determines the achieved rendering time, which is composed of the time needed for rendering the original geometry and for the reliefboards. If the number of objects in the scene gets too large, the frame rate may decrease because too many triangles need to be rendered. In this case the amount of original geometry rendered can be adjusted by lowering the parameter t_{nz}. In this way the total number of rendered triangles is decreased. This reduces the quality but results in additional capacity for the rendering of reliefboards.

In our prototypical implementation, the size of the scene is also limited by the amount of available memory of the visualization nodes' graphics adapters. One should be able to lift this restriction by adding a simple memory management mechanism, which stores the objects that are located near the observer's position or which are likely to get into the near zone within a few frames, in the graphics cards memory.

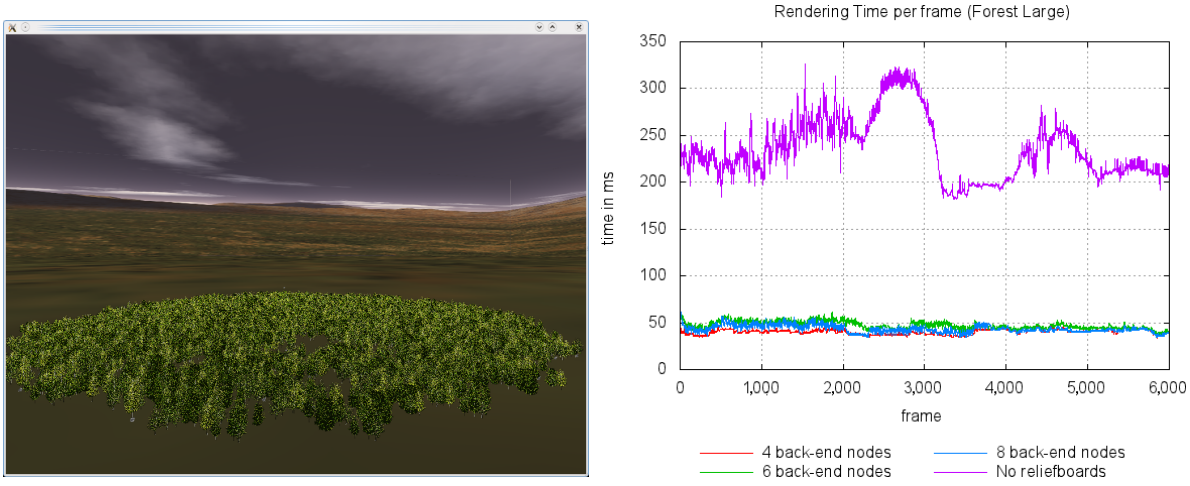


Figure 4.17: The left image shows the large forest scene that consists of $\approx 60\,M$ triangles. The diagram in the right image show the rendering acceleration reached when reliefboards are applied.

Using another hardware setup consisting of nine desktop PCs, we could render another forest scene (see right image in Figure 4.17) with about 60 M triangles with 18 fps by applying reliefboards ($t_{nz} = 3.5\,\mathrm{M}$, 550 objects displayed as reliefboard). In this configuration the visualization node is equipped with an Intel Quad Core 2.8 GHz, 8 GiB

RAM, a NVidia 260 GTX GeForce with 896 MiB graphics memory. Additionally we use multiple standard office PCs as back-end nodes. When solely rendering the original scene on the visualization node without reliefboards we only achieve about 4 fps (see left image in Figure 4.17).

Visual quality

Although the frame rate is nearly independent of the number of back-end nodes, the visual quality is heavily influenced by the response time and therefore by the number of nodes. Generally speaking, a lower response time results in less visual error which can be caused by an outdated reliefboard. Figure 4.18 shows the average distribution of the angle error for the UNC Power Plant scene. The error angle is a good indicator for the image error. The difference in the projected size is not considered as the influence is insignificant due to the chosen camera path. As expected, more nodes decrease the occurrence of errors. For the chosen setting, about 28 nodes seem to be sufficient to achieve a mean error of only 16 degrees (compare to Figure 4.1). Additional nodes are not necessary as almost every request can be processed immediately. The high peaks occur when new objects appear in the viewing frustum. In this case, the angle between an object's normal when its last reliefboard was created and the current viewing vector towards the object can be arbitrarily large.

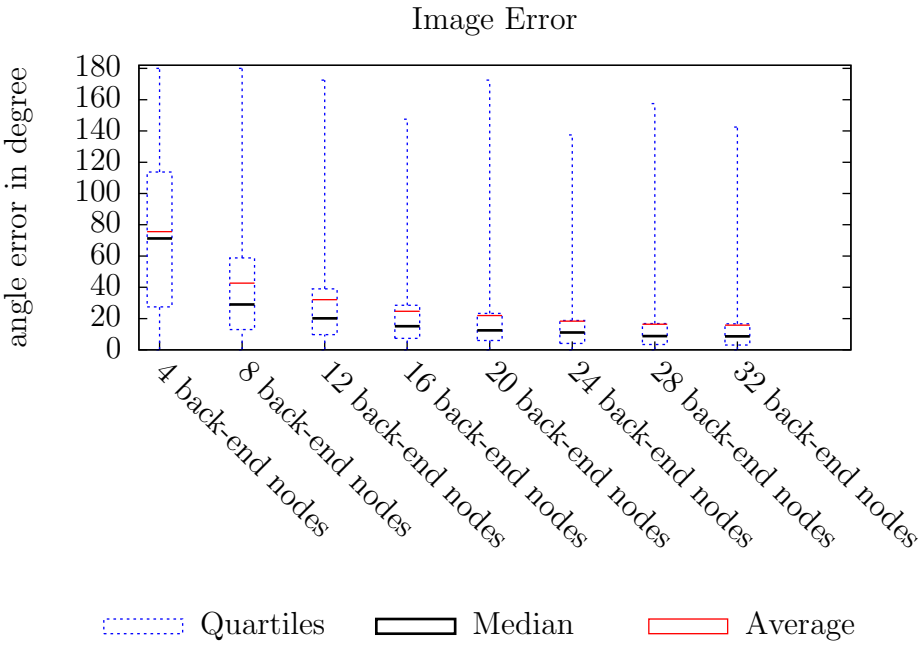


Figure 4.18: Accumulated angle error in the UNC Power Plant scene (with min, max, quartiles, median and average values): The visual quality scales with the number of available nodes.

During the real-time walk-through, a direct pixel based comparison between images where only original objects are used and images where reliefboards are used is not realizable. This is because of the time needed for this comparison (the required time for reading from the frame buffer and storing the images influence the results). Less images are produced in the same time when no comparison is made, so more time available to generate the approximations. As a result the angle in between two updates decreases which influences the image's quality positively.

Differences between images with and without reliefboards, can be seen in Table 4.1.

4.7 Contribution

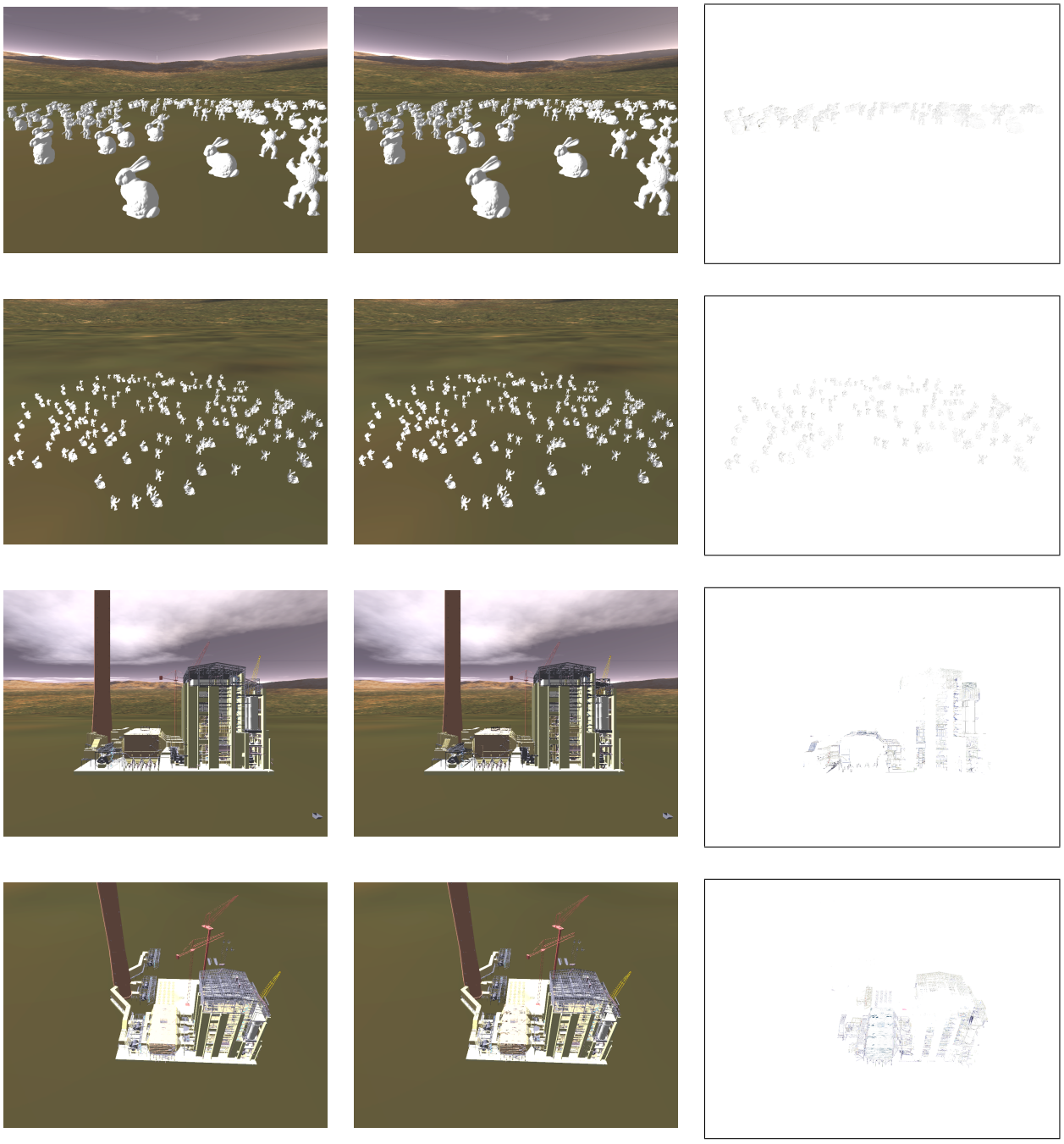
Reliefboards allow the use of a wide range of PC clusters with only limited graphic capabilities for parallel rendering, so that only one powerful visualization node is needed. In contrast to other parallel rendering approaches, this method scales in the quality of the rendered picture instead of rendering speed, when the number of PC cluster nodes is increased.

Using reliefboards allows us to render complex three-dimensional scenes that fit into the visualization node's memory. To apply these approximations, we have to compute the suitable objects for replacement by reliefboards. In order to achieve this we use a heuristic, agglomerative clustering algorithm that builds the required objects in $O(n \log n)$ time.

Because of the scene's size, we were required to apply approximations. These approximations should be easy to compute, low in complexity, and suitable for many camera positions. Reliefboards meet all these needs. These approximations can be computed by computers with weak graphic performance in a few seconds. Their complexity is controllable and limited by the granularity of the grid from which they are generated. The relief properties makes them suitable for multiple camera positions, in contrast to billboards.

To get updates of outdated reliefboards as quick as possible we had to distribute the rendering load evenly among the PC cluster nodes. Here we used the c-collision protocol to achieve this goal. Randomized object distribution results in each node having to process complex objects as well as less complex objects. The redundant distribution can reduce the nodes' maximum contention.

Table 4.1: The images in the left column shows images made with only original objects. Middle column's images are made using reliefboards. The images in the right column displays the differences between the images in the previous columns.



Scenario II:

Static Scenes Which do not fit into a Processor's Primary Memory

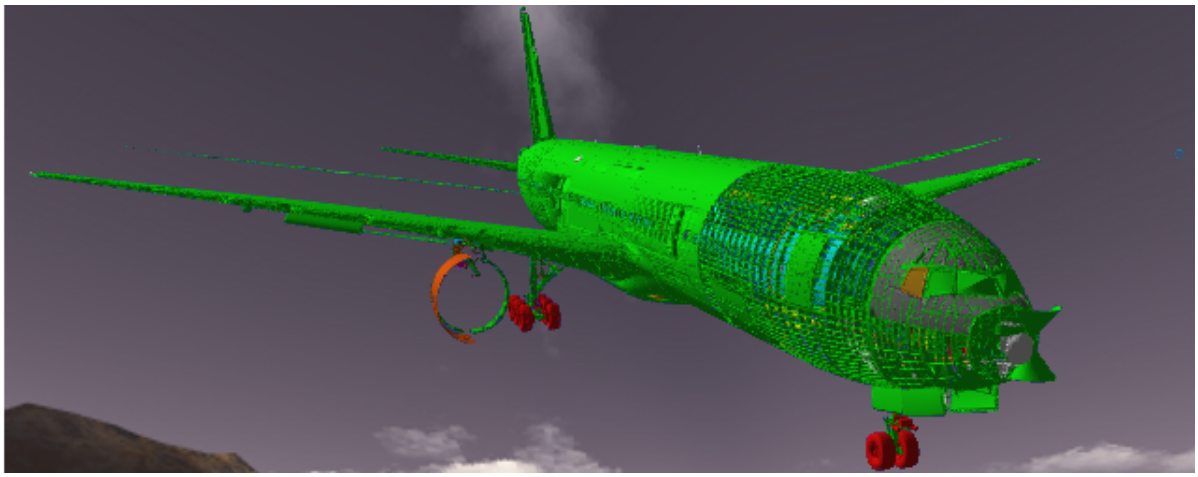


Figure 4.19: Massive Models consist of so many triangles that they do not fit into node's primary memory. This model of a Boeing 777 consists of approximately 350,000,000 triangles and requires more than 8 GiB memory.

Massive Models have such high memory requirements that these scenes do not fit into the primary memory of a single PC (see Figure 4.19). To process such scenes, out-of-core rendering mechanisms are required. In such systems, scene parts are loaded from a secondary memory device to a computer's primary memory when they are needed. Usually the computer's hard disk is utilized to store the scene parts that do not fit into primary memory. The usage of hard disks as secondary memory has advantages for our needs: this memory is cheap, scalable, modifiable, and easy to exchange. The acquisition of new drives can be done easily, in contrast to the primary memory. Usually, if no further memory is available new drives can be added or old disks can be replaced by new ones with higher capacities. The disadvantage of this memory is its speed. For our needs, hard disk latencies are high in comparison to the machine's main memory, not to mention GPU's memory. Distributed data placement, like different RAID systems, can reduce these latencies.

For such Massive Models we remove the requirement of the previous scenario that the complete scene must fit into the computer's primary memory. Here we do not require that all visible objects participate on the final rendered image. Thus, this approach can lead to image errors, but it allows us to render the images in real-time.

We developed two different parallel and approximative rendering algorithms for such large scenes. The first algorithm uses a modified version of the c-load-collision protocol to balance the rendering load and to reduce node's contention. Here we analyze the balancing if we increase the number of copies, distributed among the back-end nodes. Furthermore we tested the influences of randomized and deterministic *tiebreaker* algorithms.

As preliminary work for our second parallel out-of-core rendering system we developed the *hull tree*, a special purpose data structure. The hull tree is a spatial data structure that covers scene objects tighter than common data structures. Additionally, we de-

veloped a sequential rendering algorithm that exploits this data structure. To speedup visibility tests we also use approximations. In this way we reduce the number of triangles sent through the rendering pipeline.

To use the hull tree for the parallel out-of-core rendering system some minor modifications were needed. We combined the hull tree with another hierarchical data structure to improve its properties for our needs. For these systems we evaluate the time required to produce images. Additionally, we analyze the influence of different hardware environments on the rendering algorithm.

Our parallel out-of-core rendering approaches use the main memory of the back-end nodes in a PC cluster as secondary memory. The network adapters in PC^2's Arminius cluster enable faster access to the different data items than regular hard drives. Additionally we can use back-end nodes' computational power to filter the items sent across the network. Nevertheless, due to back-end nodes' weak computational power, we do not request the items synchronously. Similar to our approach for reliefboards we send a request and receive the answers several frames later (see Figure 4.20). At any time only a small amount of the complete scene is stored on a small group of visualization nodes, where the images for the user are produced.

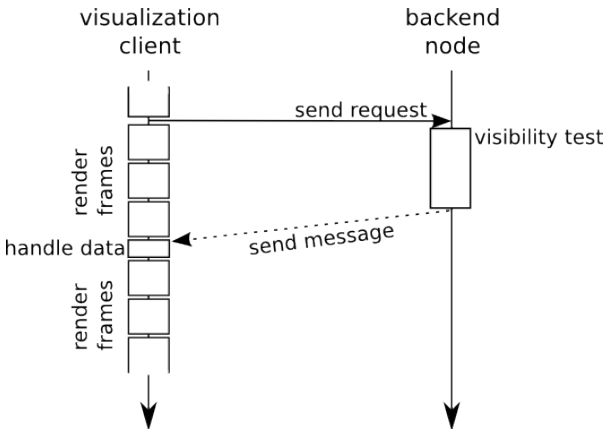


Figure 4.20: Example of the communication scheme for our parallel out-of-core rendering systems. The visualization node sends a request and continues its rendering process, while back-end nodes perform their tests and answer the requests several frames later.

5 Load-Balancing using the c-Load-Collision Protocol

In this chapter we will present a workload balancing technique for heterogeneous clusters, based on the c-load-collision protocol. In the previous part we developed a rendering technique for scenes that can be stored completely in the primary memory. The parallel out-of-core rendering systems presented next do not have this limitation. The system we present in this chapter was designed, implemented, and evaluated in Wiesemann's diploma thesis as a collaboration [Wie10]. The presented approach can be applied when a fast network is available but the data access or the computations on the different nodes is slow. In this work, we focus on the load-balancing of large groups of back-end nodes. The rendering speed itself depends mostly on the rendering performance of the small group of rendering nodes. To evaluate this protocol we implemented a parallel out-of-core rendering system with dynamic load-balancing for huge data sets (larger than 8 GiB). In this system, nodes with graphics adapters that provide a high triangle throughput are responsible for image generation, whereas the back-end nodes serve as secondary storage and for other smaller tasks [SWF10a, SWF10b, Wie10].

The nodes of PC^2's Arminius Cluster are connected via *Infiniband*. The nodes of this PC cluster provide weak rendering performance, but they can be used for other important tasks to support the rendering process. Their main memory can be used as secondary storage in an out-of-core rendering system, instead of hard drives, which are typically used if the main memory resources are exceeded. Unlike the storage systems in other out-of-core rendering systems, these nodes can also perform additional tests and calculations largely autonomously. In order to reduce network's load, we aim to keep the amount of data that is received by these nodes small. To utilize the PC cluster for our out-of-core rendering the system has to fulfill the following requirements:

- The scene should be distributed evenly among the main memories of the back-end nodes.
- The back-end nodes' tests and computations should require little information from the other nodes.
- The visibility calculations should be balanced for most camera positions.
- The back-end nodes' workload should be balanced.
- The amount of data to be sent to the visualization node must be kept small.

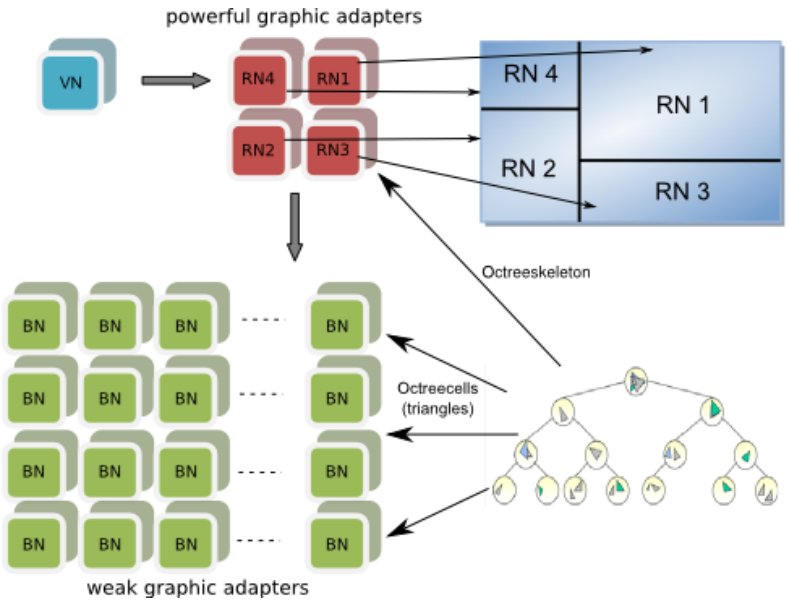


Figure 5.1: Layout of our rendering system, consisting of a visualization node (VN), four render nodes (RN) and several back-end nodes (BN). Each of the four render nodes is assigned to a portion of the screen (tile).

5.1 Overview and Summary of Results

Our system architecture requirements a single visualization node, a small group of powerful render nodes, and a large group of back-end nodes. In this approach, the visualization node does not need a powerful graphics adapter. This node is used to accept user inputs and to send them to the render nodes. Additionally, it receives the partial images from the render nodes, combines them, and displays the final image.

Each render node renders a part of the final frame. Therefore, a node stores only the necessary objects to its piece of the frame. Newly appearing objects which are requested from the back-end nodes are determined using a spatial data structure.

Instead of sending the requested objects directly, the back-end nodes test the objects for visibility. To do this, these nodes use a depth buffer they receive periodically from the render nodes. If an object it tests is visible, it is sent to the inquiring render node.

In this system we distributed the different objects redundantly and randomly. This provides good load balancing for most camera positions. To benefit from this distribution (i.e., to keep contention low and to distribute the load evenly) we use a version of the c-load-collision protocol.
To decrease the number of rendered objects, the render nodes need only a sample tree as spatial data structure. To perform the visibility test, the back-end nodes exploit render nodes' depth buffers, which are updated periodically.

5.2 Related Work

To minimize the delivery time in our parallel out-of-core rendering system fast data access is required. There are various data management protocols for networks, which balance the occurring data requests among multiple data modules. In our sense, a data module is equal to a back-end node. Dietzfelbinger and Meyer auf der Heide present a deterministic approach to simulate a PRAM on different types of *distributed memory machines* [DMadH93]. Using a c-arbitrary DMM, a data module processes at most c of the arriving requests in one step. Using a c-collision DMM, a data module processes all requests if there are at most c of them; otherwise it rejects all. The expected delay to produce an access schedule is bounded by $O(\log n)$ with high probability.

Stegmann developed the first version of the randomized *c-collision protocol* which reduces the expected maximal contention of a data module to a constant c with high probability [Ste96a]. A known issue for this protocol is that there exist situations where it cannot terminate for a given c. A rule of the protocol says that each node answers at most c requests. If all nodes receives more than c requests, none will answer. To resolve that problem, c could be increased until the protocol terminates.

The c-collision protocol can be interpreted as *balls into bins* game. In the previously described versions, the balls' weight is uniform. In contrast, Berenbrink et al. introduced a protocol for weighted balls [BMadHS97]. In their *c-load collision protocol*, for every request, three data modules are randomly chosen. Each request is sent to its chosen data module, which answers all incoming requests if and only if their sum is at most c.

5.3 The Parallel Rendering System

In the following subsection we will describe the developed rendering system in more detail. Our system requires a visualization node, render nodes, and back-end nodes as described in the previous section (see Figure 5.1). To allow for sufficiently fast data transfers we require a fast network. In our setting all nodes are connected through Infiniband network adapters. Instead of the usually used hard drives as secondary memory, our rendering system uses the main memory of the different back-end nodes. The secondary memory stores the currently unused scene parts that cannot be stored in the primary memory.

The visualization node has four different jobs: displaying the rendered images, balancing the render nodes' load, organizing the render nodes' data requests, and initiating the updates of the back-end nodes' depth buffers. The visualization node receives, combines, and displays the different frame parts from the render nodes for each frame (see Figure 5.3). To balance the individual render nodes' load, the visualization node receives the time it took to generate partial images from the render nodes. The task sizes of the render nodes are modified based on these times, similar to the approach of Abraham et al. [ACCC04]. To organize the data requests, the visualization node receives these requests from the render nodes and calculates an assignment using a variant of the c-load-collision protocol (see Section 5.4). We choose the c-load-collision protocol for

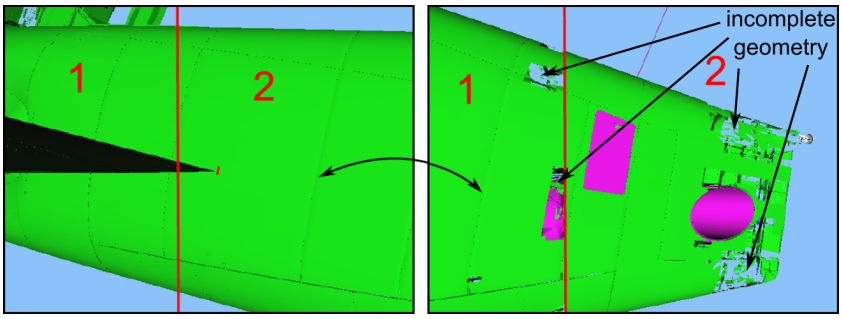


Figure 5.2: The images show the tail of a Boeing 777 from the same position, but different view angles. When the camera is moved rapidly, image-errors can occur. Both images show a tiled frame produced by render nodes 1 and 2. The right image shows missing parts, caused by delayed objects.

load balancing because it assigns requests to nodes while considering all other requests. Additionally, in general this protocol can be executed via network without keeping a centralized data structure. To allow the back-end nodes to perform a worthwhile occlusion culling, the visualization node periodically initiates depth buffer updates.

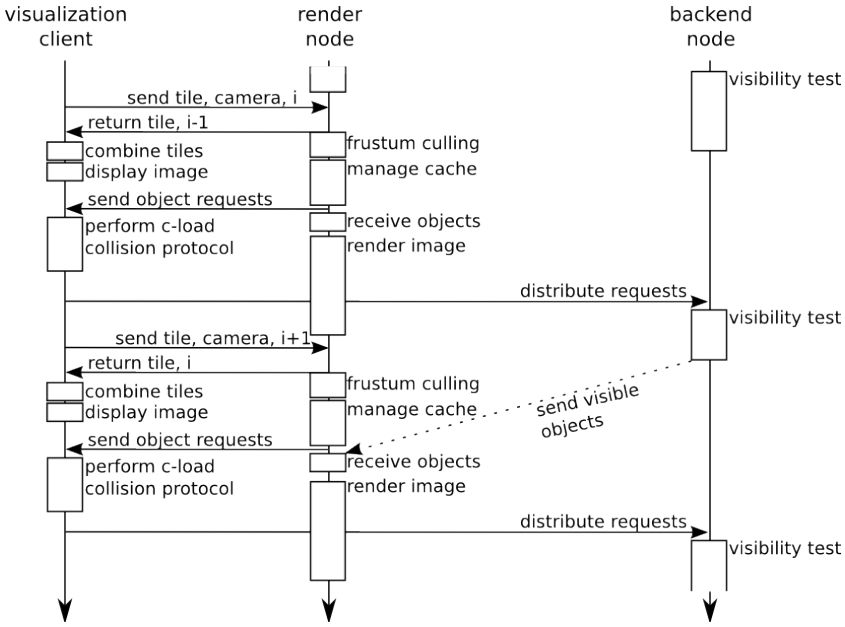


Figure 5.3: Communication flow to produce an image on the visualization node.

The render nodes' job is to render partial images of the scene. The dimensions and positions of their tiles are received from the visualization node, as are the current camera settings. After receiving the needed data, the nodes align their tile and camera settings (see Figure 5.1). Next they perform a frustum culling on their extended viewing frustum (see Figure 2.2) and reorganize their object cache. If the frustum test shows the potential requirement of objects, which are not stored in their memory, a node sends a request to

the visualization node. Due to this asynchronous communication objects arrive delayed, which can result in temporal image errors (see Figure 5.2). The render nodes check if there are new objects available before they render a frame. During rendering, the back-end nodes perform occlusion culling to determine invisible objects. These objects are stored in the object cache for potential reuse. After rendering their partial image, each render node copies it from the GPU-memory to its main memory and sends it for displaying to the visualization node. If initiated by the visualization node, the render nodes transmit their local depth buffer to the different back-end nodes.

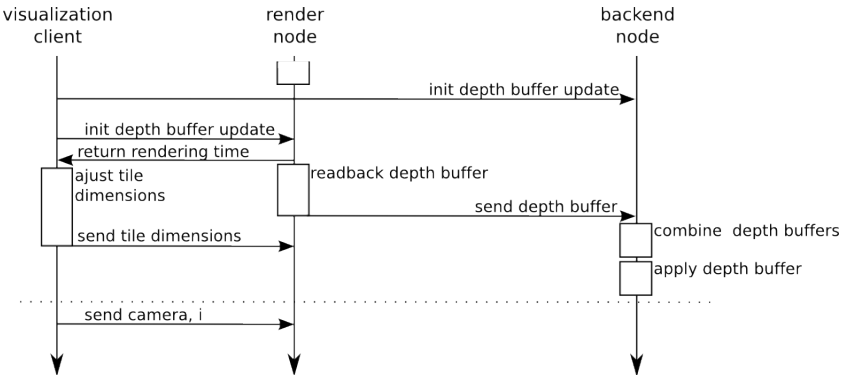


Figure 5.4: Communication flow to update the render nodes' tile dimensions and back-end nodes' depth buffers.

The back-end nodes serve as *smart* secondary memory. Instead of sending objects untested to a render node, they test objects' visibility first. Each node stores only a subset of the scene's objects. It is difficult to perform occlusion culling with only a subset of the entire scene. Due to the missing occluders, objects might be classified as visible, while they are actually hidden in the final image. For this reason, the back-end nodes periodically receive render nodes' depth buffers and combine them (see Figure 5.4). During their own visibility tests, back-end nodes update the received depth buffers successively. If they test an object and it is visible they send its data to the requesting node. Otherwise, they send nothing.

The load-balancing protocol used requires that all data items are distributed redundantly and randomly among the back-end nodes. Because the scene is too large to fit into the main memory and it is not possible to render it in real time, it has to be approximated and distributed.
For this purpose we use a sample tree data structure [KKF+02]. The sample tree partitions the scene into smaller subsets that are distributed across the network (see Figure 5.1). In the course of this chapter, an object refers to all of the geometry contained within a single sample tree cell. For fast data access, we use a modified version of the the c-load-collision protocol. A request passed into this protocol is equivalent to a request for one of these objects. Based on this protocol, the back-end nodes' load is balanced. Although there are other approaches (e.g., Levels of Detail [LWC+02]) the disadvantage of these is that surface properties usually are altered. In this way, unwanted gaps between

objects can appear and influence image quality and collision-detection algorithms. To produce a frame, the view-plane is separated in multiple tiles and each tile is rendered by a different render node (see top, right in Figure 5.1). These nodes are the high-capacity cluster nodes, which are equipped with modern graphic adapters. If needed data-items are not stored in a render node's local memory, a data request is send to the visualization node. Instead of waiting for the requested data-items, the render node renders its frame without the missing parts. In this case, errors can occur in the final image (see Figure 5.2). After rendering a tile, it is send to the visualization node.

In our environment each render node is equipped with 4 GiB of RAM, but our benchmarking model of a Boeing 777 consists of roughly 350 M triangles (≈8.5 GiB), which is more than twice the size of the primary memory.

The back-end nodes are utilized in two ways: due to their lack of graphics adapters, they serve as secondary memory and assist the render nodes by performing occlusion tests. If the tested object is visible, its data will be sent to the requesting render node. Otherwise, the request is discarded without notifying the requesting render node.

5.4 Load-Balancing Algorithm

The c-collision protocol allows for fast data access in a network, where the different data-items are distributed randomly and redundantly. Thereby the contention of different back-end nodes is limited by the constant c with high probability [Ste96b]. The protocol works as follows: each request is assigned to all back-end nodes that store a copy of this item. If a back-end node has at most c assigned requests, it replies to all of them and removes the answered requests from the other back-end nodes. If a back-end node gets more than c requests, it answers none of them. This scheme is repeated for several rounds until all requests are served (see Figure 5.5). It is assumed that all requests have equal weight. For example, weights can be the required processing time or the message size.

There are different situations in which the requests are weighted; these weights do not affect the c-collision protocol. Berenbrink et al. present in [BMadHS97] the c-load-collision protocol. This protocol is a modified version of the c-collision protocol that respects the requests' weights by involving them in the request assignment. The given load limit c can only be held if all tasks have been finished before new requests arise. It is impossible for this condition to hold if the protocol is used to balance the load of a parallel out-of-core rendering system, which is supposed to offer sufficient frame rates. Waiting until all requests have been answered would cause serious speed reductions in the rendering process. Thus, an asynchronous communication scheme is needed that allows for overlapping rounds. In our system, we investigate how two different strategies for choosing responsible nodes affect the load-balancing situation. To do so, we developed a randomized and a deterministic function for node picking. Furthermore, we evaluate how overlapping rounds affect our version of the c-load-collision protocol.

Similar to the regular c-collision protocol, all data-items are distributed redundantly and randomly across the different back-end nodes. This assignment does not change

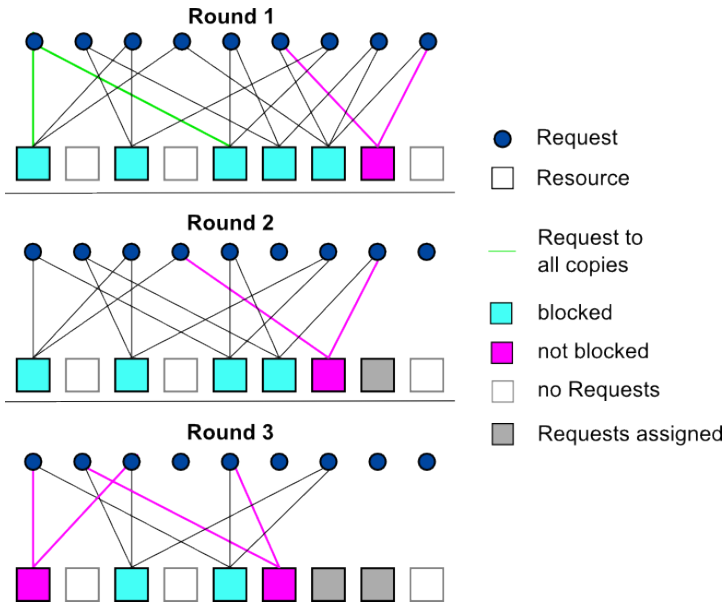


Figure 5.5: Three rounds of the c-collision protocol. A node with assigned requests will not answer again.

during runtime. The render nodes send their requests to the visualization node instead of sending them directly to the responsible back-end nodes. In every frame the visualization node receives all new requests of the render nodes and calculates an assignment of tasks to the back-end nodes. By using the protocol, we try to distribute the network load evenly among the back-end nodes. Instead performing the c-load-collision protocol on the real network it is simulated on the visualization node. This way no additional communication is needed and the balancing-time is reduced.

The pseudo-code for our implementation of the c-collision protocol is shown in algorithm 5.1. In every frame, the set of requests is divided into subsets that contain at most the same amount of elements as the number of available back-end nodes. The weight of a request is equal to the requested object's number of triangles. Using triangles for request weighting is convenient because the triangle amount influences the needed time for the occlusion tests as well as the size of the data-item to be sent.

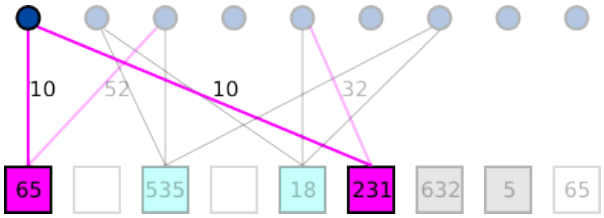


Figure 5.6: The two magenta colored back-end nodes can answer the same request. It has to be decided which one is responsible.

If a data request r can be answered by more than one back-end node, it has to be decided which back-end node will be responsible for the request (see Figure 5.6). In order to balance the work-load evenly across the nodes, we first calculate the sums of the weights of all active jobs on all nodes. The function to calculate the sum of the weights of a node PE is defined as $s(PE_i)$. The list of nodes is sorted in ascending order of these weighted sums and is processed in that order.

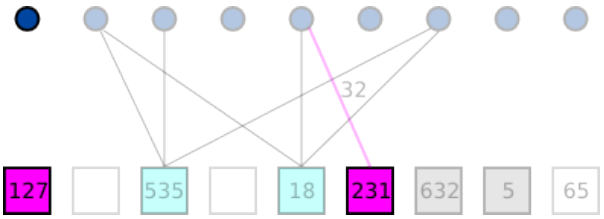


Figure 5.7: The selected strategy chose the node with the lower load to process the request.

We analyze two strategies, a deterministic one and a randomized one. Both strategies choose the responsible back-end node with respect to their load (see Figure 5.7).

- Our deterministic strategy works as follows: It always picks the first node with the smallest $s(PE_i)$, which can handle all its requests (see Algorithm 5.2).
- Our randomized strategy works as follows: We pick a node randomly from all nodes that could handle the request r. To influence the probability at which a node is chosen, we utilize the weight sums of the nodes in question (see Algorithm 5.3).

Let $\hat{P}$ be the set of back-end nodes able to handle request r. Then

$$T = \sum_{PE_i \in \hat{P}} s(PE_i)$$

is the total sum of the accumulated weights. The probability to choose node PE_i is defined as:

$$Pr(r \text{ is handled by node } PE_i) = \frac{T - s(PE_i)}{(|\hat{P}| - 1) \cdot T}$$

The probabilities of each node equate to a unique area in the unit interval. To assign a data request to a back-end node, a pseudo-random-number between zero and one is generated. The generated number always lies within an interval associated with a back-end node. This node is chosen to handle the data request.

Algorithm 5.1 c-collision protocol

Variables: nodes P, $|P| = n$,
request sequence $S = (s_0,s_1,\ldots,s_{l-1}), l = i \cdot n$.

1: $c = 2$
2: $S = (r_0 \oplus \ldots \oplus r_{i-1})$, $r_j = (s_{j \cdot n},\ldots,s_{(j+1) \cdot n-1})$,
$0 \leq j < i$
3: **for all** $r_j \in S$ **do**
4: **while** $\exists s_k \in r_j$ that is not answered **do**
5: **if** at least one request of r_j can be answered **then**
6: **for all** s_m that can be answered **do**
7: `pick` node $PE_a \in N$ that handles s_m
8: increase load of node p_a by the weight of s_m
9: mark request s_m as answered
10: **end for**
11: $c = 2$
12: **else**
13: $c = c + 1$
14: **end if**
15: **end while**
16: **end for**

Algorithm 5.2 Deterministic picking strategy

Variables: request s_u to handle

1: determine $\hat{P} = \{\hat{PE}_0,\ldots,\hat{PE}_m\}$ for s_u and T
2: **for** $k = 0$ *to* m **do**
3: determine $w_k = Pr(\hat{p}_k)$
4: **end for**
5: assign s_u to $\hat{p}_k$ with smallest w_k

Algorithm 5.3 Probabilistic picking strategy

Variables: request s_u to handle

1: $w_0 = 0$
2: determine $\hat{P} = \{\hat{PE}_1,\ldots,\hat{PE}_m\}$ for s_u and T
3: **for** $k = 1$ *to* m **do**
4: determine $w_k = Pr(\hat{PE}_k) + w_{k-1}$
5: **end for**
6: Choose a random number $0 < R \leq 1, R \in \mathbb{R}$
7: assign s_u to $\hat{p}_k$ with $w_{k-1} < R \leq w_k$

5.5 Evaluation

For our tests we use a model of a Boeing 777. The model's size $\approx$ 8.5 GiBs, thus it does not fit into the main memory of the render nodes. As test environment we used four

render nodes and up to 32 back-end nodes of PC^2's Arminius cluster. Here we run our parallel out-of-core rendering system.

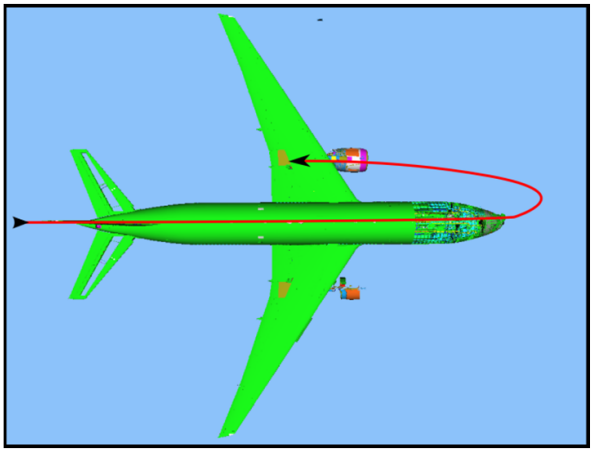


Figure 5.8: Walk-through for our tests.

Walk-throughs and load balancing tests in a real rendering system are limited to at most 32 back-end nodes. To perform load balancing tests with an arbitrary number of back-end nodes, we implemented a simulation system for the c-load-collision protocol. The input for the simulation system is a sequence of data requests, generated during a walk-through in the real rendering system using 32 back-end nodes. In Figure 5.8 the test path is visualized.

In our tests, we evaluate the level of load-balancing that can be achieved using the c-collision protocol with our modifications. Due to the randomness of the c-collision protocol, we perform multiple test runs on the system, while using different seeds for the pseudo-random-number generator. In the start-configuration of the c-collision protocol we set $c = 2$. We averaged the results of these different runs. For each position of the walk-through, we measure the load on the individual back-end nodes. The accumulated amount of triangles of processed objects serves as load indicator for the back-end nodes. To achieve comparable results at the individual walk-through positions, the results are normalized. Three different numbers of back-end nodes were used to accomplish the tests: 32, 80 and 120 back-end nodes (see Figure 5.9 - Figure 5.11).

For our tests, at each position in the walk-through, we measured how many triangles each back-end node has to process and normalized the results. We determined the median (red marks) and the 0.1/0.9 quantiles (blue marks) of the measured values.

For each test, our measurements are displayed in three diagrams: the first one shows the load-balancing of the back-end nodes at every position in the walk-through; the second one shows the relation between a given work-load and the number of data requests. The third diagram combines the previous diagrams in a 3D diagram.

While all values are sorted by the walk-through position in the first diagram, in the middle diagram all values are sorted by the number of request per frame. The last

diagram combines both views. The 3D diagrams' y-axes are equally scaled to visualize the benefit of an increased number of back-end nodes.

This sorting gives a hint to the origin of the strong peaks in the first diagram. If we only have a few requests to distribute among many back-end nodes, stronger deviations occur. In this situation, some nodes do not get any assignments at all, while others still get a few. Here, even the median value is close to zero. The requests we use are atomic and can not be distributed to multiple nodes. These peaks do not corroborate the conclusion that there are overloaded nodes, there are just not enough requests available to employ every single node.

In all the diagrams the deviation of the load is generally low. As mentioned above, the difference of the load is larger if only a few requests are generated. The more requests we have at a given position, the smaller the difference between the median and the quantiles. If the number of back-end nodes is increased, this difference is even less significant. The median value is generally close to $1/p$, where p is equal to the number of back-end nodes, which leads to the conclusion that the load is distributed evenly.

Next, we investigate how the redundancy influences the load-balancing. To accomplish this, we increased the number of redundant copies of the objects in the network from one to two. Our measurements show that the load can be distributed more evenly if the number of copies is increased. Thus, the quantiles lie closer to the median within the diagrams (see Figure 5.9 and Figure 5.12).

Next we distribute three copies of the elements among the back-end nodes and measure how this influences the load balancing 5.13. The results of these tests show that there is no significant difference between two and three copies of the elements distributed across the PC cluster.

Altogether our tests show that our version of the c-load-collision protocol can achieve good load-balancing if a sufficient number of requests are generated. This can be observed in all diagrams: the strongest deviations occur whenever the number of requests per position is very low, which leads to the conclusion that the amount of visible geometry in these positions is marginal.

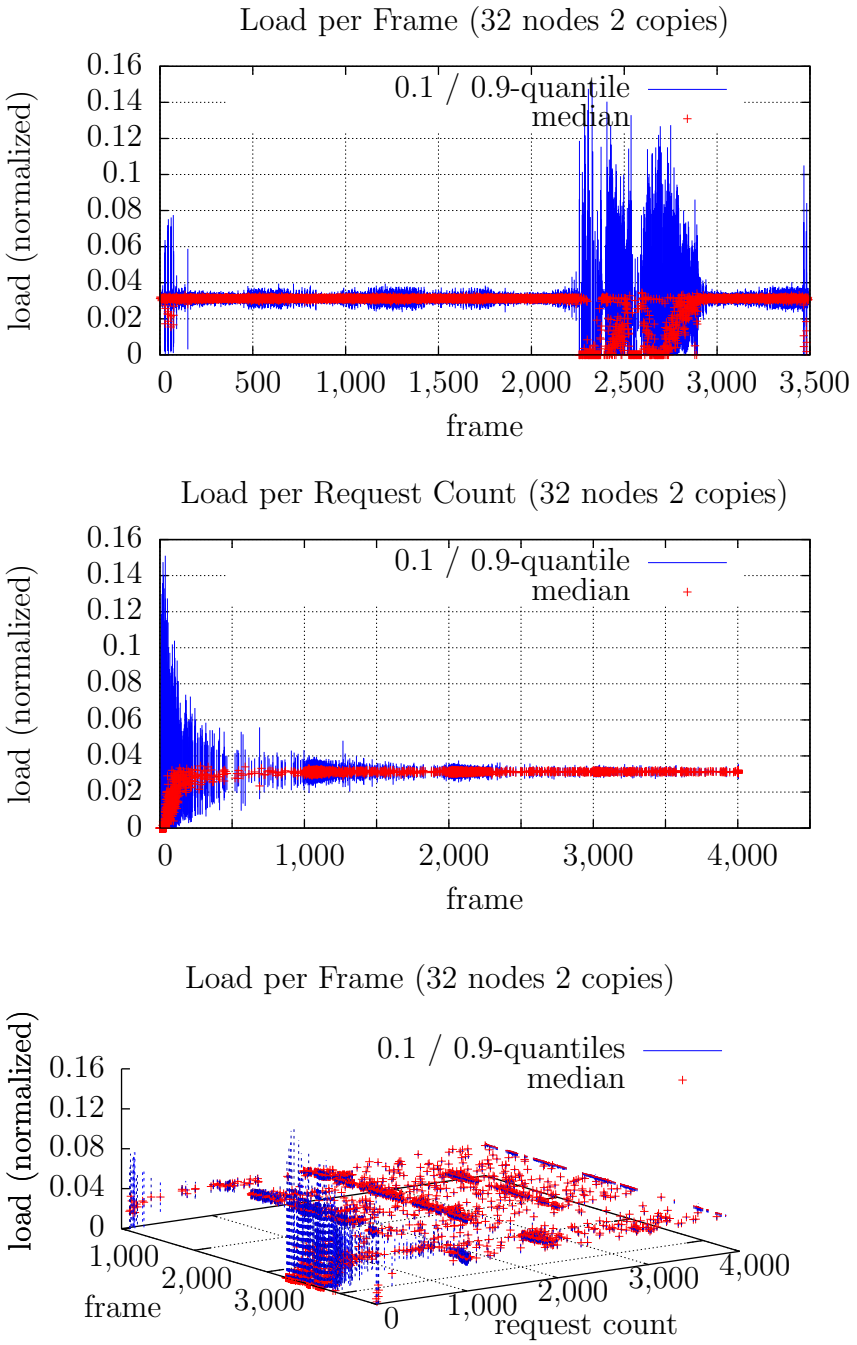


Figure 5.9: Walk-through with 32 back-end nodes and two copies of each element.

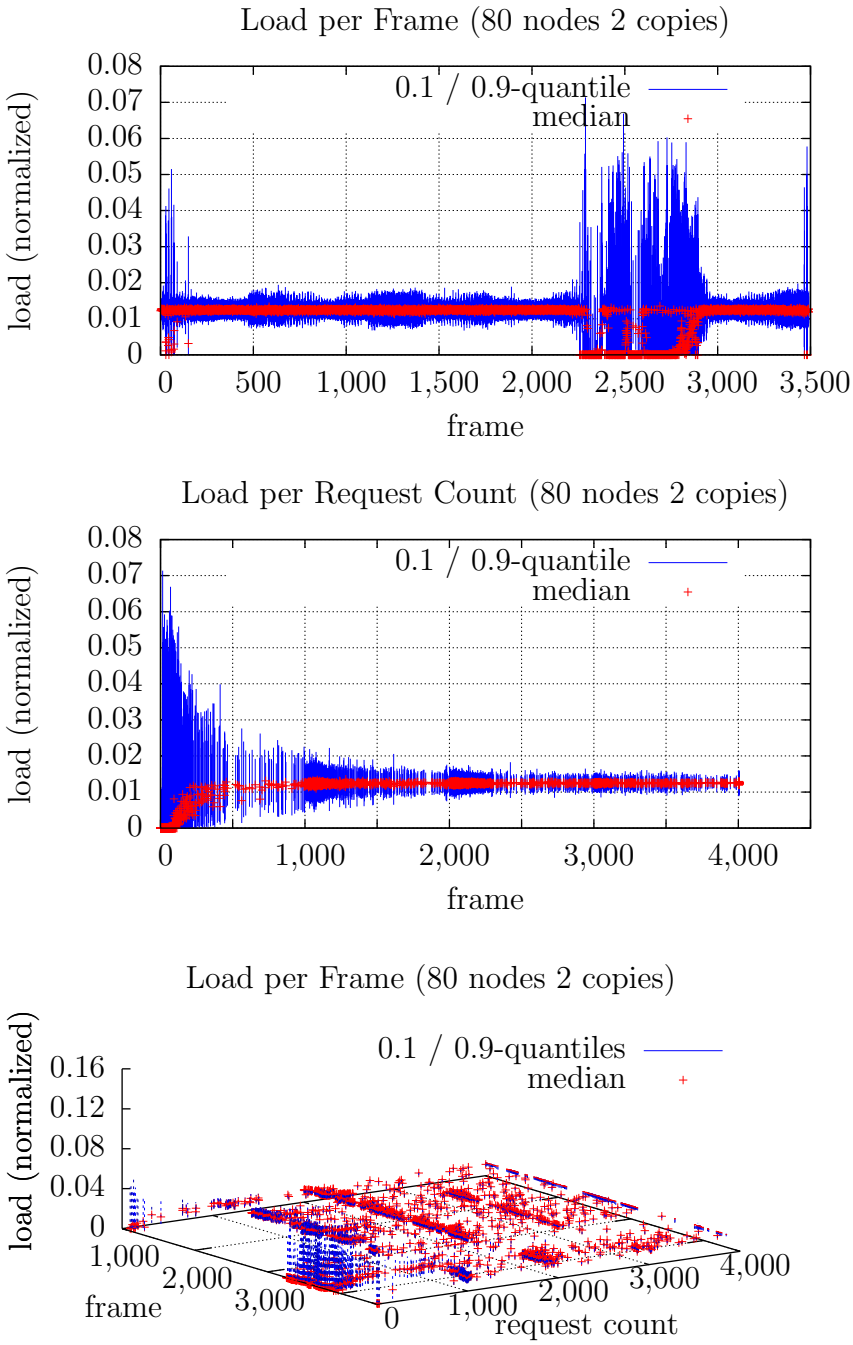


Figure 5.10: Walk-through with 80 back-end nodes and two copies of each element.

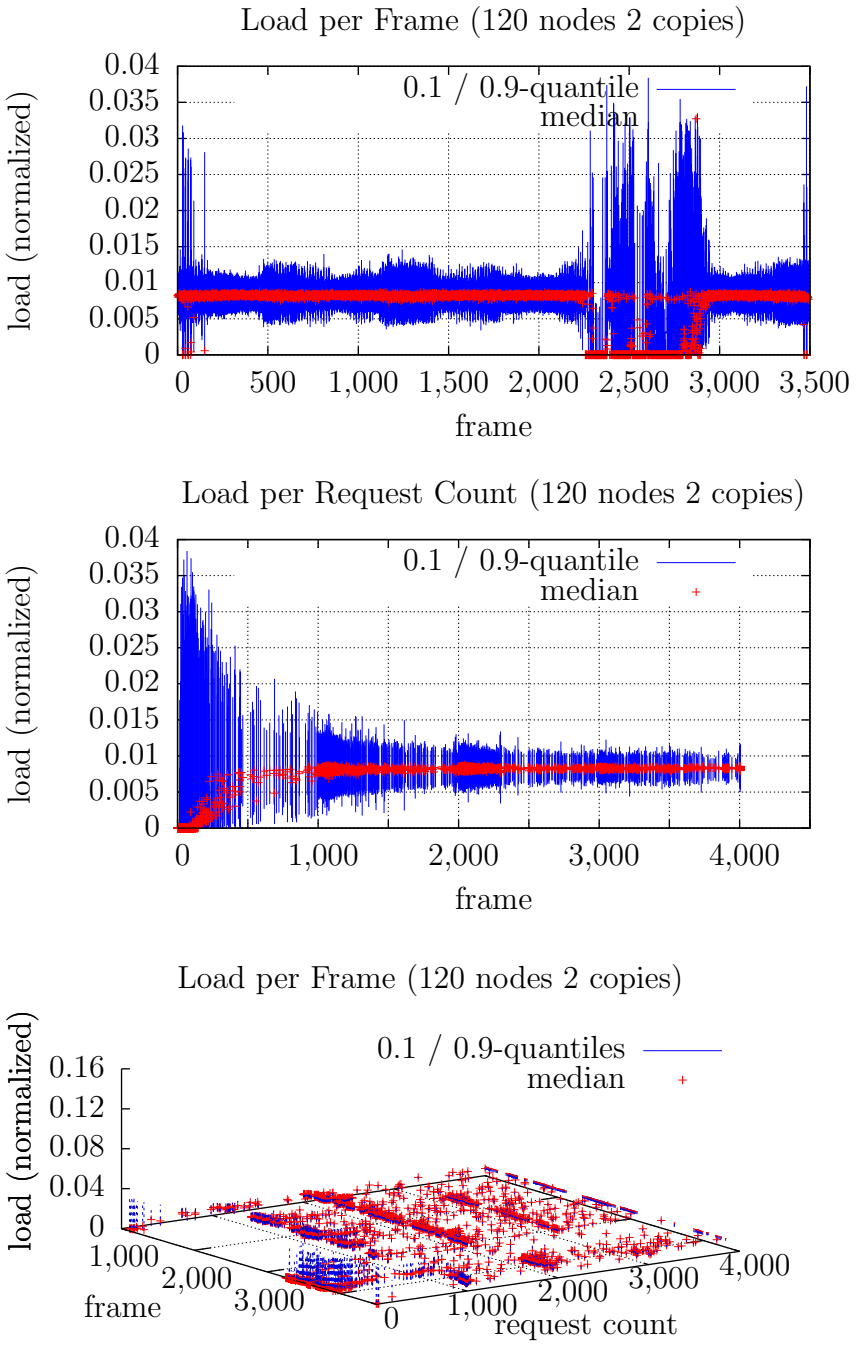


Figure 5.11: Walk-through with 120 back-end nodes and two copies of each element.

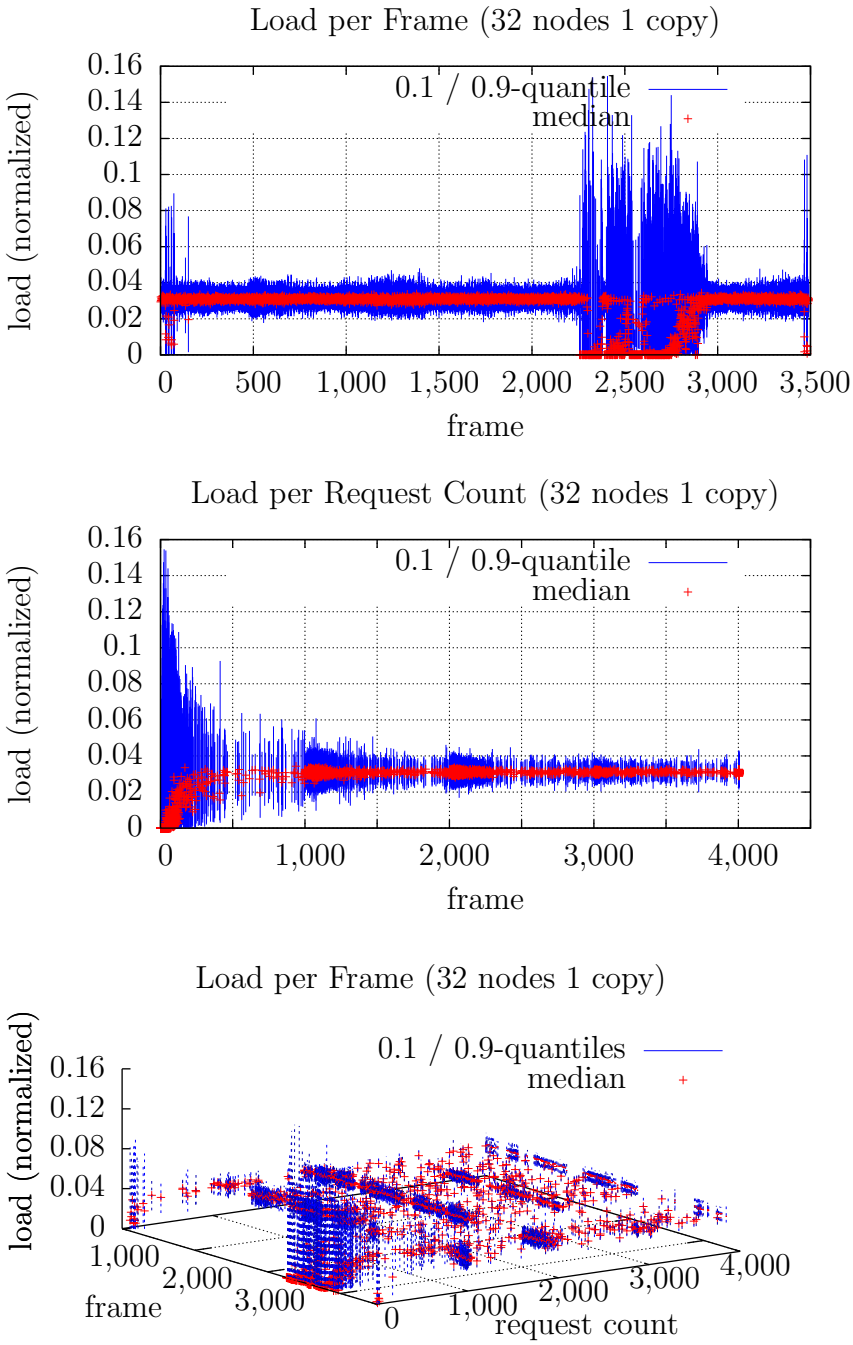


Figure 5.12: Walk-through with 32 back-end nodes and one copy of each element.

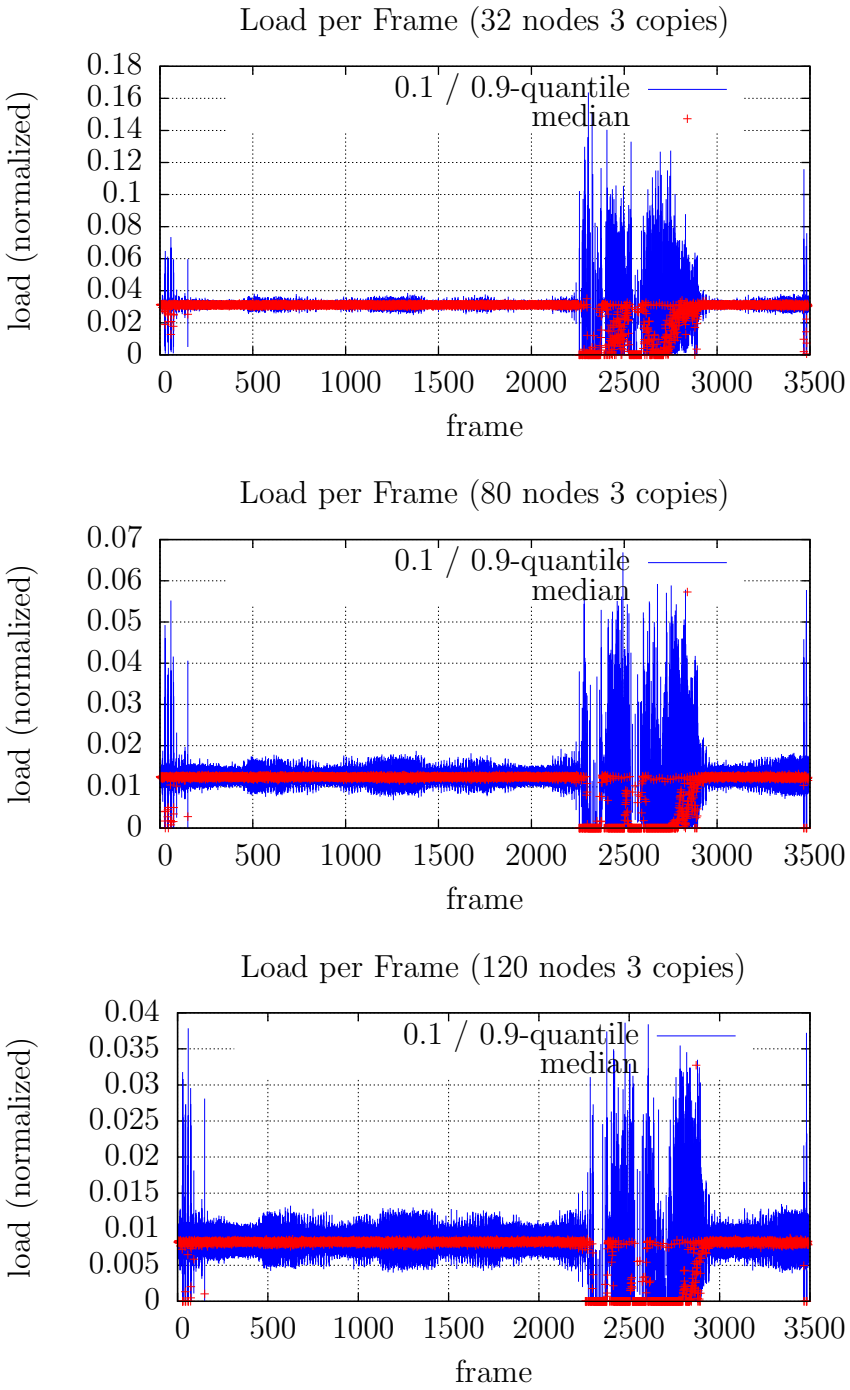


Figure 5.13: Walk-through with 32, 80, and 120 back-end nodes and three copy of each element.

Finally, we studied the differences in the balancing situation when using the randomized picking strategy and when using the deterministic picking strategy for the c-load-collision protocol, described in Section 5.4. Generally we could not find any significant differences when using either strategy. Only marginal differences could be found when reducing the amount of back-end nodes to four. In Figure 5.14, a comparison between two measurements was made: for every position of the walk-through we calculated the difference of the results in deterministic and probabilistic methods, and plotted the absolute value of this difference into the diagram. Most of these results are close to zero. In the test series spikes occur when there are only a few requests. Whenever stronger deviations occur, the differences of the two approaches get stronger as well.

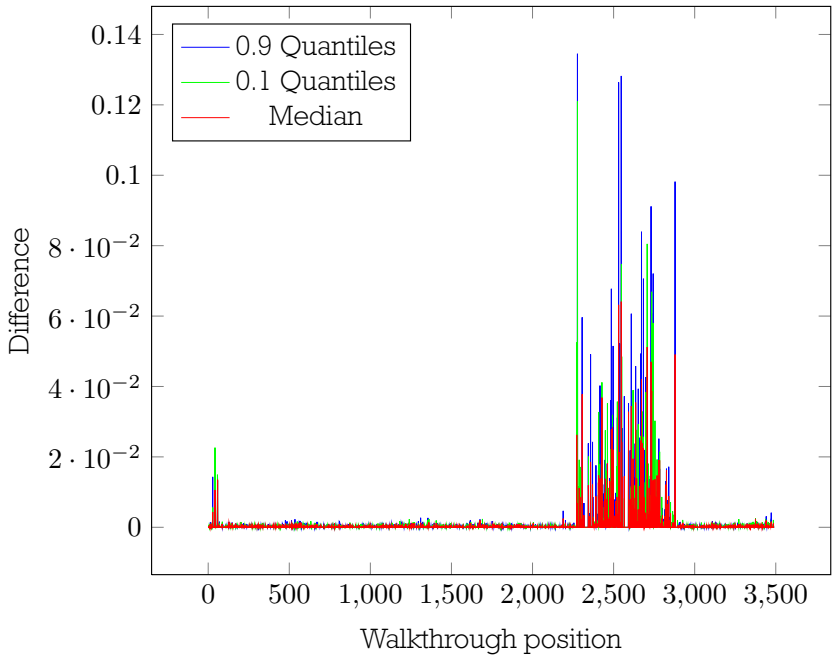


Figure 5.14: Differences between randomized and deterministic load-balancing.

5.6 Contribution

For its effectiveness in an interactive environment we tested a variation of the c-load-collision protocol. We implemented a parallel out-of-core rendering system with the c-load-collision protocol as a work load balancer. In the course of our tests we were able to show that the protocol's balancing capabilities are sufficient for this kind of application, although there were no significant differences between two and three copies of the elements distributed over the back-end nodes.
Through the occlusion culling performed on the back-end nodes, we reduced the amount of data sent to the visualization node. This increased the speed of transmitting of required objects.

The occlusion culling on the back-end nodes uses the older depth buffers of the render nodes. Due to the older depth buffers the occlusion culling is not always correct, but by exploiting spatial coherency, the tests yields good results.
In the version we presented, there are only three node types available. Slight modifications to the c-load-collision protocol used would allow a wider variety of back-end nodes to be handled and balanced. Therefore the different nodes' computational power could be included in the distribution computations of the data management protocol.

6 Hull Tree

In this chapter we present a data structure and a sequential rendering system, which is a preparatory work for our second parallel out-of-core rendering system. To render highly complex scenes in real-time, techniques are necessary to prevent objects from being sent into the rendering pipeline if they have no effect on the final image. Invisible objects must be recognized as early as possible to reduce the load. Culling, especially occlusion culling, is a common approach to reduce the number of triangles and/or objects that are sent into the rendering pipeline. However, during these visibility tests many occluded objects are sent into the rendering pipeline. We developed the *hull tree* to reduce the number of those objects, the so-called false positives. Additionally we designed a rendering method that exploits this data structure.

Hardware vendors offer *occlusion queries* to accelerate visibility tests using the GPU. The bounding volumes of all potentially visible objects or object groups are tested for visibility in front-to-back order. Only if a fraction of the bounding volume, typically an axis-aligned bounding box, is identified as visible, the associated objects are rendered. To reduce the time spent on these tests, hierarchical data structures (like an octree) are often used to test the visibility of whole object groups located in the same subtree.

Using axis aligned bounding boxes (AABBs) for occlusion tests have mixed results. On the one hand these representations have a very low complexity and are therefore easy to calculate and quick to test. On the other hand – depending on an object's projected shape – these approximations can be too conservative. Due to large empty spaces in the bounding volumes, even complex objects can be classified as visible although they are occluded. So the chosen outer approximation of an object (or a group of objects) can influence the efficiency of an occlusion culling algorithm.

To improve and accelerate the occlusion culling our rendering approach should fulfill the following requirements:

- Objects' bounding volumes must be tight to reduce the number of false-positives.
- Bounding volumes should be organized in a hierarchical data structure to test objects' visibility quickly.
- The rendering algorithm's occlusion culling must exploit the data structure to accelerate rendering process.

The hull tree is a hierarchical data structure of simple approximated exterior object hulls. This hull tree offers a much tighter coverage of the objects in the scene, while introducing only a small increase in the geometrical complexity compared to a corresponding

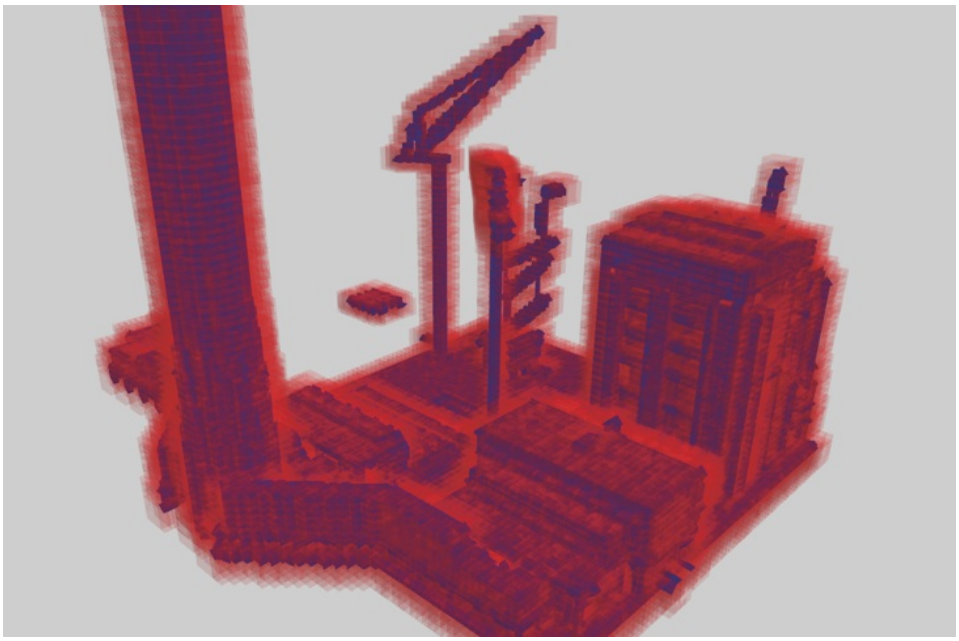


Figure 6.1: Exterior approximations of the UNC Power Plant scene. The transparent red cubes visualize an octree's cells at one level. The included dark blue polyhedrons show the much tighter shapes from our hull tree at the same level.

ordinary octree with AABBs (see Figure 6.1). In our tests the total complexity increase was limited by the factor 1.02 [SKJF11].

An additional challenge for current occlusion culling techniques is that the order in which the objects are rendered is determined by the traversal order of the data structure. For high visual quality, objects can have numerous different shader, materials, textures, etc., attached. Even if only one of these properties is changed in between the rendering of subsequent objects, a state-change in the GPU is induced, which can lead to a deceleration of the rendering process. One occlusion culling algorithm in which this problem is addressed is the CHC++ algorithm by Mattausch et al. [MBW08]. There, the objects are rendered in multiple batches that can each be sorted individually by the objects' properties. This can already reduce the number of necessary state changes. We propose an approximative occlusion culling algorithm (as defined by Nirenstein et al. [NBG02]), which not only sorts small batches of objects, but sorts all rendered objects at once according to their state. Consequently, the number of state changes can be reduced massively. The presented approximative occlusion culling algorithm uses the hull tree for the determination of visibility at a fine granularity. The occlusion information is created using the original objects and, additionally, an automatically generated inner approximation of the scene's objects that can be rendered more efficiently. While it can take a few frames until the complete visibility is determined (which results in the possibility of temporal image error) our rendering technique makes very efficient use of the graphics pipeline with only few state changes and pipeline stalls.

6.1 Overview and Summary of Results

For our rendering algorithm, we first preprocess the scene consisting of predefined objects (meshes) (Note that reasonable objects should not be too big, in relation to scene's overall complexity, for our technique to work properly.). We compute two kinds of geometry approximations: a hierarchical exterior approximation for the whole scene (later used for our occlusion queries) and an interior approximation for single objects (which can be used as an efficient occluder replacement during rendering).

For the construction of the outer hulls in the hull tree the scene is first subdivided by a regular object-based octree, in which the scene's objects are referenced redundantly by all leaf nodes that intersect with the object. By construction, an octree's node's bounding box completely encloses all children's bounding boxes and can therefore serve as an outer hull for the corresponding subtree. The main idea of the hull tree is not to directly use this AABB of a node as a hull, but to automatically select a simple volumetric hull shape out of a predefined set of shapes for all nodes in the tree. For a shape to be selected, it has to completely cover all shapes of the node's child nodes and associated scene objects. These shapes can have a significantly lower volume and projected size compared to the underlying box while offering a comparable complexity.

For interior approximation generation we use the edge-collapse algorithm introduced by Hoppe [HDD+93], with a special condition that guarantees the approximation to always be inside the original geometry (similar to progressive hulls [SGG+00]). Due to the additional condition, the degree in which an object can be simplified strongly depends on the topology of the mesh. Although one can easily construct meshes that cannot be simplified at all without violating the condition, many objects (for example originated from technical CAD-data) can be reduced to only a small fraction of their initial complexity.

Rendering cycle: Our approximative occlusion culling algorithm works in two main phases. First, all objects that were classified as visible in the previous frame are sorted by their properties and rendered while tested for occlusion using hardware accelerated occlusion queries (without waiting for the results). Second, the other scene elements' visibility classification is updated for the next frame's rendering.

This second phase exploits the prefilled depth buffer on the GPU of the first phase. The hull tree is traversed and occlusion queries based on the simple volumetric hull shapes are initiated to determining subtrees' visibility. To update the visibility of objects which have not been rendered in phase one, the objects' interior approximations are used and rendered to the depth buffer. These two phases provide a minimal number of state changes and virtually no pipeline stalls.

For this rendering method we developed the hull tree. This data structure is based on a regular octree and inherits its hierarchical properties. Instead of simple rectangular bounding boxes, this data structure consists of multiple shaped bounding volumes. Thus these hulls surfaces and volumes are significantly smaller which lead to faster visibility tests. We present an approximative render method that exploits the hull tree in combination with the used interior approximations.

6.2 Related Work

In this section we present work related to occlusion culling, mesh simplification and hull-approximations.

Occlusion Culling, in general, describes techniques which try to identify and leave out occluded parts of a scene's geometry to increase the rendering's performance. According to the classification of Cohen-Or et al. [COCSD03], the presented occlusion culling method is a point based (visibility is determined from the current observer position) technique with an image-precision definition of visibility, (i. e. an object is visible if it contributes at least one pixel to the final rendered image). Due to the special support for so-called hardware assisted occlusion queries in current graphics adapters, these kinds of techniques are used in a wide range of applications for real-time rendering (for example games). An occlusion query returns the number of pixels which pass the depth test during an object's rasterization. If this object is only a simple and cheap-to-render proxy of the original geometry, the visibility of the proxy can be used to estimate whether the original object is visible or not. One problem which arises is that the occlusion query itself can be performed quite efficiently, but it may take up to several milliseconds before a query result is available from the graphics card. This problem is addressed by several techniques. The CHC algorithm (Coherent Hierarchical Culling) by Bittner et al. [BWPP04] tries to reduce possible pipeline stalls by an asynchronous use of occlusion queries, in which the goal is to avoid idle waiting for a query's result. Several techniques try to further improve this idea. The *Near Optimal Hierarchical Culling* algorithm by Guthe et al. [GBK06] tries to reduce the number of issued queries to a minimum by estimating the outcome of the queries. Another approach is the CHC++ algorithm by Mattausch et al. [MBW08], which reduces the number of queries and the possible pipeline stalls, next to other improvements, by grouping multiple queries into batches (in which, as mentioned, objects can be sorted by their state). By allowing our algorithm to temporarily make errors in the visibility determination and therefore in the final image (approximative culling), we can benefit from a wide temporal separation from the start of an occlusion query to the reading of its result. This makes such queries very efficient to use compared to other, more conservative algorithms.

One important aspect for a culling algorithm's efficiency is the hierarchical data structure for the scene that determines the object's grouping and the bounding volumes used for the occlusion queries (for example Meißner et al. [MBM+01] examine this effect). For this purpose we use the presented hull tree, being optimized to provide tight bounding volumes for reliable occlusion queries.

Mesh simplification is another possibility to reduce the number of triangles sent into the rendering pipeline. Instead of rendering the original object, a simpler version is generated and rendered which should retain the original object's appearance as close as possible. For our rendering algorithm, we do not use simplified objects to reduce the visible object's complexity, instead simple objects are used as occluders to speed up the visibility tests. Therefore it is important for our method that an object's shape is preserved as well as possible and that the object is not enlarged, whereas the visual appearance is of no importance.

Hoppe et al. [HDD+93] present techniques to optimize and simplify meshes that can be used as LODs (levels of detail). For continuous LOD Hoppe presents *Progressive meshes* [Hop96]. These approximations cannot be used for occlusion culling directly because they provide no information on whether the surface of these approximations is outside or inside the original object's surface.

Cohen et al. [CVM+96] propose a method for mesh approximations called *simplified envelopes* where they also generate an inner and outer hull. Those hulls are only used as intermediates to confine the actual approximation (which is generated in between) to restrict the variance to the original geometry. The hulls are generated by moving the original geometry's points along their normals, thus the hulls have the same complexity as the original model. As for the approximation, there is no prediction whether it is inside or outside the original geometry. Therefore it cannot be used for our algorithm.

Sander et al. [SGG+00] present *progressive hulls*. Progressive hulls are a special case of progressive meshes. This technique allows the computation of continuous approximations, which are always inside or always outside of the original mesh. Due to the progressive property, the amount of memory is even higher than the memory needed to store the original mesh. For our purposes, we need a mesh simplification with reduced memory requirements. Our approximation is not continuous, it is one fixed state of this sequence.

Hull-approximations can be used to approximate objects. Alt et al. present a method to approximate polygons with less complex polygons or circles [ABW90]. To apply their method, the polygon must have a convex shape in order to be simplified. Held and Eibl present an approach to compute a simplified representation that is generated within a defined area, by tangent-continuous approximation [HE05]. In contrast to the presented techniques, our approach works for arbitrarily shaped polygons and meshes.

6.3 Building the Hull Tree

The hull tree's aim is to automatically create a non-overlapping, hierarchical spatial data structure in which a simple convex hull is attached to every node which covers the whole subtree tightly. In order to generate these hulls efficiently, we restrict the possible geometrical hull shapes to a predefined set of shapes which meet the following requirements: each shape must consist of only a few triangles, be completely coverable by other shapes on the next lower level, and allow an efficient decision on which shape is covering all corresponding children's shapes tightest. Our manually chosen set of possible shapes consists of three-sided prisms, cuboids, and different five- and six-sided prisms. These shapes are the distinction of our exterior hulls. Figure 6.2 shows some of these available shapes; the complete set also includes the corresponding rotated and mirrored shapes. Figure 6.3 gives an impression of two different levels of hull tree shapes compared to the corresponding octree boxes, both covering the same scene.

The construction of the hull tree begins with the construction of an object based octree covering the whole scene. The maximal depth is given as a parameter. The

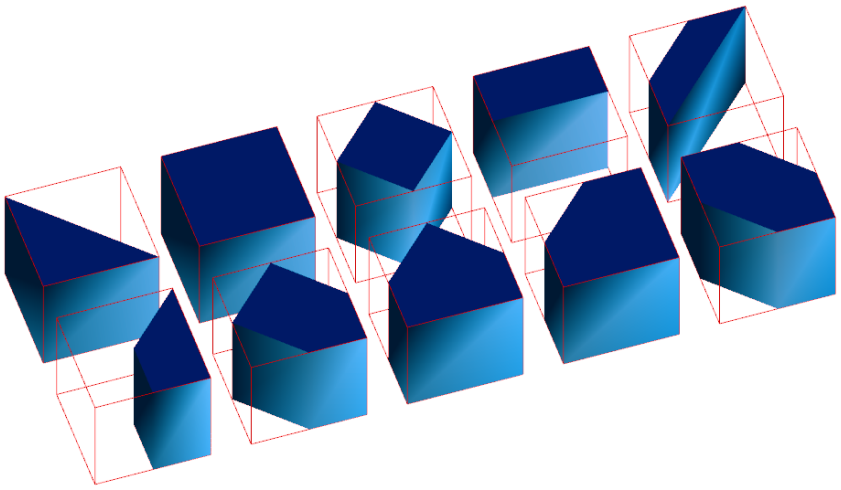
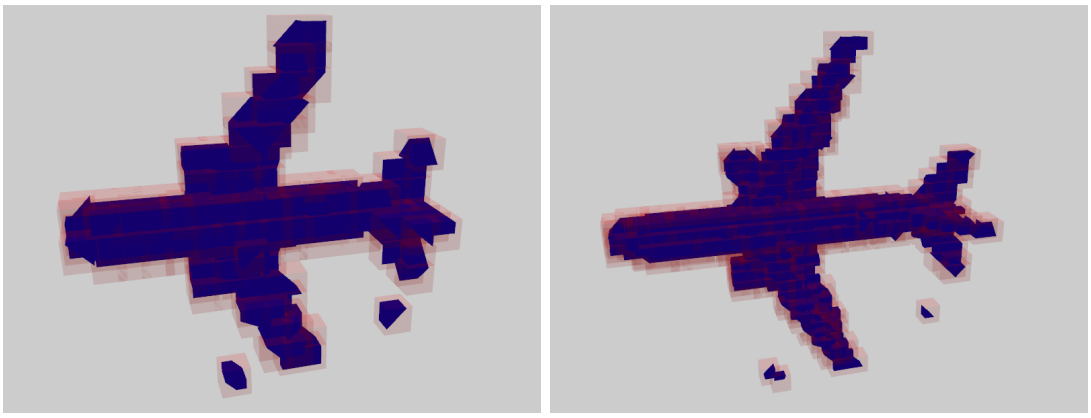


Figure 6.2: Set of possible hull shapes. The complete set also contains all reasonable rotated and mirrored versions.

objects are referenced redundantly in all leaf nodes they intersect with. Beginning with the lowest level containing the objects, the hull shapes are created for each node from the leaves up to the root of the tree. A simple bounding box is chosen as a shape for all leaf nodes. Experiments implied that the additional computational effort for choosing tighter hull shapes for leaf nodes marginally influences the later rendering process if only the chosen depth of the octree is large enough. An inner node's shape is then chosen based on its children's shapes. Because of the shapes' simple structure, it is sufficient to base the decision on the coverage of several fixed points inside of the node's bounding volume (for example if all eight corners of the node's bounding box are covered by shapes of the children, the only possible choice for the node is the complete box). In this manner, a decision tree is used to efficiently identify the smallest shape (by volume) which completely covers all child nodes' shapes (see Figure 6.4). If several shapes have the same volume, a shape with the lowest polygon count is chosen. When the root node is reached, the construction of the hull tree is complete.

6.4 Determining the Interior Approximations

The second part of the preprocessing stage is the calculation of the scene objects' interior approximations. The approach used does not require convex or closed objects. To be able to use these approximations as occluders (without creating errors by occluding more than the original object), the volume of these approximations must be smaller than the original object's volume and the approximation's surface may not intersect with the original object's geometry. These interior approximations are generated with Hoppe's edge-collapse algorithm, with a special cost function. This works in a similar fashion to the calculation of the interior volume of progressive hulls [SGG+00], but we compute only a single fixed state of the progressive version.



(a) (b)

Figure 6.3: Levels 4 and 5 of the Boeing 777 scene's hull tree. The accuracy increases with each level. The transparent red cubes visualize the cells of an octree at one level. The included blue polyhedrons show our much tighter shapes from the hull tree at the same level.

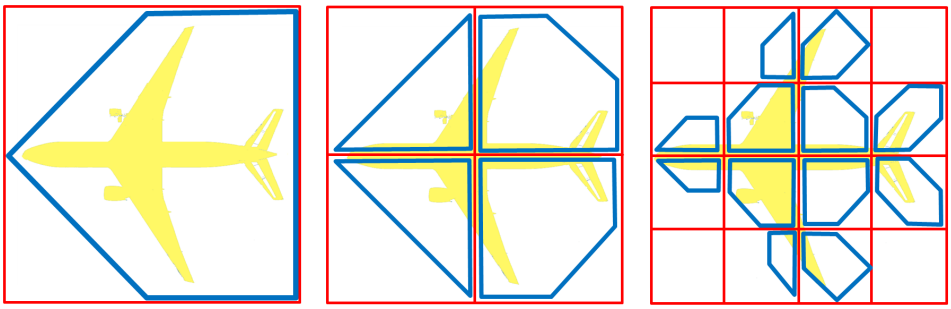


Figure 6.4: Schematic 2D illustration of a hull tree: the left square shows the root node (Level 0), and the middle and right square show Level 1 and Level 2. The hull shapes (see Fig. 6.2) are drawn with blue lines.

We use a special cost function to preserve holes and to ensure that the interior approximation is smaller than the original model. The desired goal is to collapse c edges (c is given as a parameter) while maximizing the total profit as close as possible to $c \cdot d$. For the simplification, we need the diameter d of an object's bounding box. If the vertices on an edge are not both vertices of exactly two triangles, the profit for collapsing this edge is $-\infty$. In other word, an edge that should be collapsed must connect the complete sides of two triangles. When an edge-collapse operation increases the object's volume or leads to an intersection with other triangles, the profit is also $-\infty$. Otherwise the profit is $d - v$, where v is the height of the cut off volume. Following this, the algorithm tries to keep the lost volume small (see Figure 6.5). The algorithm terminates, when c edges are joined or the only possible collapses brings a profit of $-\infty$ (see Figure 6.5). We discard normals and colors of our approximations because we do not need them for our culling algorithm.

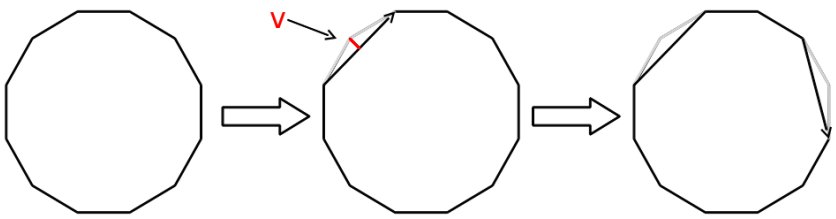
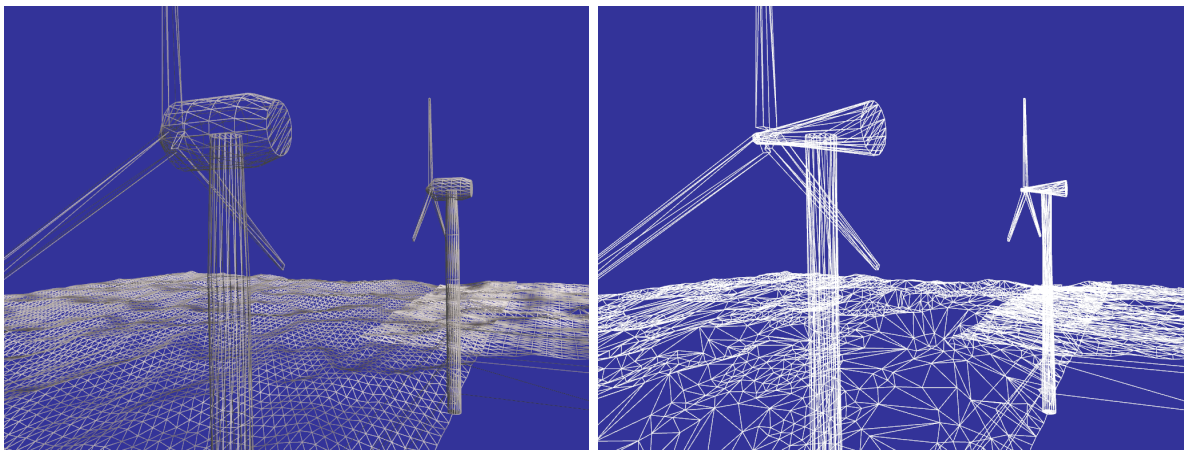


Figure 6.5: By collapsing edges using the proposed cost function, the approximation will always be located inside the original geometry.



(a) (b)

Figure 6.6: The left image shows the original geometry and the right image shows our interior approximation - both rendered as wire-frame.

These interior approximations then can be used as occluders for the visibility tests. Their projected size may be smaller than the related original objects' sizes, but if the interior approximation occludes another object, the original object would do this as well, except for rare cases (see Figure 6.6). It is easy to handle the case of a camera placed in-between the original and approximations surface: If the camera's position is inside a hull tree cell, all of its included original geometry is rendered.

6.5 Rendering Algorithm

The rendering algorithm can be subdivided into five steps (see Algorithm 6.1).

i. At the beginning of a new frame, the list of objects that were identified as visible in the previous frame are grouped by their state (shader, texture, etc.). The state-grouping can reduce the number of necessary state changes of the rendering pipeline to the overall number of different states in the scene. This can be essential for the rendering performance, especially if the rendered scene contains a huge number of objects

having many different states. Then, each group's objects are sorted by distance to the observer in front-to-back order so that a possible occlusion between objects inside a group can be detected during rendering. A sorting of the groups themselves may not reveal the occlusion between objects from different groups because of the possibly large intersections. Therefore, we now randomly permute the order of the groups themselves so that an occlusion is not guaranteed to be identified immediately but is likely to be detected after a few frames. At the end of the first step all objects are rendered into the frame buffer in the specified order while, for each object, an occlusion query is initiated. These tests return the number of object's pixels which actually passed the depth tests during rendering. These tests introduce only a minor runtime overhead since the results are read back only at a later step of the algorithm where no active waiting for results is expected. Due to the usage of possibly outdated visibility information from the last frame, visible objects can be missing in the rendered image for a few frames. This effect is mostly noticeable in the first frames (where no visibility information is yet available) and when the observer performs quick rotations or walks through a wall for example. These image errors are visible for at most d frames, where d is the depth of the hull tree.

ii. In the second step, all hull tree nodes and objects for which the visibility information should be updated are collected into lists by traversing the tree. Nodes which are collected must be located inside the viewing frustum and either be marked as visible with at least one invisible child or be marked as invisible, but having a visible parent. All objects that are referenced by the latter kind of nodes (marked invisible, but whose parents are marked visible) are added to another list. These are objects that are likely to appear when the observer moves (or may even actually be visible, but not yet identified), making them good candidates for further examination.

iii. When the observer moves through the scene, it is likely that many new objects enter the viewing frustum at once. Therefore, the interior approximations of the objects collected in the second step are rendered to the depth buffer (in front-to-back order from the observer) while an occlusion query is initialized for each rendered approximation. This improves the information in the depth buffer for further tests, gives fine granularity visibility results for the corresponding objects and is quite efficient since most nodes' inner approximation complexity is only a fraction from the original mesh's complexity.

iv. Then, exploiting the completed depth buffer, occlusion tests are initialized for all the nodes' hull shapes collected in the second step (without altering the depth buffer).

v. In the final step, the results of all occlusion queries initialized in this frame are read back from the GPU. Those objects whose corresponding test (or the test of the associated interior approximation) returns that at least one pixel has passed the depth test, are added to a list of visible objects. This list is used in the first algorithm step of the next frame. According to the results for the tested hull shapes, the corresponding nodes' visibility markings are updated.

Algorithm 6.1 RENDERING-LOOP

Variables: HullTreeRoot, visibleObjectList, possiblyVisibleObjectsList, testedElementsQueue, nodeTestList

```
    // i. Sort and render visible objects
 1: enable writing to the frame- and depth buffer
 2: group objects in visibleObjectList by their states
 3: for all groups in random order do
 4:    change state according to group
 5:    sort objects in group by increasing distance to observer
 6:    for all objects in the group do
 7:       render original object with occlusion query
 8:       add object to testedElementsQueue
 9:       mark object as rendered
10:    end for
11: end for
    // ii. Collect internal nodes and possibly visible objects
12: TraverseHullTree(HullTreeRoot)
    // iii. Handle possibly visible objects
13: disable writing to the frame-buffer
14: sort objects in possiblyVisibleObjectsList by increasing distance to observer
15: for all objects in possiblyVisibleObjectsList do
16:    if object not marked as rendered then
17:       render object's interior approximation with occlusion query
18:       add object to testedElementsQueue
19:       mark object as rendered
20:    end if
21: end for
    // iv. Init tests for inner nodes
22: disable writing to the frame- and depth buffer
23: for all objects in nodeTestList do
24:    render nodes's hull shape with occlusion query
25:    add node to testedElementsQueue
26: end for
    // v. Update visibility
27: clear visibleObjectList
28: for all elements in testedElementsQueue do
29:    if occlusion query result is visible then
30:       mark element as visible
31:       if element is object then
32:          add object to visibleObjectList
33:       end if
34:    else
35:       mark element as invisible
36:    end if
37: end for
```

Algorithm 6.2 TraverseHullTree(node)

```
Variables: testedElementsQueue, cameraPosition
 1: if node outside Viewing-Frustum then
 2:   return
 3: else if cameraPosition outside node's bounding box then
 4:   if node marked as visible then
 5:     if at least one child is marked invisible then
 6:       add node to nodeTestList
 7:     end if
 8:     for all node's child nodes do
 9:       TraverseHullTree(child node)
10:     end for
11:   else if node is leaf then
12:     add node to nodeTestList
13:     for all objects stored in the node do
14:       add object to possiblyVisibleObjectsList
15:     end for
16:   end if
17: else
18:   // Special case if the camera is located inside the node
19:   mark node as visible
20:   if node is leaf then
21:     mark all node's objects as visible
22:   else
23:     for all node's child nodes do
24:       TraverseHullTree(child node)
25:     end for
26:   end if
27: end if
```

On the one hand, our algorithm changes the detected visibility usually only for one level of the hull tree per frame, which can lead to temporary image errors. On the other hand, even complex scenes can be rendered with only few state changes of the graphics pipeline and the occlusion tests are performed very efficiently because, typically, no active waiting for the results is required.

6.6 Evaluation

We performed different tests to evaluate our approximations and our rendering algorithm. We first started with tests related to the properties of our approximation compared with standard bounding boxes. Tests were made with different scenes: the model of the UNC Power Plant, a model of a Boeing 777, and two versions of a container port. After that we performed runtime tests where we compared our rendering method with

the CHC and the CHC++ algorithm. We measured the differences in the number of executed occlusion tests, the differences in the count of the rendered objects, and pixel error that occurs if our approximate rendering algorithm is used. To expose the advantage of our algorithm with non-axis-aligned scenes, we used the container port scenes *Container Port* and *Container Port 45*, each consisting of 5,322 objects and having approximately 6.5 M triangles. Each object has its own shader. There are five different shader programs, randomly attached to the different objects. The objects in the *Container Port* scene are mostly axis-aligned, so most of the objects can be inserted in axis-aligned bounding boxes without wasting too much space. The *Container Port 45* is identical to the *Container Port* scene, except that the scene is rotated by 45 degrees around the y-axis. In that configuration, occlusion tests with AABBs lead to more false-positives.

For the tests, we implemented our rendering system in C++. The tests are performed on a standard PC, equipped with a NVIDIA GeForce 260 GTX, a Quad-Core Intel CPU, 8 GiB RAM and Ubuntu 9.04 GNU/Linux as operating system.

Surface area, volume, and memory consumption

In our first tests we evaluated how much of the surface and volume is saved if we use our exterior approximation instead of bounding boxes (see Figure 6.7). For both port scenes we build a hull tree with a depth of seven, which proved to be a good trade-off between accuracy and necessary traversal time. The hull trees for the UNC Power Plant and the Boeing 777 scene have a depth of ten.

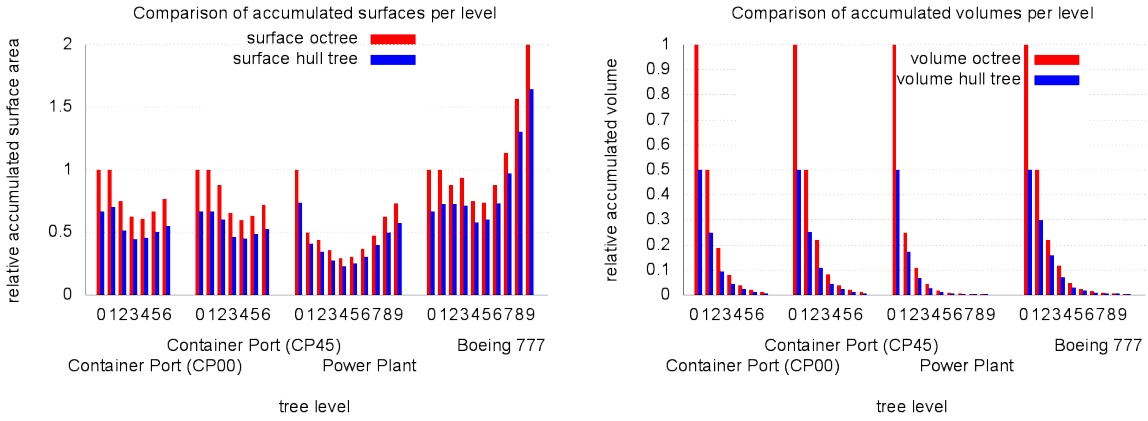


Figure 6.7: On every level of the hull tree, the surface area and the volume of the approximations is smaller than the surface area and the volume of the octree cells.

The values inside the diagrams are normalized, so that the root box of the octree has surface area 1 and volume 1. As expected, the surface and the volume of the exterior approximation was smaller than the surface and the volume of the octree cells' bounding boxes. Furthermore, the savings increased on lower levels. Through this

Table 6.1: Comparison of the triangle count of the AABBs and our approximations.

	Container Port			Container Port 45		
Lev.	Δ AABB	Δ Approx.	fac.	Δ AABB	Δ Approx.	fac.
0	12	12	1.0	12	12	1.0
1	48	40	0.83	48	48	1.0
2	144	128	0.89	168	152	0.9
3	480	460	0.96	504	500	0.99
4	1,872	1,824	0.97	1,836	1,768	0.96
5	8,208	8,232	1.0	7,740	7,848	1.01
6	37,632	37,484	1.0	35,352	35,220	1.0

	UNC Power Plant			Boeing 777		
Lev.	Δ AABB	Δ Approx.	fac.	Δ AABB	Δ Approx.	fac.
0	12	8	0.67	12	12	1.0
1	24	24	1.0	48	52	1.08
2	84	72	0.86	168	184	1.1
3	276	236	0.86	720	632	0.88
4	900	860	0.96	2,292	2,184	0.95
5	3,732	3,600	0.96	9,060	8,968	0.99
6	17,988	17,344	0.96	43,212	43,800	1.01
7	92,904	94,860	1.02	222,528	228,332	1.03
8	492,096	503,608	1.02	1,234,884	1,274,460	1.03
9	2,303,244	2,302,204	1.0	6,305,292	6,417,264	1.02

increased precision, the number of false-positive occlusion queries decreases, thus the number of initiated tests on lower levels is also reduced.

However, saving surface and volume for the visibility tests is not helpful when the proxy geometry is too complex. Hence, we evaluated how many triangles are needed for our approximations compared to the standard bounding boxes, for every octree level (see Table 6.1). The measurements show that the number of triangles needed for our exterior approximation is close to the number of triangles needed for standard bounding boxes. In some situations the approximation needs less triangles, in some it needs only a few more. These first tests show that using our exterior approximation saves a significant amount of surface area and volume, and that the approximation's complexity does not differ too much from the complexity of the standard AABBs. The construction of the hull tree took only a few seconds longer than the octree's construction. The overhead results from the additional operations required to determine the cell's tighter shape by examine its children nodes.

In the container port scenes, the interior approximation's vertex data requires only 86 MiB, while the original scenes need 276 MiB of memory. Our simplification algorithm was configured to create approximations with about 60 vertices per object, if possible.

Settings of the runtime tests

For the runtime tests of our rendering algorithm and the approximations, we used the different container port scenes (see Figure 6.9). For comparison, we chose the CHC algorithm as presented in [PF05], using a loose octree for storing the scene. This established algorithm allows for quick occlusion culling. Although this algorithm is conservative, it

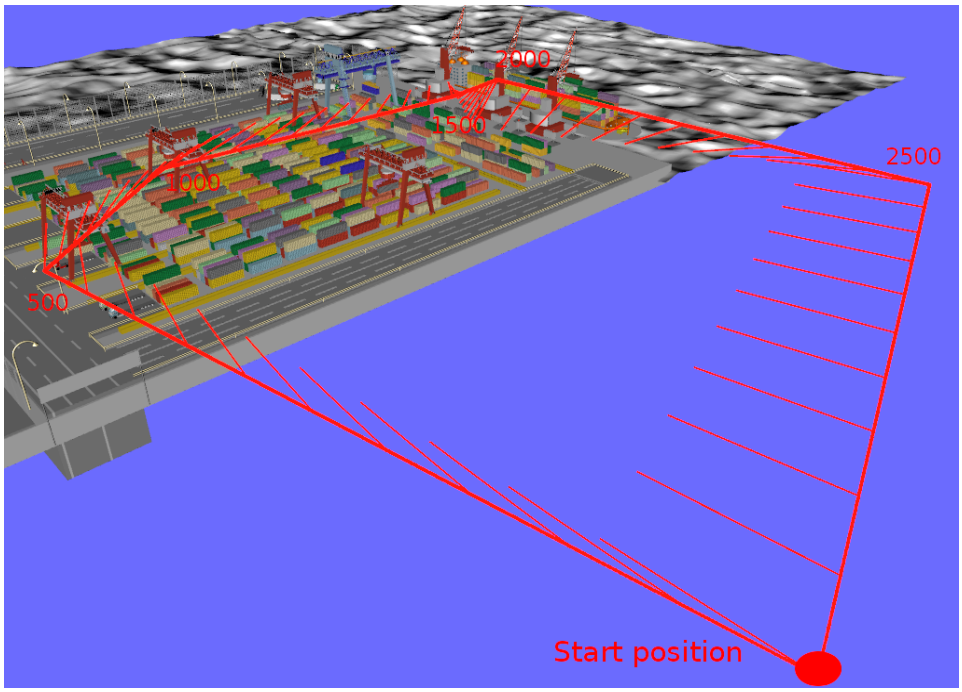


Figure 6.8: Visualization of the camera path with the viewing directions used for the tests.

basically uses the same building blocks as our algorithm and works without any additional parameters. In our opinion, this makes it a good candidate for the comparison. For the rendering time, we additionally compared the results to the CHC++ algorithm, using the parametrization as given in [MBW08].
We perform a walk-through as shown in Figure 6.8 for all of our runtime tests. We start outside the port where we can see most of the environment. We quickly move the camera to the outer left side of the container area. Then we turn right and move between the containers, until we reach the cranes. We move left beyond the cranes, outside the port, and direct our view to the center of the scene. After that we turn right again and move back to the start position of the walk-through, while we still focus on the center of the scene. The camera never holds its position, which results in a permanently changing set of visible objects. The whole walk-through consists of 3,000 frames. The walk through the Container Port 45 scene is similar, but not equal to the described walk-through. In the middle section (frame 1,000-1,500) it differs. Here the camera is moved through a channel built by many containers.

Image quality

In our first runtime test, we evaluate the rendering errors that occur if we use our rendering method. For these tests we render each frame on the camera path twice with a resolution of 1,024 × 768 pixels. Here a direct comparison is possible because the additional needed computation time does not influence the results of this test (unlike

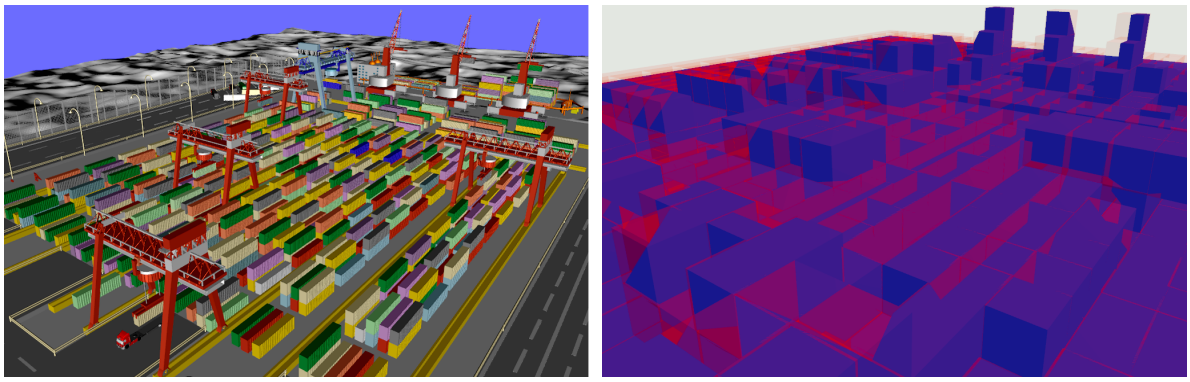


Figure 6.9: The left image shows a segment of the *Container Port* scene. In the right image the blue hulls are our approximations, and the transparent red boxes are the corresponding AABBs.

the parallel rendering system that communicate asynchronously).

First we render an image by using our rendering method and copy the resulting frame into a buffer. In a second step, we do not change the camera position or orientation and render an image with all objects, and copy the resulting frame into a second buffer. Afterwards, we compare the buffers pixel by pixel and count the differences (see Figure 6.10).

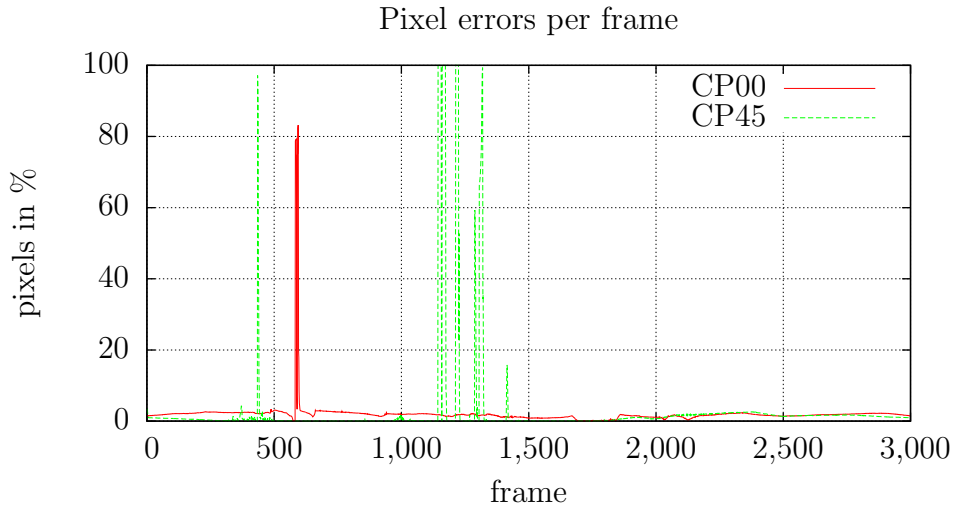


Figure 6.10: Our rendering method produces images with a typically low pixel error, with only a few extreme exceptions (thin peaks in the diagram above).

For most of the camera positions, the pixel error is close to 1%. There are a few camera positions in which the pixel error is close to 100%. These peeks occur when the camera is moved through a wall. In this situation it needs at most d frames (d being

the depth of the hull tree) to determine the set of visible objects. After those frames the image is mostly correct again.

Usually, when the visibility changes only gradually, errors occur only at small areas near the border of the image and disappear almost immediately after only a few frames. Due to this short time period, this is only perceived as a short flickering.

Tested objects in comparison with the CHC

In the comparison of our rendering algorithm with the CHC algorithm we also evaluated the number of occlusion queries per frame. Figure 6.11 shows the results of these tests. In most cases the CHC algorithm needs less tests than our rendering algorithm. We have implemented the CHC algorithm as it is given in [PF05]. The algorithm does not perform occlusion queries for each object. It stops testing when it reaches one of the utilized data structure's leaves and renders all included objects. Our algorithm also tests the different objects in the leaves, which is why it initiates more tests.

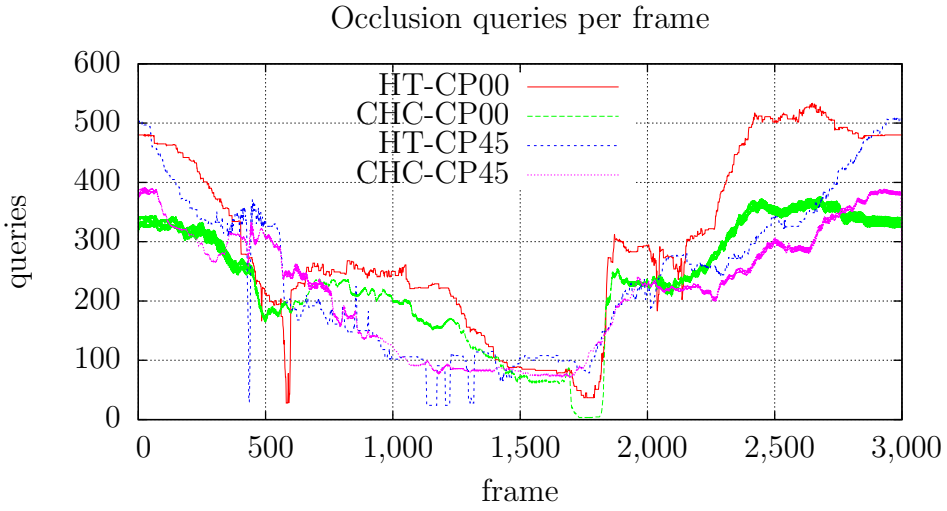


Figure 6.11: Comparison of the initiated occlusion queries using our rendering algorithm and the CHC algorithm. (Values for the CHC are floating averages over a range of eight frames to mask out the high frame to frame fluctuations.)

However, by performing more occlusion queries, we can reduce the number of objects sent into the rendering pipeline. The additional tests do not influence the frame time negatively. Our testing routine prevents the rendering pipeline from stalling, while CHC's occlusion queries can stall it. For nearly every camera position on our walk-through our rendering algorithm renders less objects, as shown in Figure 6.12.

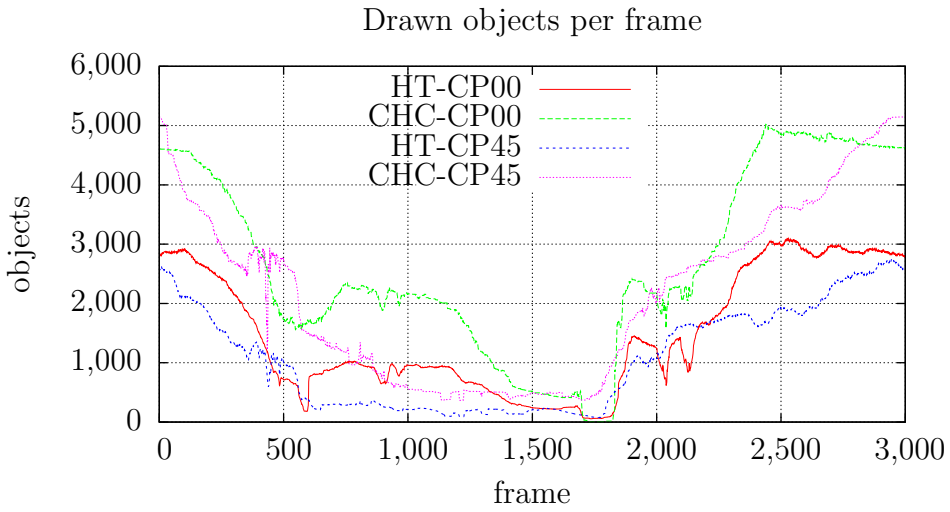


Figure 6.12: Comparison of the drawn objects' count using our rendering algorithm compared with the CHC algorithm.

Rendering performance

Next, we evaluate the performance of our rendering technique. For this, we compare the time needed to render a frame using our algorithm with the rendering time using the CHC and the CHC++ algorithms. For this test we measure the rendering time for each camera position with each algorithm. The results are plotted in Figure 6.13.

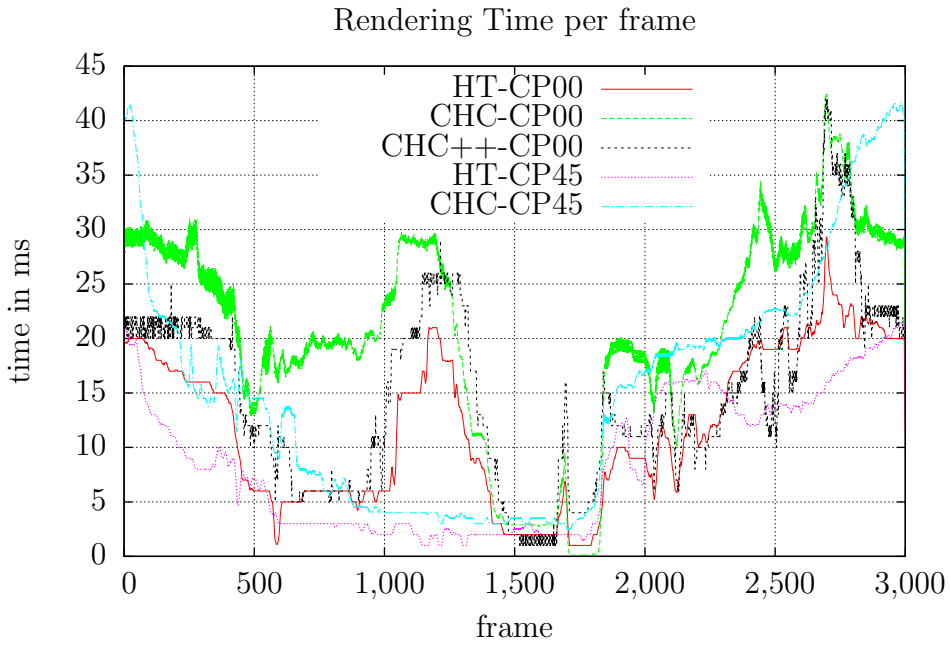


Figure 6.13: Comparison of the rendering times using our rendering algorithm and the CHC algorithm. Our algorithm is faster at nearly every camera position.

The result shows that our algorithm provides a higher frame rate than the CHC because our algorithm renders less objects and initiates fewer state changes. As expected, the CHC++ outperforms the CHC algorithm at almost all positions. Compared with our algorithm, the CHC++ produces quite similar results. However, our approximative culling is faster than the CHC++ most of the time. In the Container Port scene the sum of the rendering times is ≈ 61.752 seconds when the CHC is used, ≈ 46.839 seconds when the CHC++ is used, and ≈ 35.750 seconds when we use our rendering algorithm with the hull tree.

Number of state-changes

This test evaluates how often the shader is exchanged using our rendering algorithm and the CHC algorithm. Figure 6.14 shows that the number of state-changes is limited by the number of different shader when our rendering algorithm is used. When the CHC algorithm is used, the state of the rendering pipeline must be changed for nearly every object that is rendered.

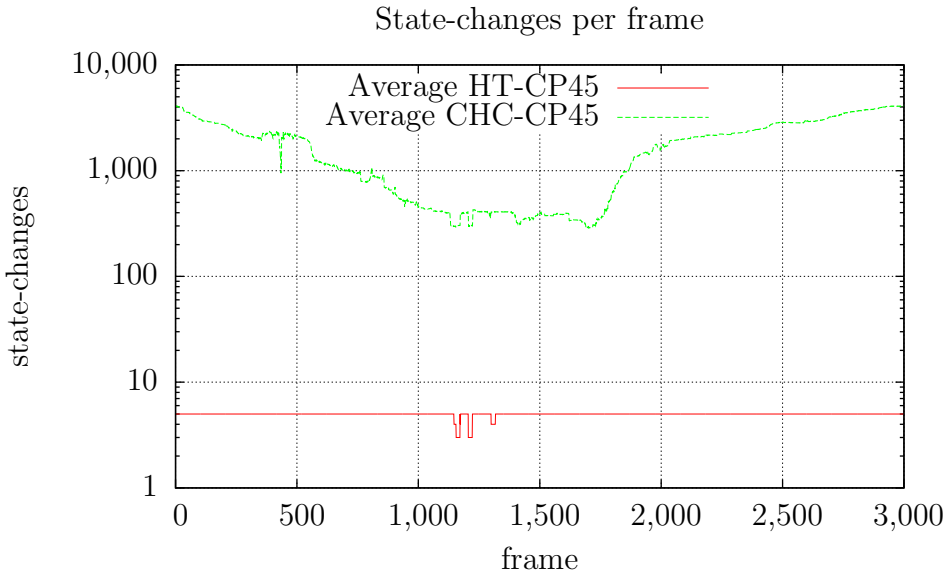


Figure 6.14: Comparison of the shader changes using our rendering algorithm compared to the CHC algorithm. The sorting of the objects by their shaders limits the number of changes to the number of different shaders.

The CHC++ algorithm can reduce the number of state-changes. To reduce waiting times, this algorithm sends a batch of objects into the rendering pipeline, and receives the occlusion query results afterwards. The objects of each batch could be sorted by their state, which influences occlusion tests' quality negatively. To achieve best results from the visibility tests, the objects should be sent in front-to-back order into the rendering pipeline, just like it is done by the CHC algorithm.

Improvements by the interior approximations and exterior hulls

Our rendering system uses the interior approximations to test the visibility of scene objects. By using these approximations instead of the original meshes we sent on average only 68% of the triangles into the rendering pipeline to perform the visibility tests (see Section 6.5, Step *iii*). When a user moves through the scene, looking around freely, the number of initiated occlusion queries is lower when the hull tree is used in comparison to a regular octree. In our tests the overall number of performed occlusion tests is reduced by up to 5% compared to the usage of AABBs, especially while the camera moves quickly and is distanced from the scene. Here the algorithm benefits from the tighter cells. The culling algorithm must test additional octree levels while it can skip tests when the hull tree is used.

In our last test we evaluated how many pixels were written into the depth buffer during the visibility tests. Here we compare the number of drawn pixels while using the hull tree with the number of drawn pixels while using an octree. If the hull tree is used for the occlusion tests only 74% of the pixels are drawn into the depth buffer on average (see Figure 6.15).

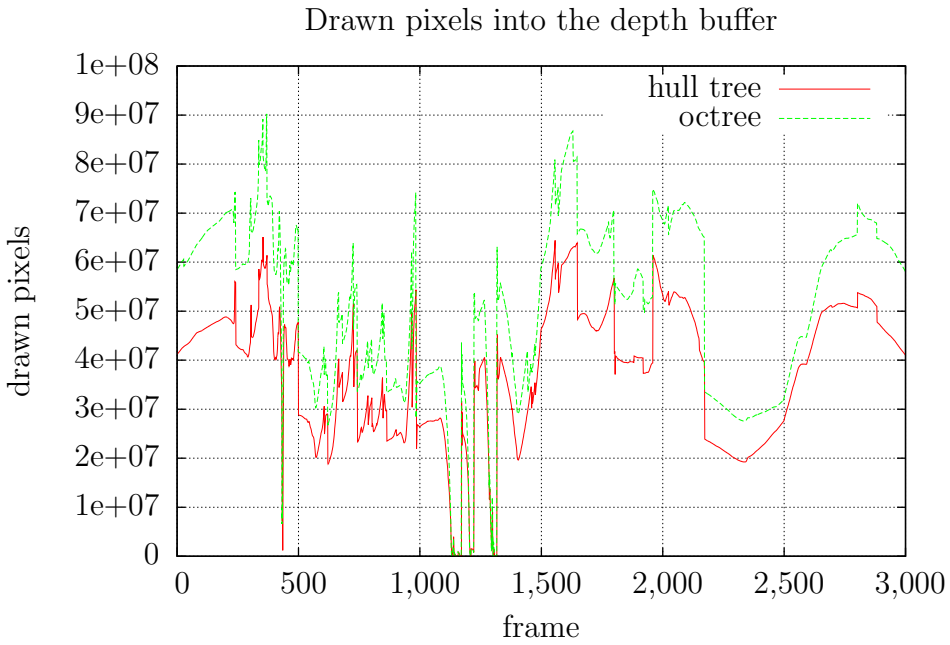


Figure 6.15: Comparison of the number of pixels drawn to the depth buffer using the hull tree and an octree.

This reduces the load on the graphics card's raster operation pipeline. This is an additional positive effect using the hull tree instead of an octree.

6.7 Contribution

In this chapter we introduced a combination of approximations, organized in a hierarchical data structure, which allows a reduction in the number of false-positives, thus reducing the overall number of performed occlusion tests. The bounding volumes of the hull tree covers scenes' objects tighter than other spatial hierarchical data structures. Additionally, the algorithm incorporates knowledge about the different states in the draw order of the objects. Therefore it can limit the maximum number of state changes to the number of different states, whereas other rendering algorithms do not utilize this information which can lead to one state change per drawn object. Due to these properties of our rendering algorithm and our approximations we can improve the rendering performance of complex scenes without any scene specific configurations.

7 Parallel Out-of-Core Occlusion Culling using the Hull Tree

In the previous chapter we introduced the hull tree data structure and a related rendering algorithm. In this chapter we modify this data structure to make it suitable for a parallel out-of-core rendering algorithm we developed [SKJ+11]. Here, we again look at a single visualization node equipped with a high-end graphics adapter, that is supported by a group of back-end nodes with weak graphics performance. The scenes to be visualized are so large that they neither can be stored in the visualization node's memory nor be displayed in real-time with most standard techniques. We must therefore consider how the slower back-end nodes can be used to accelerate the faster visualization node and simultaneously how the objects are stored in the cluster. In comparison to the back-end nodes in the approach presented in Chapter 5, these nodes receive no information besides the current camera settings.

Our approach is as follows: the back-end nodes serve two purposes. First, their main memory provides an out-of-core memory system for the visualization node. Second, they assist the visualization node's rendering by performing visibility calculations and sending only visible objects to the visualization node. In order to obtain fast rendering with our system, we have to distribute the objects among the back-end nodes in a way that not only guarantees an even distribution of the objects, but also evenly distributed visibility calculations. To achieve scalability and efficiency, our system has to meet the following goals:

- The scene must be distributed evenly (without redundancy) among the combined main memory of the back-end nodes.
- The computational load of visibility calculations should be balanced for most camera positions.
- The back-end nodes' response times (visibility calculations) must be kept as short as possible.
- The amount of data to be sent to the visualization node must be kept small and evenly distributed among the back-end nodes.

To fulfill these requirements, we use a pseudo-random distribution of the scene objects across the back-end nodes' main memory. To reduce the number of objects sent to the visualization node at once, we employ an approximate hierarchical occlusion culling on each back-end node. For this, they are equipped, in addition to the objects assigned to

them, with simplified versions of the remaining objects of the 3D scene. In the previous chapters, we have used only PC clusters whose nodes are connected via a high speed network. For the rendering system we additionally analyze the effect of only using a standard network but with increased rendering performance of the back-end nodes.

7.1 Overview and Summary of Results

The single, powerful visualization node, connected to a number of back-end nodes via network, renders the scene and allows the user to move interactively throughout the 3D scene. Scene's objects are distributed among the back-end nodes.

Object's random distribution and hull tree modifications: Usually scalability in scene size is achieved by external storage on disk, but with the drawback of slow memory accesses. Fast visibility tests are easily achieved by the redundant storage of the whole scene on each node, but with the drawback of low scalability. We solve these conflicting requirements for our system as follows: We distribute all objects randomly and without any redundancy among the back-end nodes and store them in processors' main memory. We employ an approximate hierarchical occlusion culling algorithm on each back-end node. For this, they are equipped with simplified versions of the scene objects in addition to the objects assigned to them. In other words, each back-end node contains a randomly chosen part of the scene's objects. For each object that it does not store, it maintains an interior approximation as described in the previous Section 6.4. The random distribution of the objects results in the even distribution of final image artifacts, while also ensuring that the amount of data to be sent is uniformly distributed among the back-end nodes.

General rendering loop: The main rendering loop works as follows: The visualization node maintains a set of visible objects in its GPU memory and renders them in successive images (visibility computations are not performed). The back-end nodes compute the visibility of the objects and send a list of objects to be added to or removed from rendering to the visualization node (see Figure 7.1). In case of objects to be deleted, only unique object identifiers are sent. Due to slow response times of the back-end nodes, the visualization node receives at most one update per rendered frame. After receiving updates from a given back-end node, the visualization node sends the current camera position back to that node. Now, the given back-end nodes perform occlusion tests for different camera positions. Parallelization is realized via parallel computation of visibility tests for different camera positions, unlike other methods which perform parallel computation for a single camera position.

Visibility computations: For our parallel rendering algorithm, we store either the original objects or their interior approximation in the hull tree at each back-end node. This differs from the sequentially used hull tree where we store both in tree's leaves. This tree is used to compute a hierarchical occlusion culling, whereby the objects, as well as the simplified objects, are used as occluders. The result of a hull tree traversal is a list of potentially visible objects valid for the camera position received from the visualization node (see Section 7.3). For those original objects whose visibility has changed, the back-

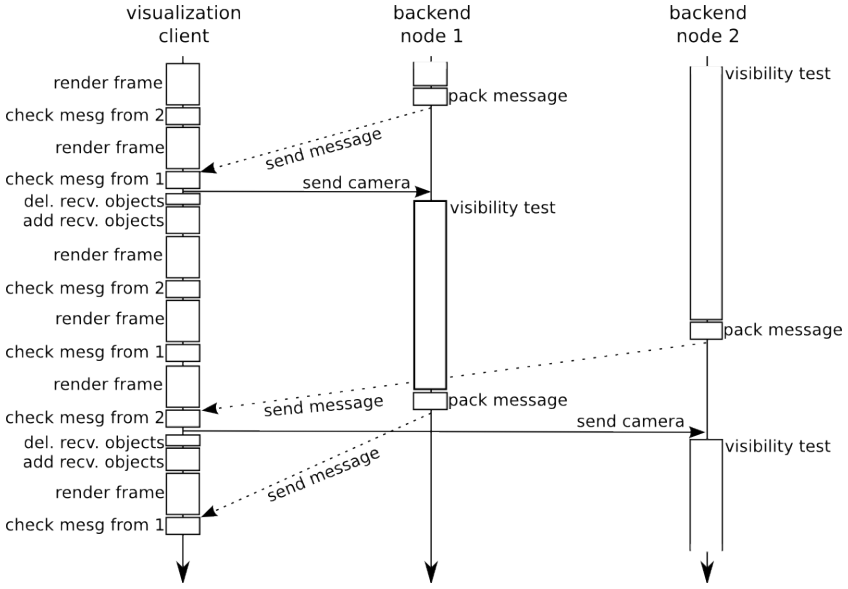


Figure 7.1: Communication scheme between visualization node and back-end nodes.

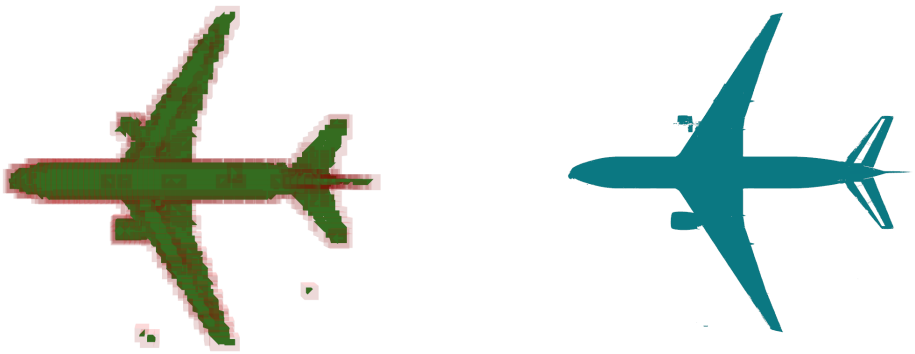
end node sends an update to the visualization node. If an object becomes visible then its geometry is sent, if it becomes invisible then its unique identifier is transferred.

In this system each of scene's original objects is stored on exactly one PC cluster node. Additionally, every node has approximations for the other objects. Due to the randomized distribution of the objects, the load through the visibility tests and the data sent across the network is evenly distributed among the back-end nodes. The utilization of the hull tree accelerates the back-end nodes' visibility tests, which results in a reduction in time between initiating a request and its receiving answer. Through the combination of a hull tree with a randomized sample tree, we keep the network load low. Tests in different hardware environments have shown that the graphics adapters' performance influences images' quality more than the network speed. This is reasoned by obtaining faster visibility updates for more reasoned camera positions.

For this system, we did not face the problem where the set of visible objects do not fit into the visualization node's primary memory. In this case, one could exploit the hull tree's combination with a randomized sample tree to keep the memory requirements low. It should be simple to combine the rendering algorithm for the randomized sample tree with our algorithm. Instead of loading all visible objects, we would discard objects on lower levels of the hierarchical data structure that are far away from camera position.

7.2 Related Work

One Parallel occlusion culling technique was suggested by Naga K. Govindaraju et al. [GSYM03]. Their system consists of three nodes: two occlusion-test nodes to perform the occlusion tests, and one visualization node to render the visible objects of the scene.



(a) (b)

Figure 7.2: Both images show approximations for the model of a Boeing 777 used by our rendering algorithm. The left image shows the exterior approximations in the sixth level of a hull tree (red shaded areas are related octree cells) and the right picture shows the composition of the interior approximations.

The objects are organized in a scene-graph, the nodes of which are addressable by a unique ID. In every frame one of the occlusion-test nodes performs an occlusion culling and sends the ids of the visible scene-graph nodes to the other computers. At the same time, the visualization node uses the previously received IDs to render the image visible for the user. The second occlusion-test node renders these objects as well. This image is the base for the node's next occlusion test. When all nodes have finalized their procedure, the two occlusion-test nodes switch their behavior. By sending only the ids of objects that must be rendered the network load is reduced. However, every node must also store the complete scene. In contrast, our system sends more data across the network but scene-data is distributed over the different back-end nodes. In this way scenes can be rendered that do not fit into node's memory.

Hua Xiong et al. presented different parallel occlusion culling methods [XPQS06]. These techniques are designed for a CAVE (partially or completely surrounding display walls) equipped with 15 projectors, each associated to a single computer. For their proposed methods, the scene-data is spatially partitioned in voxels. These voxels are distributed over the different nodes. In the *Sort-First Occlusion Culling* approach, every computer computes the occlusion culling on its own (the culling process is not distributed). In the *Sort-Last Occlusion Culling* approach, each node computes the visibility for all voxels in its local memory. Afterwards, the geometry of the visible voxels is sent to the nodes assigned to the tile where it has to be displayed.

7.3 Scene Preparation

To prepare a virtual scene for the presented technique, two different preprocessing steps have to be performed. In order to exploit spatial coherence during the occlusion tests, the scene objects are stored in a hull tree. The hull tree used in the sequential rendering

system does not meet all our needs. To achieve better performance, we combine the hull tree with the ideas of the randomized sample tree [KKF+02] . Next we describe the necessary preprocessing steps.

Bounding volume hierarchy

A loose octree serves as the basis for the construction of the bounding volume hierarchy of the scene's objects. In contrast to an ordinary octree, the boxes of the child nodes do not split up the parent's box disjointly, but they overlap. This allows objects to be stored uniquely in a node at a tree level corresponding to its size (see [Del00]). To combine our hull tree with the randomized sample tree we need to store each object only once in the data structure. The hull tree for sequential rendering does not provide this.

One important property of a bounding volume hierarchy is that, if the bounding volume of an inner node is evaluated as fully occluded (and the observer's position is outside the volume), all objects stored in the corresponding subtree can also immediately be discarded as invisible. If only a fraction of the bounding volume is visible (but no object), the visibility of the nodes in the subtree can not be predetermined and have to be evaluated through additional tests. To increase the accuracy of the occlusion tests, we use a modified version of the hull tree (see Chapter 6). A result can be seen in Figure 7.2a, where the tighter volumes can be compared to the larger bounding boxes.

The hull tree's tighter bounding volumes are employed to increase the accuracy of the occlusion tests (decreasing the number of necessary tests) and thereby increasing the efficiency of the occlusion culling. Additionally we apply another technique to slightly increase the number of objects that are incorrectly classified as visible, but without increasing the number of necessary occlusion tests. We randomly choose some objects (weighted by their surface area) to be stored at a higher level in the tree than they would normally belong to, due to their size. This is comparable to the technique used by the randomized sample tree. Experiments showed that, during the walk-through, intentionally adding these false-positives results is like a simple pre-fetching mechanism which may increase the image quality while only marginally influencing the overall performance of the system. For example, if the observer is standing in front of a wall covering the whole screen, the issued occlusion queries may evaluate some objects as visible although they are occluded by the wall. When the observer steps through the wall, the visualization node can immediately render the former wrongly classified objects, while all other newly visible objects only appear later (after the next test results have arrived). In the meantime, the images can contain significant errors, but despite this a user can retain orientation. The interior approximations are determined as described in Section 6.4.

7.4 Data Distribution and Rendering

As mentioned, the system aims at visualizing scenes with a complexity exceeding the capacities of a single node's main memory. The data is therefore distributed over the nodes in the following manner:

The visualization node stores the data of those objects which are currently classified as visible in the graphics memory. As soon as an object is identified as invisible, its data is deleted.

Each back-end node stores a subset of the scene's objects, the precomputed interior approximations of the corresponding remaining objects and the hierarchical data structure in their main memory. The assignment of the objects to the back-end nodes is done by using a pseudo-random hash function, which leads to a uniform distribution where each object is stored on exactly one back-end node. The scene is thereby completely stored by the back-end nodes and can be efficiently transferred to the visualization node without the need of any low latency hard disk access. Furthermore, every back-end node has a simplified representation of the whole scene in order to be able to perform the occlusion culling.

Algorithm 7.1 RENDERING-LOOP - VISUALIZATION NODE

Variables: objectMap (objectId $\mapsto$ objectData)

```
1: loop
2:    if data is available from a back-end node then
3:       receive invisible objects' IDs and delete them from objectMap
4:       receive and add new objects to objectMap
5:       send current camera position to back-end node
6:    end if
7:    clear buffers, set camera, ...
8:    render all objects in objectMap
9:    handle user input to update camera position
10: end loop
```

Rendering on the visualization node

The rendering loop of the visualization node is simple and straightforward (see Algorithm 7.1). In every frame, it checks if a back-end node has finished its current visibility calculations. If new data is available, the IDs of the objects that were classified as invisible are received and the corresponding objects are removed. Then the mesh data of newly appearing objects is received and stored in the graphics card's memory. Afterwards, the back-end node is informed of the observer's new position (and viewing direction) to begin with the next visibility calculation. If the data of more than one back-end node is available, its processing is postponed to the next frame to reduce the fluctuations of the frame rate. Finally, all stored objects are rendered and the camera position is updated according to the user's input before the rendering loop starts over.

In the proposed setting, in which a cluster with a high network bandwidth is given, the actual time needed for receiving the updated mesh data in most situations does not influence the overall frame rate. If the network between the visualization node and the back-end nodes becomes a bottleneck, or if a more stable frame rate is required, this

problem can be reduced by receiving the data on the visualization node asynchronously to the rendering process. In system's evaluation, we show that network's load is so low that a standard 1 GBit Ethernet, using TCP/IP, is quick enough to meet our needs.

Occlusion culling on the back-end nodes

Due to the relatively low graphic performance of the back-end nodes and scene's high complexity, we chose to use an occlusion culling algorithm which determines the visibility of the whole scene adaptively over several frames instead of in one frame, similar to the occlusion culling algorithm for the hull tree (see Section 6.5). During the culling process, the visibility of the hull tree's exterior approximations and of their corresponding objects are determined. The depth in the tree at which elements may be classified as visible is thereby adaptively changed by only one level towards the leaves per frame. On one hand, this leads to a very efficient culling process with almost no pipeline stalls. On the other hand, it may take several frames until all objects (especially the smallest ones at the leaf nodes) are classified as visible after they actually became in view.

The rendering algorithm works in four phases: Initializing the depth buffer and testing formerly visible nodes, testing potentially visible objects, fetching and evaluating the test results and then finally sending the data to the visualization node. The algorithm's pseudo-code is given in Algorithm 7.2 and 7.3.

In the first phase the depth buffer is filled by rendering the objects which were visible in the last frame. This is done in a front-to-back order and a hardware assisted occlusion query is initiated for each rendered object to determine if it became occluded by other objects. The rendered objects can be both original meshes stored on this back-end node or the corresponding interior approximations. To prevent a possible pipeline stall caused by retrieving the results of occlusion queries before they passed through the entire rendering pipeline, the retrieval of the results is postponed to a later phase.

The second phase is used to test objects inside visible hull tree nodes and those nodes whose parent in the tree is visible or which have an invisible child by recursively traversing the tree (see Algorithm 7.3). The set of visible marked nodes can thereby be extended or reduced by one level of the tree per frame. For the occlusion test of the nodes, the precomputed bounding volumes are used.

In the third phase the results of all issued occlusion queries are evaluated. For tested tree nodes, just their visibility flag is updated. The visibility flag of every tested object is also updated and all visible objects are collected into the list of visible objects used for the next frame. Those objects whose original mesh is stored on this specific back-end node are processed further. If they are found visible, but their data is not present on the visualization node, their data is prepared for sending. If they are found invisible and their data is present on the visualization node, their ID is added to the list of disappeared objects.

In the last phase, the back-end node sends the IDs of disappearing objects to the visualization node, followed by the mesh-data of the appearing objects. Finally, it receives the new camera parameters before the visibility testing loop starts over again.

Algorithm 7.2 TESTING-LOOP - BACK-END NODE

Variables: visList (list of visible objects), delList (list of object-ids to delete), sendList (list of objects to send), testQueue (queue of tested nodes and objects)

```
1:  loop
2:    clear buffer, set camera, ...
      // i. Init depth buffer & test previously visible objects
3:    for all objects in visList in front-to-back order do
4:      init occlusion-query and render orig. mesh or approx. to depth buffer
5:      add object to testQueue
6:      mark object as rendered
7:    end for
8:    clear visList
      // ii. Test potentially visible nodes and objects
9:    TraverseTreeFrontToBack(RootNode)
      // iii. Fetch and evaluate test results
10:   for all elements in testQueue do
11:     if occlusion-query result is visible then
12:       mark element as visible
13:       if element is interior approx.  then
14:         add object to visList
15:       else if element is original mesh then
16:         add object to visList
17:         if not stored on visuNode then
18:           add object to sendList
19:           mark object as stored on visuNode
20:         end if
21:       end if
22:     else
23:       mark element as invisible
24:       if is orig. mesh and is stored on visuNode then
25:         add object-id to delList
26:         mark object as not stored on visuNode
27:       end if
28:     end if
29:   end for
      // iv. Send the data to the visualization node
30:   send ids in delList
31:   send complete data in sendList
32:   receive new camera position (and direction)
33: end loop
```

Algorithm 7.3 TraverseTreeFrontToBack(node)

```
Variables: testQueue (queue of tested nodes and objects)
 1: if cameraPosition inside node then
 2:   mark node and all node's object as visible
 3:   for all node's children in front-to-back order do
 4:     TraverseTreeFrontToBack(child)
 5:   end for
 6: else if node in viewing frustum then
 7:   if node marked as visible then
 8:     for all node's objects not marked as rendered do
 9:       init occlusion-query and render orig. mesh or approx. to depth buffer
10:       add object to testQueue
11:     end for
12:     if at least one child is invisible or isLeaf then
13:       init occlusion-query for node's b. volume
14:       add node to testQueue
15:     end if
16:     for all node's children in front-to-back order do
17:       TraverseTreeFrontToBack(child)
18:     end for
19:   else
20:     init occlusion-query for node's b. volume
21:     add node to testQueue
22:   end if
23: end if
```

7.5 Evaluation

In this evaluation we analyze our system's properties and performance. First, we introduce our system environment, the used scene, the properties of the precomputed object approximations, as well as the used camera path. We show that the chosen pseudo-randomized distribution of the data leads to evenly distributed memory consumption and work load on the back-end nodes. Next we analyze the achieved rendering performance and network load of our system. Here we present the measured times for rendering the images and receiving data, as well as the amounts of data sent across the network. We show that our rendering performance allows for real-time interaction and that the network is not saturated. Finally we evaluate the *visibility delay*, where we measure the time until all visible objects have been send to the visualization node. We compare the visibility delay when the camera is *teleported* to random scene positions to the delay during walk-through.

Benchmark

For our evaluation we used the small PC cluster configuration of the PC^2 as described in Section 3.2. Additionally, we used a PC cluster consisting of 17 nodes. Below we refer to this configuration as the *standard network*. In this configuration the visualization node is equipped with an Intel i7 (4× 2.8 GHz), 12 GiB DDR3 RAM, and a NVidia GeForce 480 GTX. The 16 back-end nodes are equipped with an Intel Core 2 CPU (2× 2.66 GHz), 4 GiB DDR2 RAM, and a NVidia Quadro FX 3500. All nodes are connected via 1 GBit Ethernet. The installed operating system is 64-bit OpenSuSE-11.2.
Our application is written in C++, using the GCC-4.4, OpenMPI, and OpenGL. For our walk-through tests we use the model of a Boeing 777 whose storage requirements is ≈8.5 GiB. The total size of the interior approximations is ≈2.2 GiB, the size of the exterior approximation is ≈320 MiB.

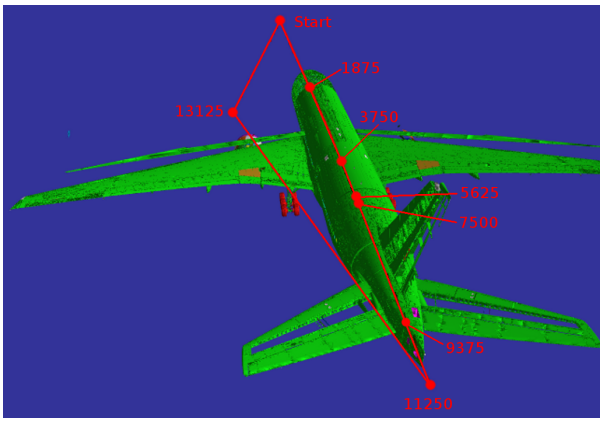


Figure 7.3: Visualization of the camera path through the Boeing 777 used for our tests.

For our tests we perform a walk-through as shown in Figure 7.3 using 15,000 fixed camera positions and alignments. The walk-through starts in front of the model, focusing the cockpit. We move the camera straight forward through the cockpit, into the first cabin. Here we reduce movement's speed and move into the third cabin where we keep the camera position unchanged for a while. Afterwards, we move through the airplane's tail, while rotating the camera. After we leave the airplane's model, the camera is looking at the turbine of the left wing. We move forward while rotating the camera until we pass the turbine and point the camera directly into the engine. We then move along the shortest route back to the starting point of our walk-through, completing the cycle. As rapid changes of the visibility are the most challenging situations for the algorithm, we try not to use portals, like doors, windows, gateways etc., between different model sections during the walk-through, but move the camera through walls and the airplane's inventory.

The speed of the camera movement during the walk-through has a strong influence on the overall behavior of the system. For the evaluation, we chose to limit the frame rate to 30 fps, while always stepping one fixed step forward on the path each frame. On average, this corresponds to the speed of a slowly walking person.

Memory distribution and culling performance

In our first tests we evaluate the memory distribution and the culling performance on the back-end nodes.
To evaluate the memory distribution among the back-end nodes, we tested our system using 9 to 15 back-end nodes, measuring the minimal and the maximal amount of data stored by the nodes (see Figure 7.4). As expected, the measurements show that the pseudo-randomized data assignment produces a well-balanced data distribution, with a difference between the minimum and the maximum amount stored of only ≈75 MiB (≈2 % of the measured median).

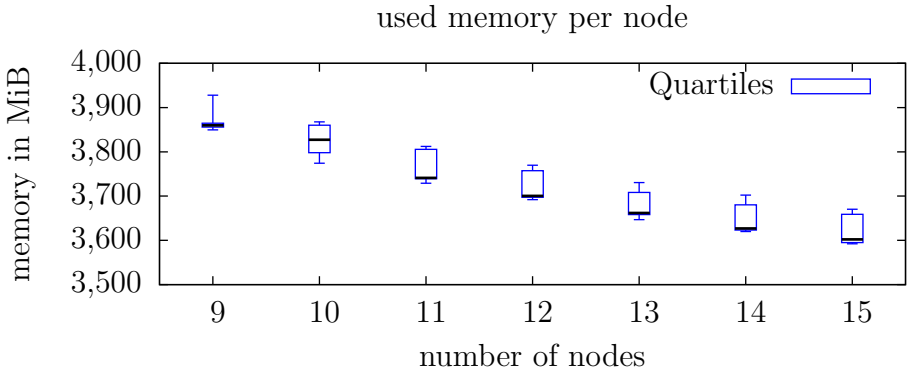


Figure 7.4: Required memory of 9 to 15 back-end nodes.

It was not possible to run this test in the standard network configuration using a comparable amount of nodes to the PC cluster configuration. The system software on the back-end nodes required much more memory than the software on the back-end nodes in the PC cluster. For this reason, we were not able to run our rendering system with less than 16 back-end nodes on the standard network configuration.

Figure 7.5 shows the time spent for the occlusion culling on the different PC cluster back-end nodes, measured during the walk-through. At each point in time the workload for all 15 back-end nodes is almost equal, although the nodes perform their tests at slightly different camera positions. The time needed for one pass ranges from some milliseconds up to over 10 seconds. It is important that the different back-end nodes work asynchronously, especially at those extreme points, so that the visualization node receives a continuous stream of new objects. Although it takes some time until the created image shows all details, the image quality rapidly achieves a suitable level, even if the user steps through a wall. These aspects are further evaluated in Section 7.5.

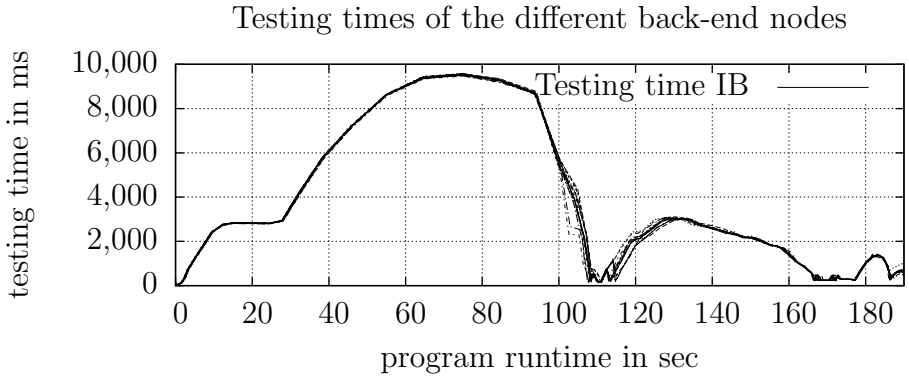


Figure 7.5: Required testing times of 15 back-end nodes during program's execution. At any time the load of all back-end nodes is almost equal.

Rendering performance

In our next tests we analyzed the performance of the visualization node. The time needed on the visualization node to receive the data from the back-end nodes and to render an image is shown in Figure 7.6. As the visualization nodes in the different configurations do not perform any additional operations for the rendering itself and the test scene's objects are all of similar complexity, the rendering time strongly correlates with the number of drawn objects (see the 30 fps plot in Figure 7.8). The additional time spent on the visualization nodes correlates with the amount of received data from the back-end nodes (see Figure 7.7). Other measurements showed that the number of deleted objects which have to be updated on the visualization node does not significantly influence the overall performance.

An important indicator for the image quality is the frequency of incoming messages at the visualization node. The highest update frequency is reached in the standard network configuration. Here, the red peaks in Figure 7.6 diagram are densest. The number of drawn objects per frame is in this configuration the highest as well. On the other hand, the number of drawn objects does not differ much if we use Infiniband or 1 GBit Ethernet in the PC cluster.

In all configurations the number of drawn objects per frame is small in relation to the complete number of objects (about 720,848 in total). In the standard network configuration, the visualization node renders more than 8,000 objects for only few frames. In this exterior view, the visualization node in this configuration renders significant more objects than in the PC cluster configuration. On our walk-through with 30 fps, the number of drawn objects never exceeds 5,700 in both PC cluster configurations, which is the reason why the frame rate is high enough to allow real-time interaction with the system.

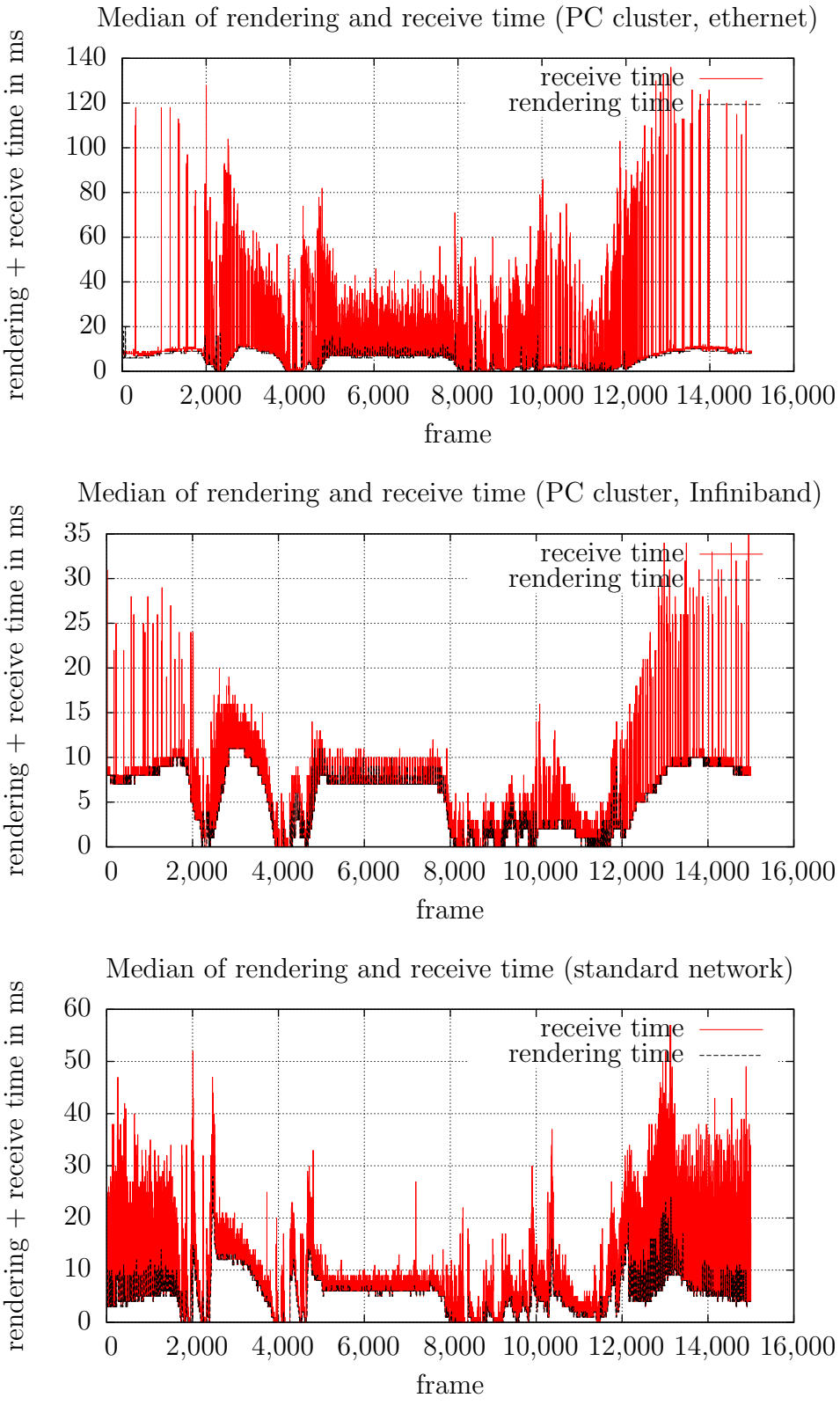


Figure 7.6: Required rendering time in the different PC cluster configurations. The measured times include the reception of updates as well as the pure rendering. The black base lines illustrate the net rendering time, where as the red peaks show the time spent to receive and process updates from the back-end nodes.

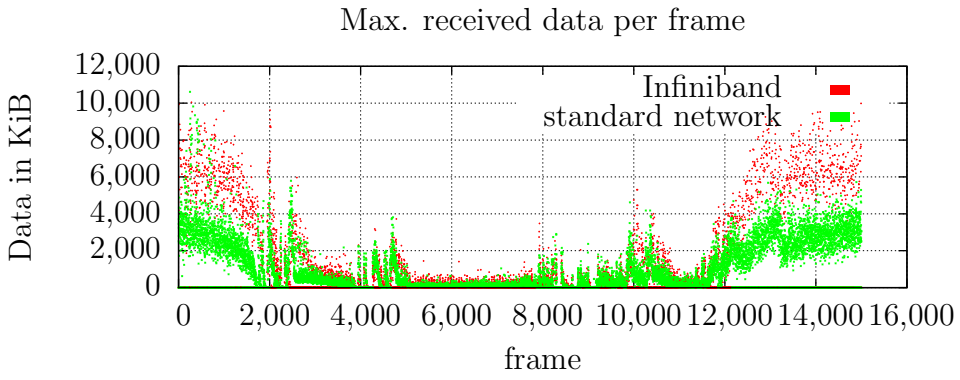
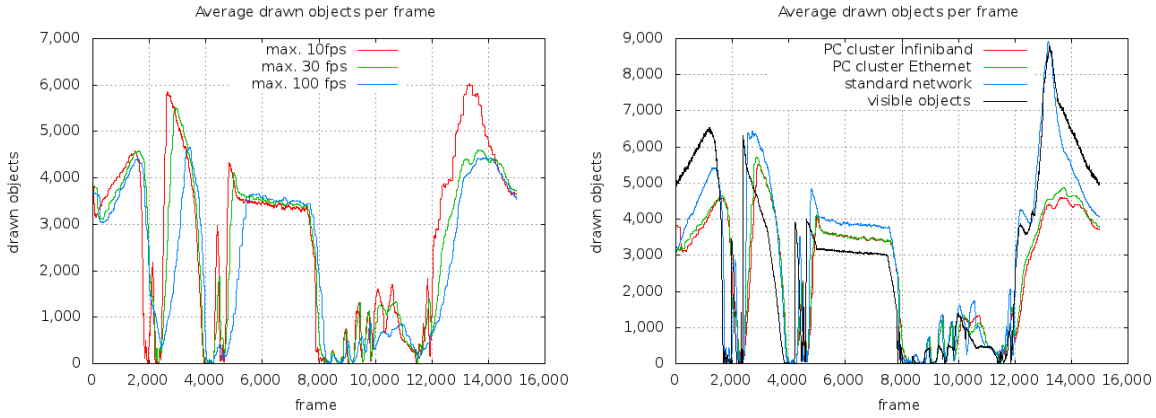


Figure 7.7: Received amount of data at the different camera positions in KiB.



(a) (b)

Figure 7.8: (a) Drawn objects per frame using the PC cluster with Infiniband. (b): Drawn objects per frame in the different PC cluster configurations (using 30 fps). The black line shows the number of visible objects.

Visibility delay

As the occlusion culling on the back-end nodes is significantly slower than the rendering on the visualization node, the outdated visibility information leads to temporary image errors. Depending on the speed that the user moves through the scene, the number of drawn objects changes (see Figure 7.8). In our system the frame rate limits the number of fixed-sized steps that can be done per second. The slower the user moves (e.g., with 10 fps), the more objects are rendered on the visualization node as the back-end nodes have more time to adapt their own visibility information closer to the actual position. If the user moves faster (e.g., with a limit of 100 fps), the visibility tests on the back-end

nodes do not have time to reach deeper levels of the scene tree and only the larger objects (and those which were randomly lifted up) are identified as visible.

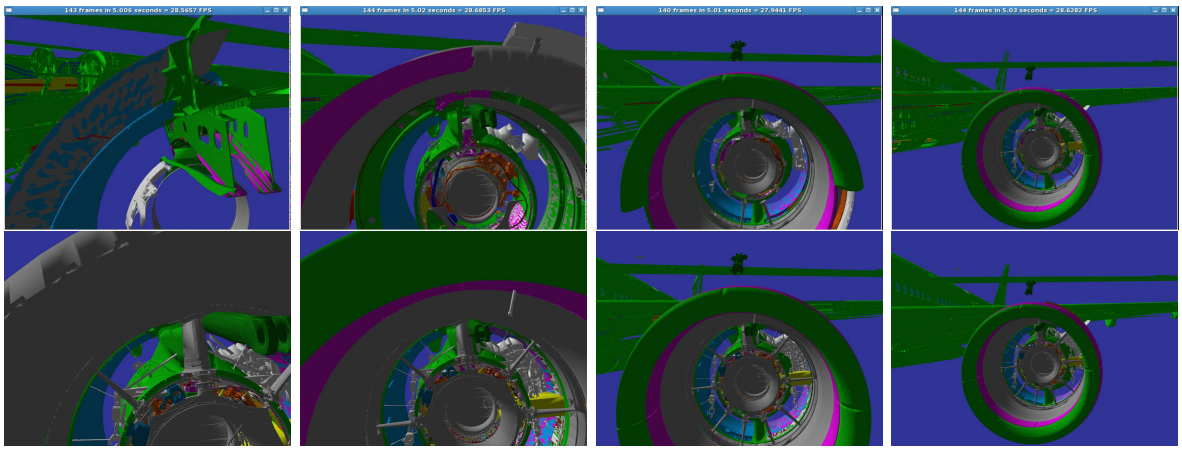


Figure 7.9: The images in the upper row show screenshots with a gap of two seconds of different camera positions, during the walk-through. The lower row shows the corresponding positions in the standard network configuration.

The evaluation setting, with a frame rate limitation to 30 fps, provides a good trade-off between real-time properties and visibility delay. The images of Figure 7.9 shows snapshots made during a walk-through with a gap of two seconds between each other. These images give an impression of the image quality achieved during runtime using the PC cluster with Infiniband. Even if a larger number of actual visible objects have not yet been identified, two effects result in a relatively high image quality during the walk-through: First, the asynchronously working back-end nodes produce an almost continuous stream of new objects. Although a single culling pass may take several seconds to complete, every few frames one node has new objects available, so the image quality continuously improves (if current visibility situation does not change too much). Second, the presented data structure in combination with the occlusion culling algorithm used prioritizes the bigger, and thereby more important, objects. The first objects to arrive from a back-end node after a severe change of the visibility are thus probably those which actually contribute most to the rendered image.

A worst-case scenario for the image quality is when the observer does not walk to a position in the scene but immediately teleports to a new position. Figure 7.10 shows the positions and measurements for such teleports. The different camera settings were chosen due to their different properties. The loading times using the PC cluster with Infiniband depends greatly on the target position. The time needed until the image is complete and no further objects are transmitted ranges from 6 seconds (in the Boeing's cockpit) to 50 seconds (when the complete Boeing is in sight). Using the PC cluster with 1 GBit Ethernet, the processing time increases slightly. In the standard network configuration, using standard 1 GBit Ethernet but with better graphics adapters better results were achieved. The time interval for a complete load is strongly decreased.

A user in general moves more steadily through the scene, allowing us to exploit spatial coherence. While the complete reload of the turbine needed 31 seconds after a teleport, it took only 8 seconds during the walk-through to load the complete engine. Another observation is that in those experiments the different back-end nodes still work in a synchronized manner and their updates arrive closely together. This results in larger popping artifacts where many objects become visible at once. During the walk-through, the nodes soon start to send their results more evenly distributed over time.
In general, the experiments have shown that the system allows a fluid navigation through a complex scene with a high frame-rate, while achieving a reasonable image quality.

7.6 Contribution

In the presented parallel out-of-core rendering approach we use weak back-end nodes to compute visibility tests and their main memory as secondary storage. In order to enable back-end nodes to perform occlusion tests for complex scenes despite their limited memory, we use object approximations. We achieved a fairly uniform data distribution and load balancing for most camera positions by distributing the scene's objects pseudo-randomly on the back-end nodes. Due to the exploitation of spatial coherence, our rendering system copes with large delays of the visibility tests. Because of these properties, our rendering system allows real-time interaction with large scenes while producing images whose error depend on the user's movement speed.

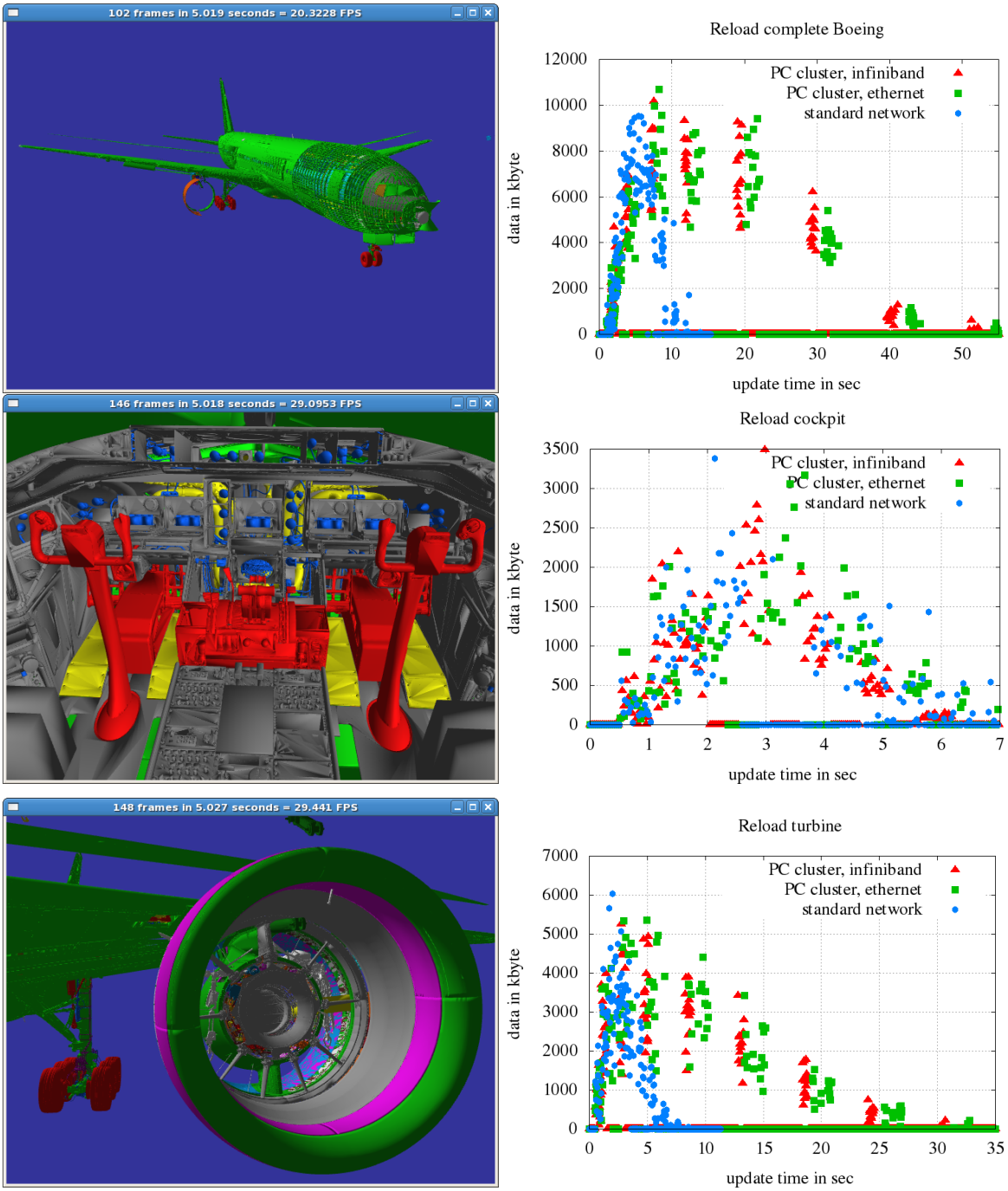


Figure 7.10: The left column shows screenshots of different camera positions, while the right column shows the required update times on these different scene's locations when the camera is teleported to those positions. The plots' y-axis represents the amount of received data and is plotted in KiB. The x-axis shows the time since the teleport in seconds.

Scenario III:
Large Dynamic Scenes

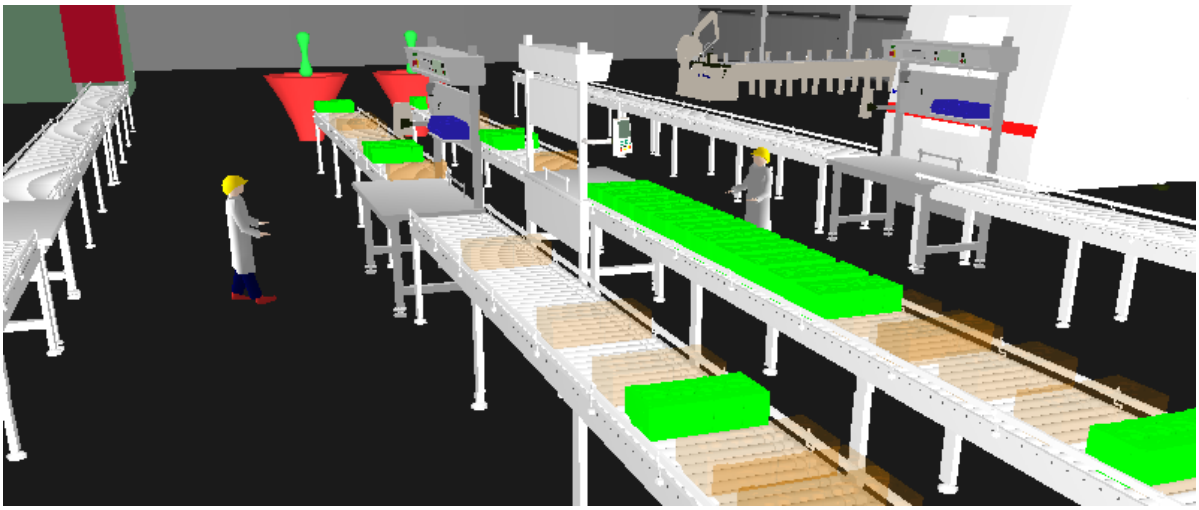


Figure 7.11: Rendering static scenes is already challenging. Dynamic scenes make this challenge even more difficult.

The rendering of large, static, three-dimensional scenes is already a well-known challenge. The previous chapters introduced techniques to processes such scenes in real-time. If a scene also includes many moving objects, the problem become even worse. To place the dynamic objects correctly, their positions must be updated in every frame. In our case the objects are controlled by an external simulation, written in Java, that sends the updates via a network.

The case-study where we render large scenes with dynamic objects in parallel is the simulation of material flows [DFH+08, SFH+09]. This is one well-established method for planning, safeguarding and improving production processes. This is an application of discrete event simulation which has been introduced by Law and Kelton [LK00]. Problems studied are, for instance, lot size planning, layout improvement, queue size evaluation, human resource planning, and so forth.

During the last years, the 3D-visualization of material flow simulations has had a growing importance. It helps the user in understanding the underlying behavior of the modeled system better and more quickly than other methods. Moreover, the 3D-view on the simulation model offers helps in communicating results and simulation based decisions to persons who are not experts in simulation. As the use of material flow simulations in industry increases, the size and the complexity of models is also growing. The complexity of not only the simulation model, but also the 3D representation, is high because they are often imported from CAD-applications.

8 Visualization of Multiple Synchronous Simulations

In the previous parts our techniques were focused on large static scenes, without any dynamics. In this chapter we present a rendering technique that allows to render multiple dynamic *discrete event simulations* simultaneously using a PC cluster. One main aspect for our application of discrete event simulation is the intensive use of random variables to model, for instance, processing times or break down intervals. This randomness leads to the following problem: a single simulation run must not be regarded as the representation of model behavior because it is possible to view the best case, the worst case, or anywhere in-between. In classic simulation study, one would repeat simulation runs in order to allow the assessment of each simulation run in the context of a set of simulation runs. These replications only differ in their random variables, in such a manner that the flow of jobs is affected but not the type of machines. An average over the whole set is created and it is assumed that this mean is a valid representation of the real system and can be used to make reasonable decisions.

The need for studies with a set of simulations and the benefit of simulation visualization cannot be satisfied by current simulation tools because just one single simulation run can be visualized. The system presented in this paper was designed to offer a solution to this problem: It enables the easily and fast analyzing of model behavior and it supports the user to concentrate on significant representations.

The general idea of the developed approach is to visualize not only one simulation run, but to animate all simulation runs in one 3D-visualization interface by using special techniques such as transparency or tinting of each dynamic object of each simulation run [DHL+06, FLH+07]. By aggregating transparent dynamic objects a "visual average" is created (see Figure 8.1). It will then be possible to visualize the stochastic derivations of the simulation study in one user interface. Moreover, the identification of extreme simulation runs can be done faster and the direct analysis for reasons of extreme behavior is supported. The identification of special simulation runs can be assisted by tinting dynamic objects in different colors, depending of their simulation. To compare specific simulations, multiple simulations can be visualized next to each other (see Figure 8.2). Another feature of our system is that the number of simulation replications can be altered during runtime. This is useful when the user wants to analyze system parameters, which he has not thought of at the begin of the simulation. To increase the number of simulations, a cloning method was implemented.

Our system has to simulate and visualize a large, and possibly growing, number of complex models simultaneously. Each task cannot be accomplished by a single computer.

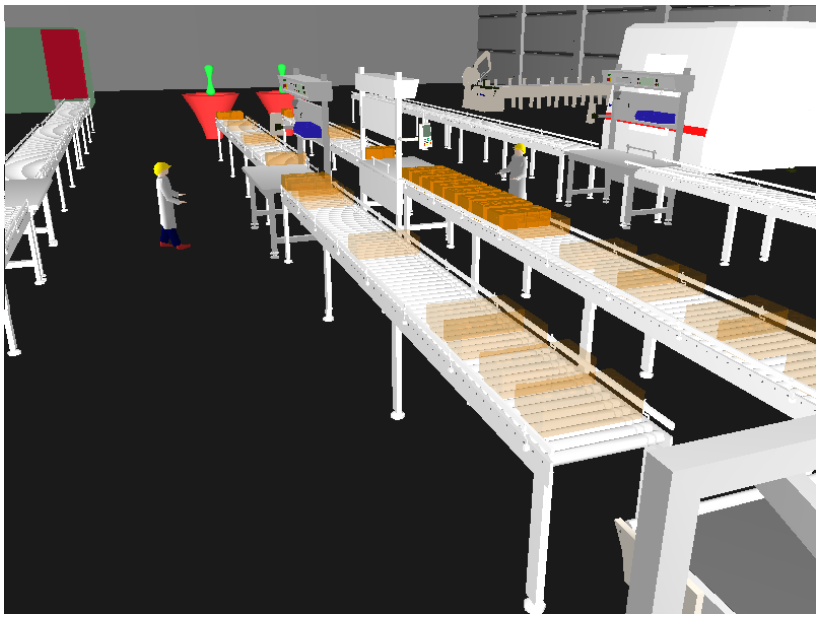


Figure 8.1: Merging Dynamic Objects.

The execution of the simulation and the rendering is done on a PC cluster and a thin-client is used for visualization and interaction. By using a PC cluster, a good scalability of the development process is possible. The thin-client makes mobile application possible. In order to do this we must solve the following problems:

- We must render large scenes that contain more static objects than a single node is able to process.
- We must be able to display simulations that contain more dynamic objects than one single node can process.
- We must join the images of static and dynamic scene parts quickly and efficiently.

8.1 Overview and Summary of Results

For the parallel simulations and the rendering system we use the large configuration of PC^2's Arminius PC cluster. For the controlling and interaction, an additional external visualization client is used. The communication between PC cluster and the external visualization client is realized by a TCP/IP connection. The nodes in the cluster communicate over Infiniband interfaces. The communication-flow is shown in Figure 8.3. For this rendering system, the PC cluster nodes are separated into three groups: a master node, a large amount of static-rendering nodes, and a number of dynamic-rendering nodes. The rendering performance of all utilized nodes is low. For each simulation there is exactly one dynamic-rendering node. The external client is connected to the master node. On each static-rendering node, a static-scene renderer is started and there are

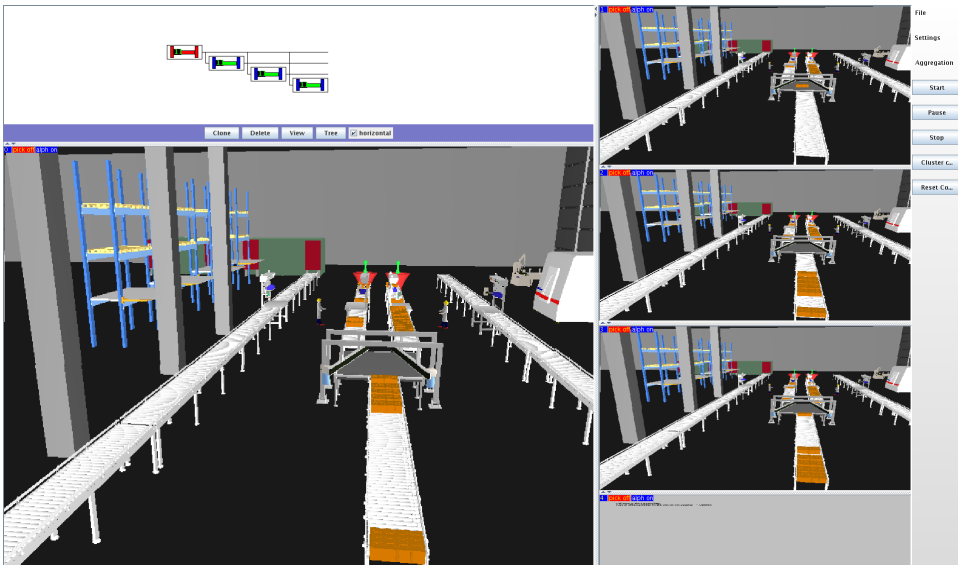


Figure 8.2: The visualization client.

no other processes started by the program on these nodes. On each dynamic-rendering node, a dynamic-scene renderer and simulation kernel is started (see Figure 8.4).

The master node receives commands from the client and sends them to the static-rendering nodes. Load-balancing of the static-rendering nodes, composing of the pictures from the static and the dynamic-rendering nodes, and sending of composed pictures to the client are also done by the master node.

The static and the dynamic parts of a 3D-scene are strictly separated. It is assumed that the overall complexity of the static objects is higher than that of the dynamic objects. The static objects do not differ from one simulation to another, so it is possible to render only one picture of these objects for all simulations. In contrast, each dynamic object is associated to exactly one simulation and typically simulations have different behavior.

After receiving a user command, the static-rendering nodes render the static parts of a simulation. This is repeated for every frame. The frame and the depth buffer are sent to the master node. Additionally, the depth buffer and the last user command are sent to the dynamic-rendering nodes. The nodes wait for new commands after their message has been sent.

Every dynamic-scene renderer is connected to the simulation kernel which has been started on the same node. There is no need for a simulation to be running on this kernel as it is possible that a kernel is waiting for a simulation as input. This happens when, for example, a user wants to clone a simulation. Cloning means to create a copy of a simulation which acts exactly like the original. If a simulation shall be cloned to another simulation kernel on another dynamic-rendering node, all simulation kernels in the PC cluster have to be suspended at the same time. The simultaneous suspension of all kernels keeps the simulation synchronous. When all kernels are suspended, the selected simulation will be serialized to a byte-stream. This stream will be send to

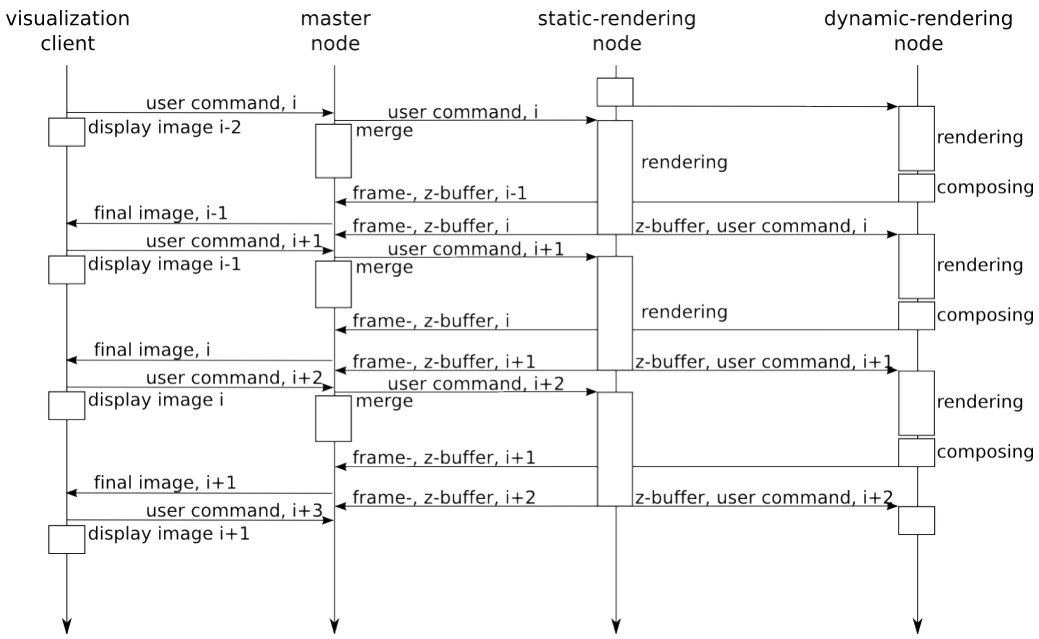


Figure 8.3: Communication Diagram.

a dynamic-rendering node whose simulation kernel is not running a simulation. The stream will be deserialized in the simulation kernel and all simulation kernels will be resumed simultaneously.

If a simulation is running on a dynamic-rendering node, the dynamic-scene renderer receives the signals of the simulation kernel, the depth buffers and the user commands from the static-rendering nodes. After composing the incoming depth buffers the dynamic-rendering nodes evaluate the received user command and compute a picture of the dynamic objects of the simulation it is associated to. After finalizing the rendering process, the depth buffer of the rendered frame is compared with the composed depth buffer of the static-rendering nodes. By doing this, occluded parts will be erased. The pictures of the dynamic rending nodes are composed hierarchically by MPI-mechanisms. The final picture of the visible dynamic parts will be sent to the master node. Here the static and the dynamic pictures will be merged and sent to the client (see Figures 8.5 and 8.6 also showing occlusion of dynamic objects by static objects).

A user is able to interact with and modify the simulations. All of these interactions have to be lead through the PC cluster to the dynamic-rendering nodes where they are transmitted to the simulation kernels.

To render large, static scenes we rendered these scene parts in sort-first manner. Thus we do not use all available PC cluster nodes. We balance their rendering load using a heuristic that determines the load of the different rendering nodes by the previously needed rendering time. The dynamic scene parts are rendered by a second group of back-end nodes. Each node in this second group renders the dynamic objects of exactly one simulation. The produced images are combined in sort-last manner. Tests have shown that the fastest merging of the images is done by FPGAs.

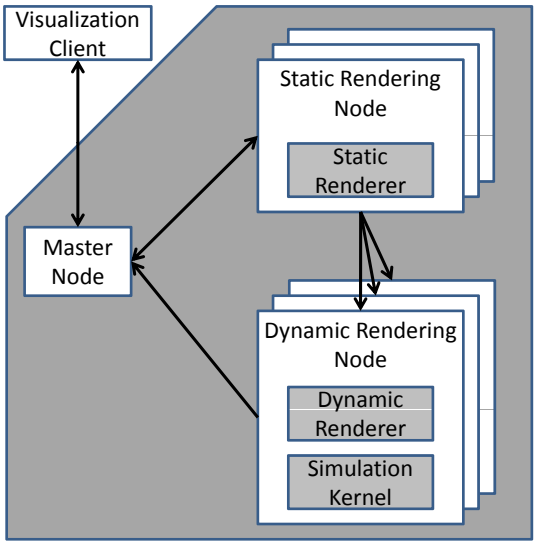


Figure 8.4: Simplified visualization of the system configuration. The master node is the communication center which handles all data into and out of the PC cluster.

8.2 Related Work

In this section, a short overview of the related work in the areas of simulation visualization and parallel and distributed simulation shall be given.

The technique of Interactive simulations with 3D visualization has successfully found its way into discrete simulation technology over the last decade. Before that the discrete simulators rendered the model in a 2D fashion with icons and lines. Common arguments for 3D visualization of simulation models are that they provide a better understanding of the regarded problem by all stakeholders and the easier identification of errors [KM00] (for example the detection of accumulations of packets on conveyor belts). By allowing to change simulation's parameters during runtime the number of repeated simulation runs is decreased. Thus, the total evaluation time decreases as well [DDLF09]. The changes to the simulation model can be done with a 3D editor.

Recent results of experimental studies that tested the impacts of Virtual Reality (VR) on Discrete-Event Simulation show that it is easier and faster to spot errors in a 3D/VR model than in 2D [AB05]. Despite the advantages of 3D visualization, many remain cautious because of a shallow learning curve for 3D software. However, Renken et al. have shown that 3D models can be used for motion planning in production facilities [FRL+10].

Most of today's discrete event simulation systems (for example Quest, Automod, and others) already provide integrated 3D visualizations. In these systems, a simulation and the rendering are running on the same machine. For performance reasons, simulation and rendering can be executed on different machines and bidirectionally coupled with each other [SSLR05]. When doing so, the synchronization of both tools (i.e. the time advance in both tools) must be coordinated.

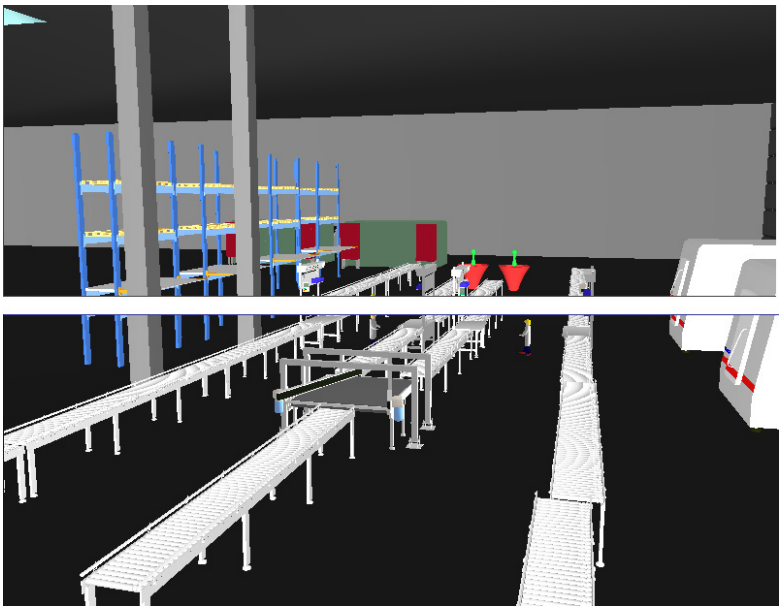


Figure 8.5: Showcase of two static-scene renderers' frame parts.

Parallel and distributed simulation is used to speed up the computation, in other words, to reduce the time until reliable results are available. There are two methods to exploit parallel processors for simulation. First, one simulation can be distributed on a multiprocessor cluster to speed-up the simulation [Fuj98]. To obtain reliable results, the simulation runs are executed sequentially. Second, as implemented in our system, on every processor of a cluster one replicated simulation can be executed [GH91] (i.e., simulation runs are executed in parallel).

8.3 System Architecture

The system consists of program parts written in C/C++ and *Java*^{TM}. The different components communicate via network. The static and dynamic scene parts are strictly separated.

The simulation kernel

The simulation kernel used is a Java based discrete event simulation tool named d^{3}FACT *insight* [DMMH05]. This tool has be developed in-house by members of Prof. Dangelmaier's group. d^{3}FACT *insight* is specialized in modeling and simulating material flow systems, especially job shop problems. A model consists of a graph of entities representing machines, conveyor belts, buffers, etc. Every entity is a static object with respect to rendering and an instant of an entity-class. The jobs are represented by token objects, dynamic objects with respect to rendering. Token allocate the entities of the graph and are sent from entity to entity, creating the material flow. This sending, receiving and alteration of tokens is done by event routines, which are executed at discrete

times when according events occur. To visualize a model, each entity and token has a 3D-representation and a position in $\mathbb{R}^3$. The logic data of models (i.e., entities description and simulation input data) is separated from the 3D-data. Data of both data types are stored in a central database and are linked by referencing the DB-key of the 3D-representations in the logic entity description.

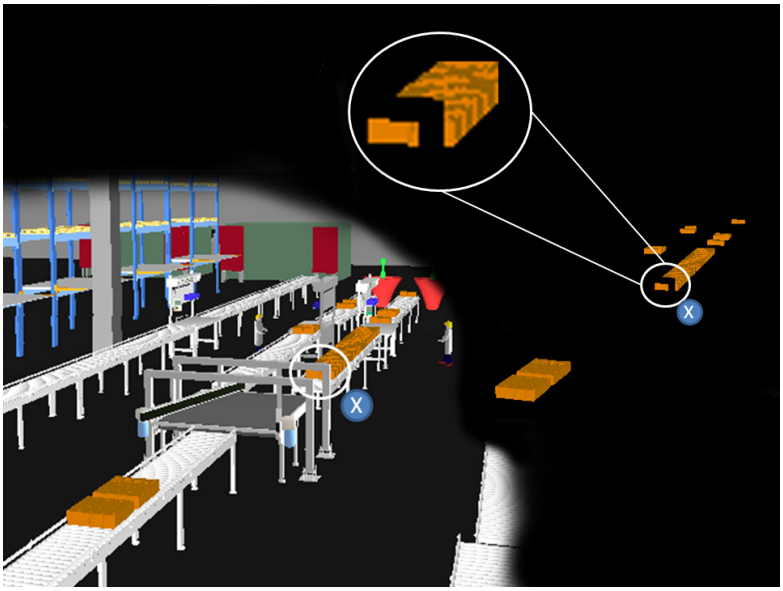


Figure 8.6: Frame of the dynamic-scene renderer (on black) and the complete frame.

Like in every other common simulation tool, there is a set of random distributions (e.g., normal, exponential, triangular) to model processing times, breakdown behavior, etc. To each random variable a specific seed is assigned, allowing the generation of different random values in each variable. This is used in simulation studies when a set of simulation runs is used to analyze system behavior. In a study, every variable gets a different seed in every run, thus resulting in different model behavior. To capture model behavior, every entity has several statistic variables, like utilization, idle time, breakdown time.

To enable online visualization of a simulation, there is a simulation-to-real-time ratio parameter in the simulation kernel. This parameter, though not typical in discrete event simulation, is necessary to create realistic animations. Normally a next event time advance function is used, which advances the simulation clock directly from one event to the next [LK00].

To connect to a simulation for online control, user interaction, visualization and data collection (or, in more general terms, message reception and transmission) a well defined message protocol is used. To receive these messages, a local socket connection is necessary.

One special kernel module enables the cloning of simulations. The state of a simulation can be saved (i.e., the variable values, the list of scheduled events, the position of dynamic objects). This saved simulation state can be transferred to another simulation kernel,

which is not running a simulation yet. This simulation kernel can load the saved state and initialize a simulation equal to the original. After cloning, both simulation kernels can run the "same" simulation. If complete equality is not desired, the cloned simulation can be altered by changing the seeds of the random variables. If the original simulation is suspended until the clone is ready, both simulations run nearly synchronously because of the time ratio parameter of the simulation kernel.

The master node

The master node is the main communication center in our system. It sends the rendered pictures to an external client and receives the inputs of a user (movement) and propagates them to the rendering nodes. Additionally, the rendering job sizes of the static-scene renderer are also organized by this special and unique node (load balancing). The composition of dynamic and static subpictures is also performed on this node. The view space is divided by a recursive algorithm. The splitting is related to the technique that Abraham et. al. [ACCC04] present in their work. The number of static rendering nodes is $N = 2^n$. For the first frame of the visualization, the tiles are equally sized for all nodes. Every node measures for every frame the time it needs to render its tile. The needed rendering time is sent to the master node to determine the new tile sizes. The N nodes are split in two fixed groups, g_1 and g_2, where the nodes of g_1 have rendered the upper half of the screen and the nodes of g_2 the lower half.

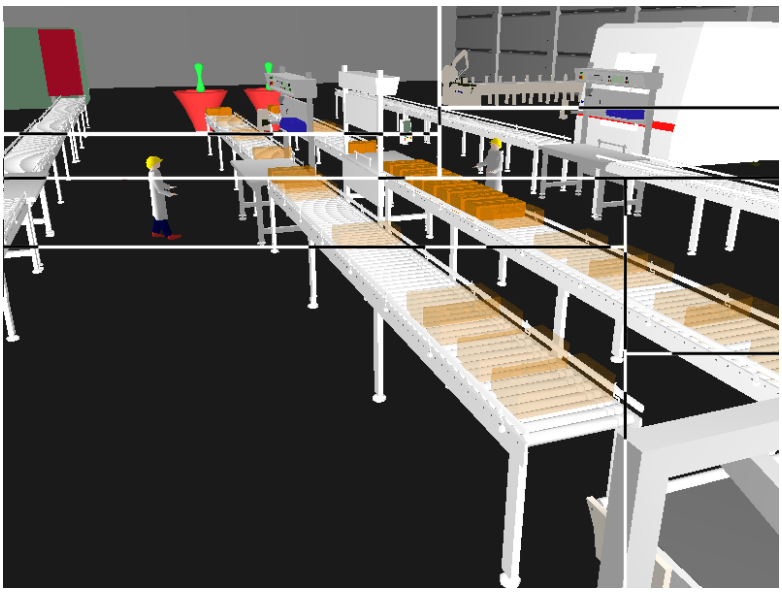


Figure 8.7: Partition of tiles.

This arrangement in groups is persistent for the whole simulation. In Figure 8.7, an example partition after several balancing steps is shown. The differently colored areas visualize the different groups where every group has 2^{n-1} members. For both groups, the average of the last frames' required rendering times is calculated. The calculated average of g_1 will be called m_1 and for g_2 will be called m_2. If m_1 is bigger than m_2,

typically the nodes of g_1 have had to render more geometry than the nodes of g_2. In this case the border between this groups will be shifted in relation to the difference of the averages. The area of g_1 will be shrunk and the area of g_2 will expand. If the average rendering time of g_2 is bigger the behavior is reversed. If the averages are (nearly) equal, the border will not be shifted. This harmonization will be done recursively until a group only consists of one node. Computing the load-balancing in this way is very fast and tests show that the load of all nodes is well balanced after a few frames [ACCC04].

The dynamic-scene renderer

Every dynamic-scene renderer has its own exclusive simulation. On every dynamic-rendering node a native MPI process runs for the rendering and a simulation process in a Java-VM. Both processes communicate locally over a TCP/IP socket using a compact protocol. The rendering and the simulation processes exchange binary-coded message over this channel. After starting the MPI process it will instantiate a simulation kernel in a Java-VM immediately. The dynamic-scene renderer connects a threaded socket-client to a socket-server threaded by the simulation kernel. Between visualization and simulation, all commands and events are transmitted over this channel.

First the simulation kernel, which is running on the dynamic-rendering node with the lowest ID, loads all meshes which are needed to visualize the running simulation from a database and stores them on the hard disk. When storing has been finished, a message is sent from the simulation kernel to the connected dynamic-scene renderer. This dynamic-scene renderer notifies all other renderes – static and dynamic – that the loading of the scene data can be initiated.

When all renderers have finished loading and organizing their data, they wait for user commands and the separated depth buffers of frame's static parts. The itemized depth buffers have to be combined into a single buffer for the whole frame. The depth buffer is needed to erase dynamic objects that are occluded by static objects (the frame buffer is not needed). The dynamic-scene renderer handles the user commands and renders a picture of the dynamic objects of its corresponding simulation in front of a black background. When the rendering process has been finished, a dynamic-scene renderer reads back the frame and the depth buffer from the memory of the graphics adapter to its local memory. The merging of the separated frames can be done in several ways depending on the chosen visualization technique.

If all simulations will be shown in only one frame, a MPI-Reduce function is called where the depth buffer values of all frames of the dynamic-scene renderer are compared with one another. The comparison is organized hierarchically in a tree. In this way a pixel will get the color of the object that is closest to the view-plane. The root of the reducing function (the node where the final result is stored) sends the merged frame and depth buffer to the master node and all nodes wait for new instructions.

The behavior of the system is similar if all simulations shall be visualized with transparencies in one frame. Instead of merging the dynamic objects opaquely (occluding each other) the several frames are merged proportionally with transparencies. If there are n simulations that should be visualized in one frame, the alpha value of all pixels

is set to $\frac{1}{n}$, where 0 is transparent and 1 is opaque. For the computation of the color values of all of the pixels, the *Porter Duff algorithm* is used in a hierarchical tree as well. Similar to opaque merging, the root of the reduction sends the final image to the master node (see Figure 8.1).

The transparent visualization is reasonable up to 16 simulations. Our system allows merging of more than 16 transparent frames, but for a user it is nearly impossible to distinguish between one or two tokens at the same position.

If a lot of simulations are visualized in only one frame, a user can lose the overview. However, sometimes a user wants to know how a specific simulation differs from all the others. This rendering system offers an mechanism to tint the dynamic objects of one simulation in an eye-catching color. This way it is possible for a user to see whether the behavior of a simulation corresponds with the others.

The merging of the different dynamic-scene renderers' images consumes a considerable amount time. For this reason we looked into alternative methods to accelerate this subroutine. These results are presented in Section 8.4.

The static-scene renderer

The static-scene renderer processes only the static parts of a scene (machines, racks, etc.). These objects are rendered by a sort-first rendering approach. This means that the viewing plane is divided in separated tiles. In this system, every static-scene renderer is responsible for exactly one tile, whose relative position never changes.

After receiving a *load-ready message* from a dynamic-rendering node, all static-rendering nodes load the data for the whole scene into their memory and organize them in a hierarchical data structure which can be chosen by the user (octree or k-d tree). This data structure is needed for frustum culling of every tile of a picture. Every static-rendering node receives a user command and a tile position with its dimension from the master node. The view-port will be trimmed and the geometry in this area of the view-plane will be specified (by doing a frustum intersection test) traversing the built data structure. Every static-scene renderer computes its subframe, reads the image data from the graphics adapter's memory to its main memory. It then sends the frame buffer, the depth buffer and the needed rendering time to the master node. The master node receives all tiles from the static-render nodes and combines them to a complete image of the static parts of the scene. However, the dynamic objects are still needed for displaying a correct frame. Thus, the depth buffer is required to add the images of the dynamic-rendering nodes. The pure rendering times are needed to balance the load of the static-scene renderer.
The depth buffer is also sent to all dynamic-rendering nodes, just the same as the last user command. When sending is completed, the static-render node waits for new input from the master node.

The visualization client

The external control application is a thin-client written in Java. This decision has been made to reduce the platform dependency. Additionally we decided that the client does not need any 3D-graphic-accelerator. This decision makes it possible to develop

control-interfaces that run on weak computers, such as PDAs or cell-phones. The client only provides an user-interface to control the cluster, the visualization and the itemized simulations (see Figure 8.2). All computations are done by the processors in the PC cluster. The client does not even know about the properties (utilization, speeds, ...) of the several simulations. If there is the need for any information from the cluster or the simulations, it has to be requested from the master node.
All images to be displayed on the client application are transmitted from the master node as raw RGB-images, we discard pixels' alpha values to reduce the network load. After receiving an image, it will be visualized and a new user command is sent to the master node. If there was no user interaction, a new image is requested anyway.
Using the client application it is also possible to select which simulations shall be shown or modified. The properties of machines can be requested and, after this, a user can modify them. For example a user can change the distribution ratio of a switch or the processing time of a machine. After subscribing to machine properties, the user will be informed about all changes until he unsubscribes from this information.

8.4 Joining the Partial Images

The rendering system renders the static scene parts in a sort-first manner. This way, composing the different tiles can be done with less effort. In contrast, composing the images (including the dynamic part) of the image consumes a significant amount of time. In order to accelerate this image processing step, we test if different hardware components can be utilized [SSPP09, SSPP11]. Here we aim to analyze the composing speed and the rendering speed. As benchmark we render different sections of the UNC Power Plant. We decrease the hierarchy tree's depth by composing many images on one node. We analyze the acceleration that can be achieved if images are merged via a pixel's depth value comparison. These tests are made separately in a different environment. In one step we try to combine two up to eight images on one node, while we usually combine two images on one node per step. In order to test a wide range of hardware components we use the *XtremeData's XD1000 Architecture* as visualization node. This node is equipped with a 2.2 GHz AMD Opteron CPU and 4 GiB main memory. An Altera Stratix II EP2S180-3 FPGA and additional 4 GiB external memory is placed in a second Opteron socket. The CPU and FPGA communicate via a 16-bit-wide HyperTransport link. Additionally, this node is equipped with a NVidia GeForce 8800 GTS (using 16× PCI-e). In this configuration, only one back-end node is available. It is equipped with an Intel Clovertown featuring 2 quad core CPUs running at 2.66 GHz and 8 GiB RAM. The nodes are connected via Infiniband with a peak bandwidth of 10 GBit/s. We perform the tests with a resolution of 800×600, $1{,}024 \times 768$, and $1{,}280 \times 1{,}024$ using 32 bit/pixel. For the composition we used a simple algorithm (see Algorithm 8.1).

We evaluate six different configurations to accelerate the composition: CPU without its streaming unit, CPU using SSE2, GPU using GLSL-shader, GPU using CUDA, and FPGA. We achieve the best results with the FGPA. The second best results are achieved with the CPU without the streaming unit.

Algorithm 8.1 Simple sort-last image composing algorithm.

```
Variables: color array SrcC1, SrcC2 depth array SrcD1, SrcD2
1: for i = 0; i < SrcC1.length; i++ do
2:    if SrcD1[i] > SrcD2[i] then
3:       SrcC1[i] = SrcC2[i];
4:       SrcD1[i] = SrcD2[i];
5:    end if
6: end for
```

The usage of processor's streaming units does not pay off. This is due to the number of CPU instructions which are performed in every loop. To perform the composition using SSE2 required more than twice the processor instructions than the comparison and copy operations in the merge algorithm. If the chosen pixels are equally distributed over both partial images, without SSE2 we can expect that half of the copy instructions can be skipped. On the other hand, no operations can be skipped because of SSE2's SIMD characteristic.

The composition time using the GPU does not differ much when CUDA or GLSL is used. The problem here is not the composition itself (the junction of several images on the GPU can be performed rapidly). The problem is the host-to-device data transfer. Composing four and seven 800×600 pixel images, we achieve roughly $11.4 - 17.1\,fps$ using the CPU, while we achieve only round about $9.7 - 15.5\,fps$ if we use CUDA. Four images with a resolution of $1{,}280 \times 1{,}024$ can be composed and display using the CPU with a rate of $6.4\,fps$, while we achieve only $5.8\,fps$ if the GPU is used.

The usage of the FPGA outperforms the CPU. The benefit achieved with the FPGA is displayed in the diagram of Figure 8.8.

Using the FPGA, the rendering speed increases linearly with the number of composed images until the network is saturated. If the CPU is used, the speed-up collapses earlier. By increasing the number of processors, every node has to process fewer objects. Additionally, the overhead to produce an image (like copying buffers) does not change.

8.5 Contribution

In this chapter we have presented a system for the simultaneous visualization of several parallelly executed simulations. By using multiple processors for simulation execution and parallel rendering, huge models can be simulated and visualized. Because of this, new techniques to display aggregated data of many simulation runs had to be developed. We separated the static scene parts from the dynamic. While we render the static parts in sort-first manner, we render each simulation's dynamic objects independently. To accelerate the composition of the different images, we determined that the fastest way is the usage of a FPGA in comparison to other techniques.

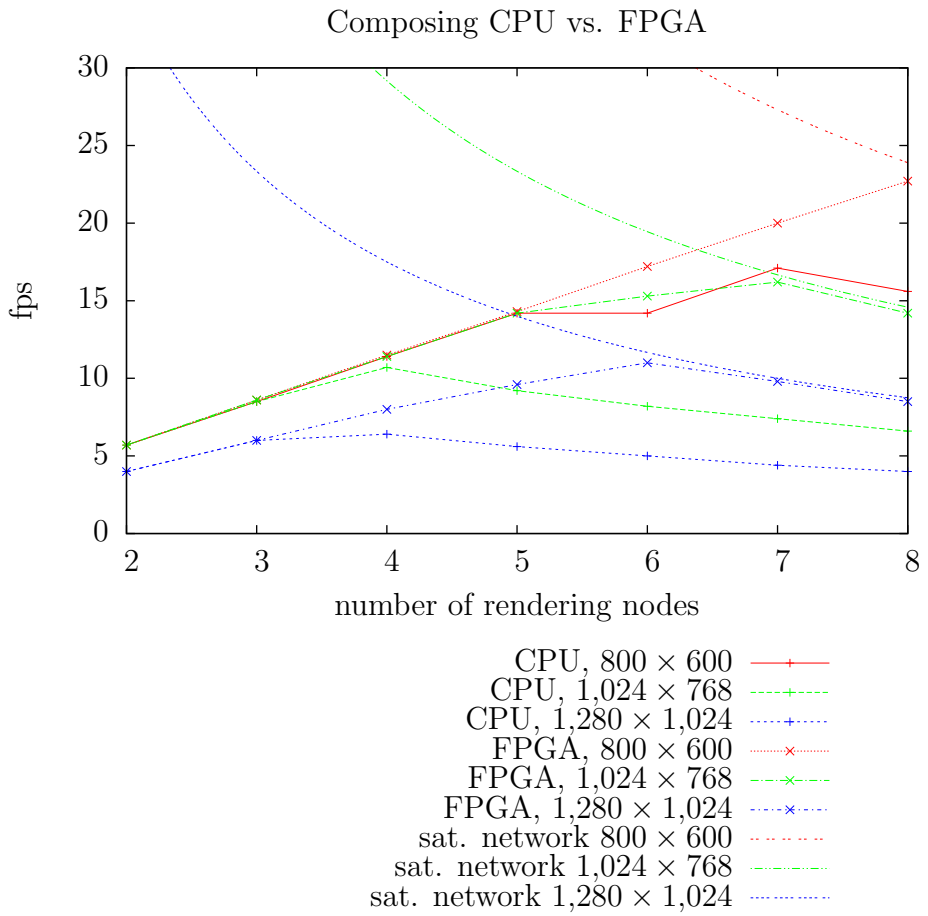


Figure 8.8: Comparison of the composition speed using the CPU without SSE2 and the FPGA. The speedup of FPGA is stopped by the network's saturation.

9 Conclusions

In this thesis we have shown how heterogeneous PC clusters can be utilized for parallel image rendering. In this context we do not require that all nodes are equipped with high end hardware. In the developed approaches, only a small group of powerful nodes is necessary, the rest of the PC cluster nodes can offer weak graphic performance. Due to this support of mixed hardware configurations it is possible to upgrade only a few processors instead of the complete PC cluster system. This allows the users to use hardware's latest features like GPU-extensions, shaders, streaming units. In this way, the upgraded group can be tested sufficiently while the other nodes run unmodified. These properties can influence a system's overall stability and performance positively. Additionally, these configurations save energy and costs, which is an important topic for PC cluster providers and their users. The providers can also use a PC cluster for a longer time because the system improves constantly. Based on this it is easier for providers and users to adjust their software for future PC cluster system configurations.

9.1 Contribution

Collectively, we developed four parallel rendering systems, one sequential rendering system, and a spatial hierarchical data structure. Additionally, we evaluated how sort-last's image combination can be accelerated through the usage of different hardware.

With the developed parallel rendering systems we answered the following questions:

- Can heterogeneous PC clusters be utilized for parallel real-time rendering?
- What tasks can be handled by the weak nodes?
- How can asynchronous communication protocols be applied?
- How can the rendering, data, and network load be distributed among the different nodes?

In this thesis we have shown that heterogeneous PC clusters, consisting of a few powerful nodes and many weak nodes, can be used for parallel real-time image rendering. We have introduced different methods where many weak nodes support a few powerful nodes. These methods include techniques as simplification generation, visibility test, and storing data. Additionally, we have shown that few nodes equipped with special purpose hardware can influence the rendering performance positively. Our parallel rendering systems for static scenes improve the image quality if the number of back-end nodes is

increased. This is different to other systems that increase the frame rate if the number of nodes is increased.

The weak nodes can be utilized for tasks which are not time-critical. In the reliefboard approach the weak back-end nodes compute the simplifications asynchronously, while the display images are computer on the powerful visualization nodes. In the out-of-core rendering systems, the back-end nodes serve as "smart" secondary memory that filter requested objects by performing visibility tests. Our first introduced parallel out-of-core rendering systems used older depth buffers to determine visible objects. To accelerate the visibility test of our secondly introduced parallel out-of-core rendering system we developed a spatial, hierarchical data structure. The hull tree reduces the amount of data that each back-end node has to store in its primary memory.

Due to the back-end nodes' weak performance, the suggested computations require much processing time. As a result, the back-end nodes have to perform tasks whose results are suitable for the displaying visualization nodes even when they arrive several seconds later. Because of slight fluctuations in the various task sizes, it is unusual that results arrive at the visualization nodes on time. For that reason, the communication does not have much influence on the rendering process for the visualization nodes. In the reliefboard approach the back-end nodes compute the simplification which can be used for many frames on the visualization node. The parallel out-of-core rendering systems reduce the load of the network by determining visible objects on the back-end nodes and sending only visible objects.
In the parallel rendering system to visualize multiple simulations we separated the static scene parts from the dynamic parts. While the static scene is rendered in a balancing sort-first manner, the dynamic scene parts of each simulation is processed on a separate back-end node. The different images are combined in sort-last manner. To accelerate the merging process we use a node equipped with an FPGA, while other promising techniques (like streaming units or GPUs) were not found to be beneficial, during our tests.

A randomized distribution of the tasks and the data leads to good load balancing in the presented systems. In the reliefboard approach and the first presented parallel out-of-core approach, we distributed the different objects randomized and redundant among the back-end nodes. In the parallel out-of-core rendering system that uses the hull tree we renounced redundant storing of the data items. Due to the huge amount of data items and the relative small number of requests, the load was sufficiently evenly distributed.

In addition to these parallel rendering methods, we also developed a sequential occlusion culling technique with a associated spatial hierarchical data structure (which was later modified for parallel rendering purposes). The data structure covers its included objects more tightly than other data structures, such as regular octrees. Thus it is more suitable for occlusion culling.

9.2 Open Questions and Future Work

We tested the developed rendering techniques with CAD-models without the use of image manipulating procedures, such as a different post-processing shaders. It would be interesting to analyze how our techniques can be combined with these techniques. Image manipulating shaders can improve the image quality, which can lead to a higher level of realism for the observer. Additionally, it would be interesting to evaluate how the user perceives the approximations. On the one hand, massive popping effects can disturb users. On the other hand, a low frame rate is also disturbing and, in some situations, worse than the spontaneous appearance of objects.

The developed techniques are tested on two different types of nodes. The load-balancing algorithms of reliefboards and the first out-of-core, rendering system presented should be expandable to support more types of nodes. The out-of-core renderer using the hull tree would need a new data access protocol.

In the future, multi-core CPUs will have a huge amount of independently working cores. This computation parallelism should allow transferring the reliefboards technique directly on a single computer. A transfer of the other parallel rendering techniques would probably require more efforts.

For the static scene renderer, we always distributed the different objects randomly. It would be interesting if there are other easily computable distributions (especially for large scenes streaming algorithms for clustering) that could be used.

For two of our rendering techniques we use interior approximations. The computation of these simplifications needs a significant amount of time. Additionally, for certain mesh configurations it is not possible to determinate this approximation or to reach the aimed triangle count. Faster and more robust methods are required here.

Bibliography

[AB05] Justice I. Akpan and Roger J. Brooks. Experimental investigation of the impact of virtual reality on discrete-event simulation. In *Proceeding of the 2005 Winter Simulation Conference*, WSC '05, pages 1968–1975. Winter Simulation Conference, 2005.

[ABB+07] Carlos Andújar, Javier Boo, Pere Brunet, Marta Fairén, Isabel Navazo, Pere Pau Vázquez, and Alvar Vinacua. Omni-directional relief impostors. *Computer Graphics Forum*, 26(3):553–560, September 2007.

[ABW90] Helmut Alt, Johannes Blömer, and Hubert Wagener. Approximation of convex polygons. In *Proceedings of the 17th International Colloquium on Automata, Languages and Programming*, volume 443 of *ICALP '90*, pages 703–716. Springer-Verlag, 1990.

[ACCC04] Frederico Abraham, Waldemar Celes, Renato Cerqueira, and Joao Luiz Campos. A load-balancing strategy for sort-first distributed rendering. In *Proceedings of the XVII Brazilian Symposium on Computer Graphics and Image Processing*, SIBGRAPI '04, pages 292–299, Washington, DC, USA, 2004. IEEE Computer Society.

[ACW+99] Daniel Aliaga, Jon Cohen, Andrew Wilson, Eric Baker, Hansong Zhang, Carl Erikson, Kenny Hoff, Tom Hudson, Wolfgang Stuerzlinger, Rui Bastos, Mary Whitton, Fred Brooks, and Dinesh Manocha. Mmr: an interactive massive model rendering system using geometric and image-based acceleration. In *Proceedings of the 1999 symposium on Interactive 3D graphics*, I3D '99, pages 199–206, New York, NY, USA, 1999. ACM.

[Ada05] Paul Adams. Performance comparisons of visualization architectures. In *Proceedings of the Users Group Conference 2005*, DOD-UGC '05, pages 388–393, Washington, DC, USA, 2005. IEEE Computer Society.

[AMHH08] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008.

[BHP07] Beat Brüderlin, Mathias Heyer, and Sebastian Pfützner. Interviews3d: A platform for interactive handling of massive data sets. *IEEE Comput. Graph. Appl.*, 27:48–59, November 2007.

[BMadHS97] Petra Berenbrink, Friedhelm Meyer auf der Heide, and Klaus Schröder. Allocating weighted jobs in parallel. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '97, pages 302–310, New York, NY, USA, 1997. ACM.

[BMW+09] Jiří Bittner, Oliver Mattausch, Peter Wonka, Vlastimil Havran, and Michael Wimmer. Adaptive global visibility sampling. In *ACM Transactions on Graphics*, volume 28, pages 94:1–94:10, New York, NY, USA, August 2009. ACM.

[BWPP04] Jiří Bittner, Michael Wimmer, Harald Piringer, and Werner Purgathofer. Coherent hierarchical culling: Hardware occlusion queries made useful. *Computer Graphics Forum*, 23(3):615–624, 2004.

[CDL+96] Bradford Chamberlain, Tony DeRose, Dani Lischinski, David Salesin, and John Snyder. Fast rendering of complex environments using a spatial hierarchy. In *Proceedings of the Conference on Graphics Interface '96*, pages 132–141, Toronto, Ont., Canada, Canada, 1996. Canadian Information Processing Society.

[CDR02] Alan Chalmers, Timothy Davis, and Erik Reinhard, editors. *Practical parallel rendering*. A. K. Peters, Ltd., Natick, MA, USA, 2002.

[CKS02a] Wagner T. Corrêa, James T. Klosowski, and Cláudio T. Silva. iwalk: Interactive out-of-core rendering of large models. Technical Report TR-653-02, Princeton University, http://www.cs.princeton.edu/~wtcorrea/papers/iwalk.pdf, 2002.

[CKS02b] Wagner T. Corrêa, James T. Klosowski, and Cláudio T. Silva. Out-of-core sort-first parallel rendering for cluster-based tiled displays. In *Proceedings of the Eurographics Workshop on Parallel Graphics and Visualization*, EGPGV '02, pages 89–96, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[CKS03] Wagner T. Corrêa, James T. Klosowski, and Claudio T. Silva. Visibility-based prefetching for interactive out-of-core rendering. In *Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, PVG '03, pages 1–8, Washington, DC, USA, 2003. IEEE Computer Society.

[Cla76] James H. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19:547–554, October 1976.

[COCSD03] Daniel Cohen-Or, Yiorgos Chrysanthou, Cláudio T. Silva, and Frédo Durand. A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):412–431, 2003.

[Coo84] Robert L. Cook. Shade trees. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, pages 223–231, New York, NY, USA, 1984. ACM.

[CVM+96] Jonathan Cohen, Amitabh Varshney, Dinesh Manocha, Greg Turk, Hans Weber, Pankaj Agarwal, Frederick Brooks, and William Wright. Simplification envelopes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pages 119–128, New York, NY, USA, 1996. ACM.

[DDLF09] Wilhelm Dangelmaier, Robin Delius, Christoph Laroque, and Matthias Fischer. Concepts for model verification and validation during simulation runtime. In *European Simulation and Modelling Conference*, ESM '09, pages 49–53. EUROSIS, EUROSIS-ETI, 26 - 28 October 2009.

[DDSD03] Xavier Décoret, Frédo Durand, François X. Sillion, and Julie Dorsey. Billboard clouds for extreme model simplification. *ACM Transactions on Graphics*, 22:689–696, July 2003.

[DDTP00] Frédo Durand, George Drettakis, Joëlle Thollot, and Claude Puech. Conservative visibility preprocessing using extended projections. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 239–248, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[DE84] William H. E. Day and Herbert Edelsbrunner. Efficient algorithms for agglomerative hierarchical clustering methods. *Journal of Classification*, 1:7–24, 1984.

[Del00] Mark. Deloura. *Game Programming Gems*. Charles River Media, Inc., Rockland, MA, USA, 2000.

[DFH+08] Wilhelm Dangelmaier, Matthias Fischer, Daniel Huber, Christoph Laroque, and Tim Süß. Aggregated 3d-visualization of a distributed simulation experiment of a queuing system. In S. J. Mason, R. Hill, L. Moench, and O. Rose, editors, *Proceedings of the 2008 Winter Simulation Conference*, WSC´ 08, pages 2012 – 2020. IEEE, Omnipress, 2008.

[DH00] Michael Doggett and Johannes Hirche. Adaptive view dependent tessellation of displacement maps. In *Proceedings of the ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, HWWS '00, pages 59–66, New York, NY, USA, 2000. ACM.

[DHL+06] Wilhelm Dangelmaier, Daniel Huber, Christoph Laroque, Mark Aufenanger, Matthias Fischer, Jens Krokowski, and Michael Kortenjan. d^{3}fact insight goes parallel - aggregation of multiple simulations. In Thomas

Schulze, Graham Horton, Bernhard Preim, and Stefan Schlechtweg, editors, *Proceedings of the Simulation und Visualisierung 2006*, SimVis '06, pages 79–88. SCS European Publishing House, 2006.

[DMadH93] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. Simple, efficient shared memory simulations. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '93, pages 110–119, New York, NY, USA, 1993. ACM.

[DMMH05] Wilhelm Dangelmaier, Kiran Mahajan, Bengt Mueck, and Daniel Huber. d^3fact insight - simulation of huge scale models in cooperative work. In Jörg Krüger, Alexei Lisounkin, and Gerhard Schreck, editors, *Proceedings of the Industrial Simulation Conference*, ISC '05, pages 150–158. EUROSIS-ETI, 9 - 11 June 2005.

[DMS01] Laura Downs, Tomas Möller, and Carlo H. Séquin. Occlusion horizons for driving through urban scenery. In *Proceedings of the 2001 symposium on Interactive 3D Graphics*, I3D '01, pages 121–124, New York, NY, USA, 2001. ACM.

[DN09] Philippe Decaudin and Fabrice Neyret. Volumetric billboards. *Computer Graphics Forum*, 28(8):2079–2089, 2009.

[DS10] Dominic Dumrauf and Tim Süß. On the complexity of local search for weighted standard set problems. In *Proceedings of the 6th Conference on Computability in Europe*, pages 132–140, 30 June - 4 July 2010.

[DY08] Steve Dominick and Ruigang Yang. Anywhere pixel router. In *Proceedings of the 5th ACM/IEEE International Workshop on Projector Camera Systems*, PROCAMS '08, pages 7:1–7:2, New York, NY, USA, 2008. ACM.

[EMB01] Carl Erikson, Dinesh Manocha, and William V. Baxter, III. Hlods for faster display of large static and dynamic environments. In *Proceedings of the 2001 Symposium on Interactive 3D graphics*, I3D '01, pages 111–120, New York, NY, USA, 2001. ACM.

[EP08] Stefan Eilemann and Renato Pajarola. Direct send compositing for parallel sort-last rendering. In *ACM SIGGRAPH ASIA 2008 courses*, SIGGRAPH Asia '08, pages 39:1–39:8, New York, NY, USA, 2008. ACM.

[FLH+07] Matthias Fischer, Christoph Laroque, Daniel Huber, Jens Krokowski, Bengt Mueck, Michael Kortenjan, Mark Aufenanger, and Wilhelm Dangelmaier. Interactive refinement of a material flow simulation model by comparing multiple simulation runs in one 3d environment. In *Proceedings of the European Simulation and Modelling Conference*, ESM '07, pages 499–505. EUROSIS, October 2007.

[Fly72] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9):948–960, September 1972.

[FRL+10] Matthias Fischer, Hendrik Renken, Christoph Laroque, Guido Schaumann, and Wilhelm Dangelmaier. Automated 3d-motion planning for ramps and stairs in intra-logistics material flow simulations. In *Proceedings of the 2010 Winter Simulation Conference*, WSC '10, pages 1648 – 1660. IEEE, Omnipress, 5 - 8 December 2010.

[Fuj98] Richard M. Fujimoto. Parallel and distributed simulation. In Jerry Banks, editor, *Handbook of Simulation*, pages 429–464. John Wiley & Sons, 1998.

[GBK06] Michael Guthe, Ákos Balázs, and Reinhard Klein. Near optimal hierarchical culling: Performance driven use of hardware occlusion queries. In T. Akenine-Möller and W. Heidrich, editors, *Proceeding of the Eurographics Symposium on Rendering*, pages 207–214. The Eurographics Association, June 2006.

[GBSF05] Anselm Grundhöfer, Benjamin Brombach, Robert Scheibe, and Bernd Fröhlich. Level of detail based occlusion culling for dynamic scenes. In *Proceedings of the 3rd International conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia*, GRAPHITE '05, pages 37–45, New York, NY, USA, 2005. ACM.

[GFB10] Johannes Ghiletiuc, Markus Färber, and Beat Brúderlin. A Highly Scalable Image-Based Remote Rendering Framework. In Jürgen Gausemeier and Michael Grafe, editors, *Augmented & Virtual Reality in der Produktentstehung*, HNI-Verlagsschriftenreihe, Paderborn, pages 317–330. Universität Paderborn, HNI Verlagsschriftenreihe, Paderborn, 2010.

[GH91] Peter W. Glynn and Philip Heidelberger. Analysis of parallel replicated simulations under a completion time constraint. *ACM Transactions on Modeling and Computer Simulations*, 1:3–23, January 1991.

[GH99] Stefan Gumhold and Tobias Hüttner. Multiresolution rendering with displacement mapping. In *Proceedings of the ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, HWWS '99, pages 55–66, New York, NY, USA, 1999. ACM.

[GM05] Enrico Gobbetti and Fabio Marton. Far voxels: a multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms. *ACM Transactions on Graphics*, 24(3):878–885, July 2005.

[GMBP10] Prashant Goswami, Maxim Makhinya, Jonas Bösch, and Renato Pajarola. Scalable parallel out-of-core terrain rendering. In *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization*, EGPGV '10,

pages 63–71, Norrköping, Sweden, May 2010. Eurographics Association, Eurographics Association.

[GSYM03] Naga K. Govindaraju, Avneesh Sud, Sung-Eui Yoon, and Dinesh Manocha. Interactive visibility culling in complex environments using occlusion-switches. In *Proceedings of the 2003 Symposium on Interactive 3D graphics*, I3D '03, pages 103–112, New York, NY, USA, 2003. ACM.

[HDD+93] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh optimization. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, pages 19–26, New York, NY, USA, 1993. ACM.

[HE05] Martin Held and Johannes Eibl. Biarc approximations of polygons within asymmetric tolerance bands. In *Computer-Aided Design*, volume 37, pages 357–371, 2005.

[Hop96] Hugues Hoppe. Progressive meshes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pages 99–108, New York, NY, USA, 1996. ACM.

[HSLM02] Karl Hillesland, Brian Salomon, Anselmo Lastra, and Dinesh Manocha. Fast and simple occlusion culling using hardware-based depth queries. Technical report, Department of Computer Science, University of North Carolina at Chapel Hill, 2002.

[HWC10] Mark Howison, Bethel E. Wes, and Hank Childs. MPI-hybrid Parallelism for Volume Rendering on Large, Multi-core Systems. In *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization*, EGPGV '10, Norrköping, Sweden, May 2010. Eurographics Association, Eurographics Association.

[JW02] Stefan Jeschke and Michael Wimmer. Textured depth meshes for real-time rendering of arbitrary scenes. In *Proceedings of the 13th Eurographics workshop on Rendering*, EGRW '02, pages 181–190, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[KBF05] David J. Kasik, William Buxton, and David R. Ferguson. Ten CAD challenges. *IEEE Computer Graphics and Applications*, 25:81–92, March 2005.

[KDG+08] David Kasik, Andreas Dietrich, Enrico Gobbetti, Fabio Marton, Dinesh Manocha, Philipp Slusallek, Abe Stephens, and Sung-Eui Yoon. Massive model visualization techniques: course notes. In *ACM SIGGRAPH 2008 classes*, SIGGRAPH '08, pages 40:1–40:188, New York, NY, USA, 2008. ACM.

[KKF+02] Jan Klein, Jens Krokowski, Matthias Fischer, Michael Wand, Rolf Wanka, and Friedhelm Meyer auf der Heide. The randomized sample tree: a data structure for interactive walkthroughs in externally stored virtual environments. In *Proceedings of the ACM symposium on Virtual Reality Software and Technology*, VRST '02, pages 137–146, New York, NY, USA, 2002. ACM.

[Kle97] Reinhard Klein. Multiresolution representations for surfaces meshes. In *Proceedings of the Spring Conference on Computer Graphics 1997*, SCCG '97, pages 57–66, 1997.

[KM00] Vineet R. Kamat and Julio C. Martinez. 3d visualization of simulated construction operations. In *Proceedings of the 2000 Winter Simulation Conference*, WSC '00, pages 1933–1937, San Diego, CA, USA, 2000. Society for Computer Simulation International.

[KPH+10] Wes Kendall, Tom Peterka, Jian Huang, Han-Wei Shen, and Robert Ross. Accelerating and benchmarking radix-k image compositing at large scale. In *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization*, EGPGV '10, pages 101–110. Eurographics Association, Eurographics Association, May 2010.

[KS00] James T. Klosowski and Cláudio T. Silva. The prioritized-layered projection algorithm for visible set estimation. *IEEE Transactions on Visualization and Computer Graphics*, 6:108–123, April 2000.

[KTI+01] Tomomichi Kaneko, Toshiyuki Takhei, Masahiko Inami, Naoki Kawakami, Yasuyuki Yanagida, Taro Maeda, and Susumu Tachi. Detailed shape representation with parallax mapping. In *Proceedings of the 11th International Conference on Artificial Reality and Telexistence*, ICAT '01, pages 205–208, 2001.

[Lai05] Samuli Laine. A general algorithm for output-sensitive visibility preprocessing. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, I3D '05, pages 31–40, New York, NY, USA, 2005. ACM.

[LE97] David Luebke and Carl Erikson. View-dependent simplification of arbitrary polygonal environments. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 199–208, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[LK00] Averill M. Law and W. David Kelton. *Simulation Modeling and Analysis*. McGraw-Hill, 2000.

[LSCO03] Tommer Leyvand, Olga Sorkine, and Daniel Cohen-Or. Ray space factorization for from-region visibility. *ACM Transactions on Graphics*, 22(3):595–604, 2003.

[Lud10] Ludic. Graphics task distribution performance scaling without compromise. Technical report, Ludic, September 2010. whitepaper.

[Lue01] David P. Luebke. A developer's survey of polygonal simplification algorithms. *IEEE Computer Graphics and Applications*, 21:24–35, May 2001.

[LWC+02] David Luebke, Benjamin Watson, Jonathan D. Cohen, Martin Reddy, and Amitabh Varshney. *Level of Detail for 3D Graphics*. Elsevier Science Inc., New York, NY, USA, 2002.

[MBDM97] John S. Montrym, Daniel R. Baum, David L. Dignam, and Christopher J. Migdal. Infinitereality: a real-time graphics system. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 293–302, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[MBM+01] Michael Meißner, Dirk Bartz, Gordon Müller, Tobias Hüttner, and Jens Einighammer. Generation of subdivision hierarchies for efficient occlusion culling of large polygonal models. Technical Report TUBSCG-1999-01, Technical University of Braunschweig, 2001.

[MBW08] Oliver Mattausch, Jiří Bittner, and Michael Wimmer. CHC++: Coherent hierarchical culling revisited. *Computer Graphics Forum*, 27(2):221–230, April 2008.

[MBWW07] Oliver Mattausch, Jiří Bittner, Peter Wonka, and Michael Wimmer. Optimized subdivisions for preprocessed visibility. In *Proceedings of the Graphics Interface*, GI '07, pages 335–342, New York, NY, USA, 2007. ACM.

[MCEF94] Steve Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, 14:23–32, July 1994.

[MCEF08] Steve Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A sorting classification of parallel rendering. In *ACM SIGGRAPH ASIA 2008 courses*, SIGGRAPH Asia '08, pages 35:1–35:11, New York, NY, USA, 2008. ACM.

[MOM+01] Shigeru Muraki, Masato Ogata, Kwan-Liu Ma, Kenji Koshizuka, Kagenori Kajihara, Xuezhen Liu, Yasutada Nagano, and Kazuro Shimokawa. Next-generation visual supercomputing using pc clusters with volume graphics hardware devices. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (CDROM)*, Supercomputing '01, pages 51–51, New York, NY, USA, 2001. ACM.

[MPHK94] Kwan-Liu Ma, James S. Painter, Charles D. Hansen, and Michael F. Krogh. Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics and Applications*, 14:59–68, July 1994.

[MPHK08] Kwan-Liu Ma, James S. Painter, Charles D. Hansen, and Michael F. Krogh. Parallel volume rendering using binary-swap image composition. In *ACM SIGGRAPH ASIA 2008 courses*, SIGGRAPH Asia '08, pages 38:1–38:9, New York, NY, USA, 2008. ACM.

[NB04] Shaun Nirenstein and Edwin H. Blake. Hardware accelerated visibility preprocessing using adaptive sampling. In *Proceedings of the 15th Eurographics Symposium on Rendering*, Rendering Techniques 2004, pages 207–216. The Eurographics Association, 2004.

[NBG02] S. Nirenstein, E. Blake, and J. Gain. Exact from-region visibility culling. In *Proceedings of the 13th Eurographics Workshop on Rendering*, EGRW '02, pages 191–202. Eurographics Association, 2002.

[OU06] Tetsuro Ogi and Takaya Uchino. Dynamic load-balanced rendering for a cave system. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, VRST '06, pages 189–192, New York, NY, USA, 2006. ACM.

[Pac96] Peter S. Pacheco. *Parallel programming with MPI.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[Paj08] Renato Pajarola. Cluster parallel rendering. In *ACM SIGGRAPH ASIA 2008 courses*, SIGGRAPH Asia '08, pages 34:1–34:12, New York, NY, USA, 2008. ACM.

[PF05] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming techniques for high-performance graphics and general-purpose computation.* Addison-Wesley Professional, first edition, 2005.

[Res09] Jon Peddie Research. Multi gpus: Needs, issues, and opportunities. Technical report, Jon Peddie Research, 2009. whitepaper.

[Ros05] Randi J. Rost. *OpenGL(R) Shading Language (2nd Edition).* Addison-Wesley Professional, 2005.

[RP05] Timothy Roden and Ian Parberry. Portholes and planes: faster dynamic evaluation of potentially visible sets. *Computers in Entertainment*, 3:3–3, April 2005.

[RRR06] Marcus Roth, Patrick Riess, and Dirk Reiners. Load balancing on cluster-based multi projector display systems. In *Proceedings of the International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, WSCG '06, pages 55–62, 2006.

[SBS06] Dirk Staneker, Dirk Bartz, and Wolfgang Straßer. Occlusion-driven scene sorting for efficient culling. In *Proceedings of the 4th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa*, AFRIGRAPH '06, pages 99–106, New York, NY, USA, 2006. ACM.

[SDDS00] Gernot Schaufler, Julie Dorsey, Xavier Decoret, and François X. Sillion. Conservative volumetric visibility with occluder fusion. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 229–238, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[SEP+01] Gordon Stoll, Matthew Eldridge, Dan Patterson, Art Webb, Steven Berman, Richard Levy, Chris Caywood, Milton Taveira, Stephen Hunt, and Pat Hanrahan. Lightning-2: a high-performance display subsystem for pc clusters. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 141–148, New York, NY, USA, 2001. ACM.

[SFH+09] Tim Süß, Matthias Fischer, Daniel Huber, Christoph Laroque, and Wilhelm Dangelmaier. Ein System zur aggregierten Visualisierung verteilter Materialflusssimulationen. In Jürgen Gausemeier and Michael Grafe, editors, *Augmented & Virtual Reality in der Produktentstehung*, volume 252, pages 111–126. Heinz Nixdorf Institut, Universität Paderborn, May 2009.

[SFL01] Rudrajit Samanta, Thomas Funkhouser, and Kai Li. Parallel rendering with k-way replication. In *Proceedings of the IEEE 2001 Symposium on Parallel and Large-Data Visualization and Graphics*, PVG '01, pages 75 – 84, Piscataway, NJ, USA, October 2001. IEEE Press.

[SFLS00] Rudrajit Samanta, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh. Hybrid sort-first and sort-last parallel rendering with a cluster of pcs. In *Proceedings of the ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, HWWS '00, pages 97–108, New York, NY, USA, August 2000. ACM.

[SGG+00] Pedro V. Sander, Xianfeng Gu, Steven J. Gortler, Hugues Hoppe, and John Snyder. Silhouette clipping. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 327–334, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[SHDG+09] Behzad Sajadi, Yan Huang, Pablo Diaz-Gutierrez, Sung-Eui Yoon, and M. Gopi. A novel page-based data structure for interactive walkthroughs. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, I3D '09, pages 23–29, New York, NY, USA, 2009. ACM.

[SJF10] Tim Süß, Claudius Jähn, and Matthias Fischer. Asynchronous parallel reliefboard computation for scene object approximation. In *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization*, EGPGV '10, pages 43–51, Norrköping, Sweden, May 2010. Eurographics Association, Eurographics Association.

[SKJ+11] Tim Süß, Clemens Koch, Claudius Jähn, Matthias Fischer, and Friedhelm Meyer auf der Heide. Ein paralleles Out-of-Core Renderingsystem für Standard-Rechnernetze. In Jürgen Gausemeier, Michael Grafe, and Friedhelm Meyer auf der Heide, editors, *Augmented & Virtual Reality in der Produktentstehung*, volume 295 of *HNI-Verlagsschriftenreihe, Paderborn*, pages 185–197. Heinz Nixdorf Institut, Universität Paderborn, May 2011.

[SKJF11] Tim Süß, Clemens Koch, Claudius Jähn, and Matthias Fischer. Approximative occlusion culling using the hull tree. In *Proceedings of the Graphics Interface 2011*, pages 79–86. Canadian Human-Computer Communications Society, May 2011.

[SLS+96] Jonathan Shade, Dani Lischinski, David H. Salesin, Tony DeRose, and John Snyder. Hierarchical image caching for accelerated walkthroughs of complex environments. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pages 75–82, New York, NY, USA, 1996. ACM.

[SSLR05] Steffen Strassburger, Thomas Schulze, Marco Lemessi, and Gordon D. Rehn. Temporally parallel coupling of discrete simulation systems with virtual reality systems. In *Proceedings of the 2005 Winter Simulation Conference*, WSC '05, pages 1949–1957. Winter Simulation Conference, 2005.

[SSPP09] Tobias Schumacher, Tim Süß, Christian Plessl, and Marco Platzner. Communication performance characterization for reconfigurable accelerator design on the XD1000. In *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 9 - 11 December 2009.

[SSPP11] Tobias Schumacher, Tim Süß, Christian Plessl, and Marco Platzner. Fpga acceleration of communication-bound streaming applications: Architecture modeling and a 3d image compositing case study. *International Journal of Reconfigurable Computing*, 2011:1–11, 2011. Article ID 760954.

[Ste96a] Volker Stemann. Parallel balanced allocations. In *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, SPAA '96, pages 261–269, New York, NY, USA, 1996. ACM.

[Ste96b] Volker Stemann. Parallel balanced allocations. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '96, pages 261–269, New York, NY, USA, 1996. ACM.

[SWF10a] Tim Süß, Timo Wiesemann, and Matthias Fischer. Evaluation of a c-load-collision-protocol for load-balancing in interactive environments. In *Proceedings of the 5th IEEE International Conference on Networking, Architecture, and Storage*, pages 448 – 456. IEEE Computer Society, IEEE Press, 15 - 17 July 2010.

[SWF10b] Tim Süß, Timo Wiesemann, and Matthias Fischer. Gewichtetes c-Collision-Protokoll zur Balancierung eines parallelen Out-of-Core-Renderingsystems. In Jürgen Gausemeier and Michael Grafe, editors, *Augmented & Virtual Reality in der Produktentstehung*, HNI-Verlagsschriftenreihe, Paderborn, pages 39–52. Universität Paderborn, HNI Verlagsschriftenreihe, Paderborn, 2010.

[SZF+99] Rudrajit Samanta, Jiannan Zheng, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh. Load balancing for multi-projector rendering systems. In *Proceedings of the ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, HWWS '99, pages 107–116, New York, NY, USA, 1999. ACM.

[SZL92] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of triangle meshes. In *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '92, pages 65–70, New York, NY, USA, 1992. ACM.

[TIH03] Akira Takeuchi, Fumihiko Ino, and Kenichi Hagihara. An improvement on binary-swap compositing for sort-last parallel rendering. In *Proceedings of the 2003 ACM Symposium on Applied Computing*, SAC '03, pages 996–1002, New York, NY, USA, 2003. ACM.

[TS91] Seth J. Teller and Carlo H. Séquin. Visibility preprocessing for interactive walkthroughs. *SIGGRAPH Computer Graphics*, 25:61–70, July 1991.

[Tur92] Greg Turk. Re-tiling polygonal surfaces. *SIGGRAPH Computer Graphics*, 26:55–64, July 1992.

[USKS06] Tamás Umenhoffer, László Szirmay-Kalos, and Gábor Szijártó. Spherical billboards and their application to rendering explosions. In *Proceedings of Graphics Interface*, GI '06, pages 57–63, Toronto, Ont., Canada, Canada, 2006. Canadian Information Processing Society.

[VM02] Gokul Varadhan and Dinesh Manocha. Out-of-core rendering of massive geometric environments. In *Proceedings of the IEEE Visualization*, VIS '02, pages 69–76, Washington, DC, USA, 2002. IEEE Computer Society.

[Wat99] Alan H. Watt. *3D Computer Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1999.

[Wel04] Terry Welsh. Parallax mapping with offset limiting: A per-pixel approximation of uneven surfaces, 2004.

[Whi92] Scott Whitman. *Multiprocessor Methods for Computer Graphics Rendering*. A. K. Peters, Ltd., Natick, MA, USA, 1992.

[Wie10] Timo Wiesemann. Effektivitätsanalyse des c-Collision Protokolls als gewichteter Datenbalancierer in einem parallelen Out-of-Core Renderer. Diploma thesis, University of Paderborn, 2010.

[WLLB97] Harvey J. Wassermann, Olaf M. Lubeck, Yong Luo, and Federico Bassetti. Performance evaluation of the sgi origin2000: a memory-centric characterization of lanl asci applications. In *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing (CDROM)*, Supercomputing '97, pages 1–11, New York, NY, USA, 1997. ACM.

[WWT+03] Lifeng Wang, Xi Wang, Xin Tong, Stephen Lin, Shimin Hu, Baining Guo, and Heung-Yeung Shum. View-dependent displacement mapping. *ACM Transactions on Graphics*, 22:334–339, 2003.

[XPQS06] Hua Xiong, Haoyu Peng, Aihong Qin, and Jiaoying Shi. Parallel occlusion culling on gpus cluster. In *Proceedings of the 2006 ACM International Conference on Virtual reality Continuum and its Applications*, VRCIA '06, pages 19–26, New York, NY, USA, 2006. ACM.

[YSK+02] Shuntaro Yamazaki, Ryusuke Sagawa, Hiroshi Kawasaki, Katsushi Ikeuchi, and Masao Sakauchi. Microfacet billboarding. In *Proceedings of the 13th Eurographics Workshop on Rendering*, EGRW '02, pages 169–180, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.