Performance challenges in distributed rendering systems

Makhinya, Maxim

Abstract: Unspecified

University of Zurich[UZH]

Department of Informatics

Performance Challenges in Distributed Rendering Systems

A dissertation submitted to the Faculty of Economics, Business Administration and Information Technology of the University of Zürich

for the degree of
Doctor of Science (Ph.D.)

by
Maxim Makhinya

Supervisor: Prof. Dr. Renato Pajarola
Co-referee: Dr. Bruno Raffin

Zürich, January 16, 2012

ABSTRACT

The amount of data acquired from a single simulation or an object scan is growing continuously due to increased computational capabilities and advances in scanning devices. This data has to be not only stored, but also displayed interactively for visual analysis and real-time exploration. However, rendering and especially communication hardware, progress at a much slower pace. Nowadays it is not possible to use a single computer with a single GPU to visualize massive modern datasets without avoiding time consuming preprocessing and significant visual quality degradation. When multiple machines are used to accelerate visualization, parallel rendering software has to be used to distribute tasks, synchronize execution and assemble partially obtained images together. Although there were many systems developed in the past, only a few are generic enough to be used by modern rendering applications.

In this thesis, various aspects of parallel rendering tasks on a small PC cluster were studied in detail. Polygonal, volume and terrain rendering applications are used to demonstrate various techniques and optimizations throughout this work, where *Equalizer* is chosen as a general-purpose parallel rendering framework, providing flexible configuration and being relatively easy to port existing applications to. Several improvements to the original framework and algorithms are presented.

First, general approaches to parallel visualization, image and data space rendering decomposition, as well as popular distributed image compositing algorithms are briefly explained.

Next, deep analysis of Equalizer library's performance is conducted in order to reveal weak and strong aspects of the framework. The importance of the efficient image compression methods is confirmed in this study, since the limited network bandwidth is the most significant bottleneck during the image compositing stage. Static data distribution is considered and several compression methods are integrated and evaluated, where the best compression schemes for different rendering scenarios are outlined; then, distributed rendering is compared to streaming approach, and an introductory study of automatic versus manual load-balancing for the case of the distributed out-of-core terrain rendering application is presented.

Further, in order to overcome the compositing bottleneck, a Region Of Interest (ROI) based method is proposed. Integrated on the level of GPU this approach allows to quickly determine which screen regions are supposed to be read-back to main memory, compressed and sent for compositing, before the actual data read-back. Being extremely fast, it allows to accelerate image assembling stage considerably. The algorithm can be applied to any rendering framework and any rendering algorithm. The only assumption used in the proposed method is that, with the increased number of resources, each resource would draw into a smaller screen area in case of data-space rendering decomposition.

Finally, a generic data fetching and caching mechanism is introduced to Equalizer framework. While the framework already provides certain support for data distribution and synchronization, the unified toolset for building out-of-core applications was missing, forcing application developers to invent or adapt their own methods every time the framework was used. Based on the proposed data management strategy a distributed out-of-core volume rendering solution for large volumes visualization is created, providing further insights into building of efficient parallel rendering applications in general.

ACKNOWLEDGMENTS

I would like to thank my advisor Renato Pajarola for this wonderful opportunity of being a PhD student at University of Zürich, for support and numerous advices that I got from him during these years.

I would also like to express my gratitude to my colleagues who supported me in research and personal life, and with whom I had successful collaborations. I especially appreciate help that was provided by Stefan Eilemann concerning Equalizer framework.

Finally, I would like to thank Bruno Raffin for being my co-adviser for this thesis.

CONTENTS

NOTATIONS

Acronyms

BS	Binary-Swap (compositing)
CAD	Computer-Aided Design
CPU	Central Processing Unit
DCT	Discrete Cosine Transform
DEM	Digital Elevation Model
DS	Direct Send (compositing)
eqPly	A polygonal rendering application supplied with Equalizer parallel rendering framework
eqRASTeR	A terrain rendering application of the Visualization and Multimedia Lab of the University of Zurich
eVolve	A volume rendering application supplied with Equalizer
EZW	Embedded Zerotree Wavelet
FPS	Frames per Second
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HDD	Hard Disk Drive
LB	Load Balancing
LOD	Level of Detail
LRU	Least Recently Used
NAS	Network Attached Storage

PC	Personal Computer
RAM	Random Access Memory
RBCT	Read-Back, Compression and Transmission
ROI	Region-Of-Interest
SAT	Summed Area Table
SSD	Solid-State Drive
VNC	Virtual Network Computing
VRAM	Video RAM
Z-Buffer	Depth Buffer

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Scientific Visualization and Data Growth

Increasing computational abilities of modern hardware, cluster computation and advances in scanning technologies lead to an explosive growth of numerical data that has to be stored, processed and visualized. Nowadays, researchers are dealing with data of dozens of gigabytes per processing unit and soon terabytes will not be something unusual. The variety of applications together with different use-case scenarios and data representations bring further challenges to the field of visualization.

Typical areas where large-scale scientific visualization is applied are engineering, medicine and natural sciences. A selection of use-case scenarios is shown in Figure 1.1.

(a) (b) (c)

Figure 1.1: *Various visualization scenarios: (a) immersive CAVE with polygonal data rendering, (b) display wall featuring terrain visualization, and (c) scalable volume rendering.*

Data growth is in particular challenging since visualization devices, despite the advances in GPU technologies, can't keep up with it. Figure 1.2 illustrates the data growth by means of polygon and voxel counts for single models; the dependency is logarithmic.

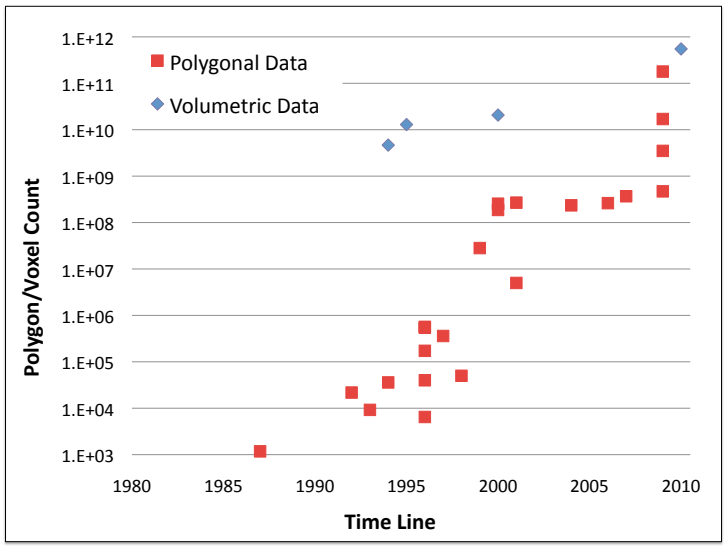


Figure 1.2: *Various popular datasets for polygonal and volumetric data, plotted according to their size and the year of acquisition.*

In order to visualize these vast amounts of data researchers have built various software and hardware solutions, nevertheless, there are fundamental problems related to the number of rendering resources utilized. The most important bottlenecks are limited network throughput and other communication overheads. The larger the dataset, the more hardware resources it takes to process and visualize it; the higher the number of hardware units being used, the more significant communication overhead becomes. Table 1.1 demonstrates this effect clearly: over the years the amount of volume data that the researchers were able to visualize increased, so did the resolution of the output image. However, when too many resources are used, as showed by [Howison et al., 2010], visualization speed dropped below interactive frame rates.

Year	Data Size	Resolution	Speed	Resources	Reference
2001	256^3	1024^2 px	4.3 fps	4 Nodes	[Magallón et al., 2001]
2004	1024^3	1024^2 px	8.0 fps	16 Nodes	[Houston, 2004]
2010	4560^3	4608^2 px	2.0 fps	216K Cores	[Howison et al., 2010]

Table 1.1: *Volume data growth, resources and performance for given setups.*

It is to be mentioned that even a single machine can provide significant render-

ing performance nowadays. Using various optimization techniques, it is possible to visualize large amounts of data with a single GPU. The price is usually paid by higher algorithmic complexity, time consuming preprocessing and lower image quality. Parallel rendering is trying to improve image quality and reduce preprocessing time by exploiting similar algorithms within individual computers and by adding another set of task distribution and synchronization techniques.

1.2 Resources of Parallel Rendering Systems

Whenever it is desired to use multiple machines for visualization, there are a few aspects to consider: how data is acquired, what the data size is, what infrastructure is already available, what space and power consumption requirements are. These would affect the choice of the visualization system setup.

Depending on the data type and the system resources, there are additional questions to resolve: where the data is coming from, which resources are used for rendering, how data is displayed, what the user interaction options are?

Finally, there are a few algorithmic aspects of any parallel rendering system to take into account: how the data visualization is separated between resources, and how the data distribution (if any) is being handled.

There are three main visualization system types to consider: *PC clusters* with GPU-based rendering; *Supercomputer-based* CPU rendering; or *Hybrid* systems, i.e. combining both, supercomputer and PC rendering cluster. Table 1.2 gives a short comparison of these solutions.

	PC clusters	Supercomputers	Hybrid
Hardware	2..30 PCs	100..300K CPUs	100..400K CPUs + 2..30 PCs
Price	Affordable	Expensive	Expensive
Rendering type	GPU	CPU	GPU
Power consumption	Low	High	High
Memory resources	Limited	Vast amounts	Vast amounts
”In Situ” visualization	Hardly	Supported	Possible
Compositing overhead	Tolerable	Large	Tolerable

Table 1.2: *Various systems' properties comparison.*

PC clusters An affordable option for moderate data size processing and visualization could be created by connecting off-the-shelf computers, so-called nodes, into a small *PC cluster*. Typically consisting of up to a few dozens of computers, it is easy to setup and maintain. Power consumption is usually not an issue either. The GPUs are the primary resources for rendering, nevertheless, they could

theoretically be used for other computation tasks as well. Data size is the main limiting factor and the data typically has to be uploaded to such a cluster prior to the visualization; Additionally, depending on the visualization task, the network interconnection could be a significant bottleneck. Programmers have to take special care about synchronization and data management between nodes, which is usually achieved by running some sort of parallel rendering framework, that may or may not require changes to original rendering code.

Supercomputer-based visualization Traditionally supercomputers were used for data simulation. Since the data is already available in the memory it is tempting to use the available computational power and fast CPU interconnection for rendering as well; at the same time such a solution can benefit from additional information about the data taken from the simulation (e.g., spatial relationships). While a single CPU is not very efficient for rendering comparing to GPU, the number of available CPUs are usually so high that it is possible to assign multiple CPUs per pixel, which, in turn, would be much more efficient.

Some of the mosts recent supercomputers are already being equipped with GPU-based NVIDIA Tesla cards for number crunching; these solutions are supposed to have a great rendering potential compared to other types of visualization systems.

The issues with supercomputer-based approaches are certainly the price of installation and the maintenance costs (upgrades are usually not cheap if at all possible), power consumption, cooling and space requirements. Regarding the rendering within a supercomputer, the problem lies in assembling of the rendered data from the individual processors. If hundreds of CPUs were used to render one single pixel, despite the fast inter-CPU communication, the resulting effort to combine the final color for this pixel from this large number of sources is cumbersome. Recent study by [Howison et al., 2010] showed that interactivity is still not achievable for such scenarios. The high number of messages being created in the system, while trying to communicate all the required information for the final image acquisition, still remains an unsolved issue.

Hybrid visualization system Creating a *hybrid system* by extending an existing supercomputer with a few rendering nodes for visualization is a common approach. It allows keeping simulation system almost intact, while having a clear separation of tasks. The data that has to be visualized can be streamed directly from the supercomputer, thus reducing storage requirements, and allowing interactive rendering as the data is being simulated. The power consumption, space and cooling requirements are inherited from the supercomputer setup, while fast GPU rendering is one of the benefits.

Hybrid systems are built on top of existing large-scale PC cluster installation. A large-scale PC cluster is used in the same role as a supercomputer, providing CPU computational resources and owning a fast network interconnection, while few additional nodes with GPUs are added and used for visualization. This type of setup was demonstrated by [Suss et al., 2010], where a large cluster was used as a data source as well as for initial culling optimizations, improving the overall rendering performance.

In this work, the focus is largely on distributed rendering with GPU-based PC clusters. This choice makes it possible to achieve interactive visualization of reasonably large datasets at low setup and maintenance costs. Consequently, PC clusters are the most popular type of visualization setups nowadays.

1.3 Challenges of Parallel Rendering

Parallel rendering can be applied in various areas, where additional performance should be gained by exploiting rendering parallelism on a scale larger than a single GPU can provide. Changing of existing applications to run on multiple GPUs and multiple machines has to be made easy. In order to make transition faster, general-purpose parallel rendering libraries should be considered, rather than writing parallel implementations from the ground up. Through this approach, the implementation details would be addressed by the framework developers, while application programmers could focus on domain specific problems.

From the generic parallel rendering framework's point of view, there are various issues that have to be addressed:

- *Image data compression*

  When multiple computers contribute to the final rendering, partial image results have to be sent between machines. These partial results are combined (composited) together in order to produce complete visualization. The compositing stage is the most time consuming task, which is performed by the rendering framework itself. Compositing strategy influences the amount of data exchange between nodes, where efficient image data compression comes into play.

  There are two main criteria that image compression has to satisfy: compression and decompression methods have to be fast enough such that the overall performance is improved, and the amount of introduced artifacts has to be as low as possible (which conflicts with high compression rates).

  Different parts of the framework and the application can affect each other's performance; therefore, it is not possible to evaluate compression separately. Properties of compression methods have to be carefully studied

within the fully functioning rendering framework, using different rendering algorithms and different distribution scenarios, since acceleration of only compression part doesn't necessary lead to overall improvement. For example, improving image data compression by making it run on multiple cores with OpenMP does increase compression speed significantly if applied in a stand-alone compression application, however the overall performance of rendering is not improved, because the framework itself, as well as the rendering algorithms, run multiple threads already, and typically there are not enough free cores to execute compression in parallel on CPU.

It is necessary to evaluate particular compression algorithms in different rendering scenarios, since different decomposition modes and different applications fill the screen differently, therefore faster but simpler methods can be beneficial in some cases, but fail to improve the overall performance in other.

Depending on the distribution scheme, the amount of compression artifacts can be tolerated differently. On the other hand, if the final values of pixels are combined from different sources (sort-last rendering decomposition) very little amount of compression error can be allowed since it would possibly introduce even more noticeable artifacts after compositing. On the other hand, when each renderer is responsible for its own screen region (sort-first decomposition), lossy compression with higher compression ratios and more artifacts is allowed. This is in particular true for scenes with strong motion, where strong movement masks compression artifacts.

- *Compositing optimization*

  The amount of information that is exchanged between two particular nodes can vary significantly depending on the assembling strategy. This is mostly related to sort-last rendering decomposition, where different parts of the same dataset are rendered on different computers. The partial images that potentially occupy the entire screen have to be assembled together. In general, methods like direct send [Eilemann and Pajarola, 2007] provide distributed compositing with constant per-node cost, only increasing number of passed messages, rather than increasing compositing cost linearly in case when all partial images would be sent to a single computer; however even in these methods compositing performance hits network throughput limits quickly.

  Special rendering and compositing techniques for particular rendering types were proposed in the past [Stompel et al., 2003; Samanta et al., 2000]. Due to requirement of tight integration of specific applications and frameworks, they can't easily be generalized into application independent frameworks.

In case of a generic parallel rendering framework, creating a compositing optimization, that does not require coupling with a particular renderer, is an important task to solve.

- *Load balancing*
  Parallel rendering in a scalable scenario (many to one rendering) requires balancing of load per rendering resource. In general, dividing of initial data equally between renderers doesn't achieve equal rendering time. Depending on the camera position and orientation some portions of the data will contain more useful information since they are, for example, closer to the screen plane and therefore slower to render, other parts could be in the background or simply outside of the view frustum, thus data separation between rendering resources has to be adjusted dynamically.

  Although different approaches for load balancing (LB) exist [Samanta et al., 1999; Marchesin et al., 2006; Hui et al., 2009; Osman and Ammar, 2002], providing efficient LB strategies for a generic application is still a largely unsolved issue. The coupling between the framework and the application has to be minimal in order to keep code reusable, while balancing should still be able to automatically equalize load well. This is especially difficult in case of out-of-core rendering applications, since the framework has no knowledge about asynchronous data fetching and caching methods exploited during rendering.

- *Application developer support*
  The parallel rendering framework has to be easy to use, while providing enough flexibility when necessary. Various building blocks for creating GUIs and data management as well as hardware abstractions are necessary. While there are several rendering frameworks available [Humphreys et al., 2002; Eilemann et al., 2009; Doerr and Kuester, 2011], none provides complete support in terms of efficient data management and load balancing for an arbitrary application. The application developers are forced to implement and adopt data distribution and management strategies every time a new application is created or ported. A significant effort from framework developers therefore should be invested in order to provide a convenient set of standard tools for integration or implementing of new types of parallel applications easier.

1.4 Contributions

In this thesis, various issues of parallel rendering frameworks and applications, outlined above, are addressed. The focus is mainly on the efficiency of the render-

ing framework and compositing stage, as well as on providing solutions for simplified creation of parallel out-of-core rendering applications within the Equalizer framework by [Eilemann et al., 2009].

The performance issues of the framework are addressed first, using sample applications for polygonal and volume rendering in a variety of rendering scenarios. New compositing optimizations and compression selection guidelines are provided as a result. Next, an existing out-of-core terrain rendering application is adapted to the Equalizer, extending its use-case options and boosting the performance using multiple computers. Load-balancing and data distribution options are then investigated in this context, and scalable solutions are proposed. Lastly, the data management system, based on a two level cache is introduced to Equalizer in order to simplify data handling for multi-GPU, multi-computer out-of-core parallel rendering applications. Details of one such application for rendering of large volumetric data, using proposed data management system, are also exposed.

Overall, the conducted evaluations in typical use case scenarios of a small GPU-based visualization cluster and different applications, provide various insights into creation of efficient parallel rendering applications; a more detailed summary is outlined below:

- *Cluster-based parallel rendering performance evaluation*

  With an extensive set of tests, a baseline for cluster parallel rendering performance is set. An affordable GPU-based PC cluster, built out of widely used components, is evaluated. The variety of applications covers the most important areas of visualization, and the presented results provide an up-to-date performance benchmark for such a typical installation. This study can be used, for example, to estimate the potential performance benefits of porting of a particular rendering application to run in a parallel environment, in order to understand beforehand if it is feasible at all and what exactly can be achieved in the best or worst case scenarios.

- *Image data compression*

  The performance of the parallel rendering framework itself largely depends on the compositing stage and essentially on the compression used during image transmission. In the following several image compression schemes are discussed and evaluated within the fully functioning rendering setup. RLE-based as well as YUV image compression methods are integrated into the framework in order to support sort-first and sort-last compositing, where YUV is implemented on GPU for better performance. Since performance of a particular compression method is strongly dependent on the rendering outcome, these methods are then studied in different rendering scenarios. and practical guidelines for their usage are concluded.

- *Compositing optimization*

  A new compositing performance optimization algorithm based on regions of interest for sort-last rendering is introduced. The method is independent of the rendering algorithm and the framework and exploits very basic assumptions about the rendering result. The main idea is to optimize the ”read-back, compression, transmission” part of the compositing by analyzing the frame buffer before the data is read from GPU to RAM. The algorithm then identifies changed frame buffer areas and splits initial GPU to RAM read-back requests into a series of smaller read-backs excluding empty regions. This method provides significant speedup for sort-last decomposition mode, comparing to other compression techniques, since in the best case it is able to reduce the RAM footprint significantly. The only assumption that is used by the proposed method is that with the increased numbed of renderers the amount of changed screen pixels usually decreases.

- *Out-of-core application parallelization*

  The terrain rendering application (RASTeR) was ported to Equalizer (eqRASTeR) in order to extend its functionality to multiple displays and scalable, many to one, rendering. RASTeR is a typical out-of-core rendering solution, which has its own data management layer; this had to be taken into account, as out of the box solutions for load balancing, provided by Equalizer, were failing in scalable setups. Various aspects of the out-of-core application were therefore studied and improved, new approaches for load balancing and data distribution are proposed in the context of eqRASTeR. Obtained results and methods can be directly translated and applied to various out-of-core applications especially for the sort-last rendering scenarios.

- *Data management and application development*

  In order to bridge the gap between applications' and rendering frameworks' developers a parallel out-of-core data management system is introduced to Equalizer. Since it is mapped to the class hierarchy provided by Equalizer, the seamless integration of out-of-core applications, supporting multiple GPUs, is now made easier, which is demonstrated by the volume rendering application for large data sets, that illustrates most of the data management functionality. The management system basically provides a two level cache, where data can be asynchronously uploaded to the RAM by multiple threads, and then can be accessed by multiple GPU threads, which asynchronously load data from RAM to GPUs memory. Custom applications can extend the framework to different types of data and rendering techniques, keeping core asynchronous behavior intact.

1.5 Dissertation Overview

In the following, a short overview on the contents of the next chapters is given.

In Chapter 2, the basics of scalable interactive visualization are briefly described. The first part explains methods like *data compression*, *frustum/object culling*, *level of detail*, *out-of-core rendering* etc. that are applied not only in a parallel rendering setup, but also within a single rendering unit. The second part of Chapter 2 gives an overview on parallel rendering systems and presents the motivation behind the choice of the Equalizer framework, explaining further the structure of parallel visualization systems in general.

Chapter 3 is about parallel image compositing and image data compression, which are the most essential parts of the parallel visualization framework. Sort-first, sort-middle and sort-last rendering decompositions are described; compositing in case of polygonal and volume rendering is introduced, as well as few parallel image compositing algorithms are explained. Since several compression methods will be evaluated later, they are also presented in Chapter 3.

Once the basics are covered, the performance of the Equalizer framework is evaluated in Chapter 4, where a polygonal rendering application with static data distribution is used. Static data distribution helps to separate application's and the framework's performances better. Different compression methods are evaluated within the fully functioning parallel visualization system on a 10-node cluster. Additionally, the performance between distributed (Equalizer) and streaming (Chromium) systems is compared; also the performance of an out-of-core terrain rendering application, ported to Equalizer, is demonstrated, as well as two approaches of automatic and static load balancing are discussed.

The compositing optimization algorithm is then presented and evaluated in Chapter 5, in a similar setup as used in Chapter 4. A detailed study of several rendering scenarios using this algorithm is provided as well.

Chapter 6 describes the data management system that was developed in order to support application developers. This system, coupled with Equalizer, makes creating of parallel out-of-core rendering applications easier. A description of such a visualization application for rendering of large volume data is also presented in Chapter 6, revealing further insights into building a parallel rendering applications with the Equalizer framework.

Chapter 7 concludes this thesis with a summary and gives directions for future work.

CHAPTER 2

SCALABLE INTERACTIVE VISUALIZATION

2.1 Generic Rendering Approaches

Interactive data visualization is important in many fields. It is vital to perform visualization in real-time or at interactive frame rates for efficient data analysis. While the performance of GPUs grow fast it is not able to keep up with the data grow. Throughput, storage as well as RAM and VRAM size limitations promoted development of various data reduction techniques to enable large-scale visualization on a single machine. Methods like data compression, frustum/object culling, Level Of Detail (LOD), multi-resolution, out-of-core and variable viewport rendering greatly improve performance, but there are still limits to their use.

Data compression The very first issue of large data is to deliver it to RAM and VRAM from computer's hard disk drive (HDD) or remote network storage. Since this initial access is the slowest link of a chain, the evident solution is to store data in compressed form. Less data then has to be read from a slow device while some computational power is used to decompress it every time the data is being fetched. On the other hand, compression of the data prior to visualization is an inevitable preprocessing overhead that often can be ignored since it has to be done only once.

Data compression is an art of its own; particular method greatly depends on

the information's characteristics and available resources. It can be a general compression method suitable for any type of information (like LZO[1] library), some area specific algorithm that explores properties of a certain data type coherencies (i.e. wavelet transform [Fout and Ma, 2007], discrete cosine transform [Yeo and Liu, 1995] (DCT), Fourier transform [Chiueh et al., 1997], tensor [Kolda and Bader, 2009] or vector quantization-based [Schneider and Westermann, 2003] compression of volumetric data, or various mesh compression schemes [Courbet and Hudelot, 2009; Gobbetti et al., 2006] for polygonal data) or combination of both.

Frustum/object culling Exploring of a large data typically includes not only observing the entire data at all times, but also taking a closer looks at small portions of it, studying different pieces of the information in detail, therefore it is important to limit the amount of data being loaded and displayed. For example, in geo-visualization (or terrain rendering) the particular area of interest is a city or a street rather than the whole earth itself, thus it is vital to load and display only a small visible or interesting portion of, probably, enormous dataset.

There are three types of methods used in practice to determine visible set of primitives: view-frustum, back-face and occlusion culling. *View-frustum culling* excludes information not visible to the virtual camera due to camera's own setup (in computer graphics visible area is typically represented as a frustum of a rectangular pyramid and consists of six clipping planes). Every portion of information is tested separately for visibility only against the frustum itself. *Back-face culling* is responsible for discarding surfaces of opaque objects that are facing away from the camera. Finally, *occlusion culling* is used to omit objects, which are occluded by other objects, it could be done in different ways [Klosowski and Silva, 2000; Koltun et al., 2000], while the occlusion queries [Bittner et al., 2004] is the most popular method, since it has hardware support in modern GPUs. Visual representation of different visibility culling methods is given on Figure 2.1.

Level Of Detail and multi-resolution rendering When dealing with perspective projection, further away objects appear smaller on the screen, this means they don't require the same level of refinement as objects closer to the camera, since these details will not be visible and thus it would be a waste of space and rendering resources. Having multiple representations of the same object with different level of detail and selecting proper version depending on the occupied screen-space is a good way to improve visualization performance, and reduce memory footprint. The major problem with discrete LOD techniques is in seamless switching between levels. If not done carefully switching between

[1] http://www.oberhumer.com/opensource/lzo/

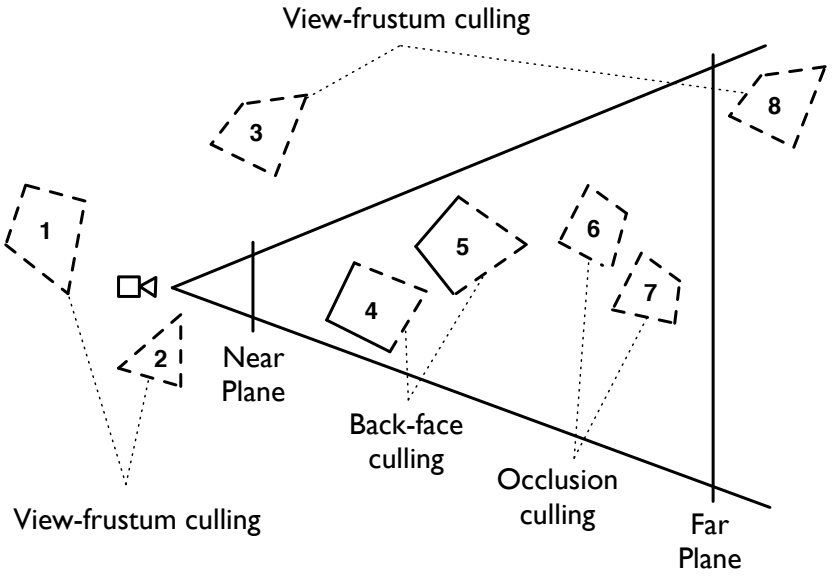


Figure 2.1: *Different culling types: objects 1, 2, 3, 8 are discarded by* view-frustum culling *since they fall out of view-frustum defined by near, far, and four side clipping planes; facing away from the camera surfaces of opaque objects 4 and 5 are rejected by* back-face culling*; objects 6 and 7 are completely obscured from the camera point of view by objects 4 and 5, thus they are ignored due to* occlusion culling.

discrete levels produces visible ”popping” effects; depending on the application these effects could be tolerated or will be considered as artifacts.

Some compression methods allow automatic discrete LOD exploration. With these methods data can be gradually decompressed until desired precision is reached (typical examples are wavelets [Fout and Ma, 2007], or DCT [Yeo and Liu, 1995] compression methods). Continues LOD methods [Hoppe, 1997; Strugar, 2009] can be applied in some scenarios, where seamless LOD change is possible even within one model of the scene.

Deriving from the LOD techniques, *multi-resolution rendering* is used to balance quality-performance ration by selecting lower resolution data representation whenever performance is more important than quality.

Out-of-core rendering Short reaction time and sufficient frame rate are important aspects of any interactive visualization. *Out-of-core* is referred to rendering of the data that doesn't fit to the main memory entirely, thus exploring data usually requires frequent loading of new portions to RAM and VRAM, delays are eventually introduced if this process is interleaved with actual rendering, exposing lower frame rates and negative user experience. Out-of-core rendering is usually combined with special data management strategies to hide this latency by load-

ing requested data in parallel with rendering of already available representation, possibly of lower quality. Lower quality representation occupies less space, thus faster to load and visualize; it gives the user a general idea of what is going to be seen next and allows some level of data exploration, while higher resolution version is being prepared in the background.

Variable viewport size If rasterization is the limiting factor of interactive visualization, another approach to improve performance is to reduce rendered resolution of the screen itself. The lower resolution image is created in the invisible buffer and then upscaled to fill desired screen space. Final image will look blurry on the screen as a result of such quality-performance trade-off.

All the aforementioned methods are tightly coupled in a modern visualization system, supporting and completing one another. Clever data compression can provide LOD, while LOD is a building block for out-of-core visualization, multiresolution rendering and data management; variable viewport size has direct influence on what LOD data is selected, where frustum/object culling can be generally applied to any rendering method.

Despite all the effort to perform visualization interactively on a single machine there are two major cases when it doesn't suffice: first, large multi-megapixel displays, display arrays, or virtual environments, where single GPU hits the rasterization limit, or physically not capable to support that large number of displays; and second, too large data to pre-process on a single machine in appropriate time (this includes scenarios where large amounts of data has to be visualized without preprocessing in order to, for example, steer ongoing simulation process).

Apparent solution is to use a system consisting of multiple GPUs and multiple computers to split visualization task. Desired features of such setups are: scalability, simplicity of configuration, easy to port of existing applications, as well as moderate cost of building and maintaining.

2.2 Parallel Rendering Systems

The early fundamental concepts of parallel rendering have been laid down in [Molnar et al., 1994] and [Crockett, 1997]. A number of domain specific parallel rendering algorithms and special-purpose hardware solutions have been proposed in the past, however, only few generic parallel rendering frameworks have been developed.

2.2.1 Domain Specific Solutions

Cluster-based parallel rendering has been commercialized for off-line rendering (i.e. distributed ray tracing) for computer generated animated movies or special effects, since the ray tracing technique is inherently amenable to parallelization for off-line processing. Other special-purpose solutions exist for parallel rendering in specific application domains such as volume rendering [Li et al., 1997; Wittenbrink, 1998; Garcia and Shen, 2002; Nie et al., 2005] or geo-visualization [Vezina and Robertson, 1991; Li et al., 1996; Johnson et al., 2006]. However, such specific solutions are typically not applicable as a generic parallel rendering paradigm and do not translate to arbitrary scientific visualization and distributed graphics problems.

Recently in [Niski and Cohen, 2007], parallel rendering of hierarchical level of detail (LOD) data has been addressed and a solution specific to sort-first tile-based parallel rendering has been presented. While the presented approach is not a generic parallel rendering system, basic concepts, such as load management and adaptive LOD data traversal, can be carried over to other sort-first parallel rendering solutions.

2.2.2 Special-Purpose Architectures

Traditionally, high-performance real-time rendering systems have relied on integrated proprietary system architecture, such as the SGI graphics super computers. These special-purpose solutions have become a niche product as their graphics performance does not keep up with off-the-shelf workstation graphics hardware and scalability of clusters. However, cluster systems need more sophisticated parallel graphics rendering libraries.

Due to its conceptual simplicity, a number of special-purpose image compositing hardware solutions for sort-last parallel rendering have been developed. The proposed hardware architectures include Sepia [Moll et al., 1999], Sepia 2 [Lombeyda et al., 2001], Lightning 2 [Stoll et al., 2001], Metabuffer [Blanke et al., 2000; Zhang et al., 2001], MPC Compositor [Muraki et al., 2001] and PixelFlow [Molnar et al., 1992; Eyles et al., 1997], of which only a few have reached the commercial product stage (i.e. Sepia 2 and MPC Compositor). However, the inherent inflexibility and setup overhead have limited their distribution and application support. Moreover, with the recent advances in the speed of CPU-GPU interfaces, such as PCI Express and other modern interconnects, combinations of software and GPU-based solutions offer more flexibility at comparable performance.

2.2.3 Generic Systems

A number of algorithms and systems for parallel rendering have been developed in the past. On one hand, some general concepts applicable to cluster parallel rendering have been presented in [Mueller, 1995; Mueller, 1997] (sort-first architecture), [Samanta et al., 1999; Samanta et al., 2000] (load balancing), [Samanta et al., 2001] (data replication), or [Cavin et al., 2005; Cavin and Mion, 2006] (scalability). On the other hand, specific algorithms have been developed for cluster based rendering and compositing such as [Ahrens and Painter, 1998], [Correa et al., 2002] and [Yang et al., 2001; Stompel et al., 2003]. However, these approaches do not constitute APIs and libraries that can readily be integrated into existing visualization applications, although the issue of the design of a parallel graphics interface has been addressed in [Igehy et al., 1998]. Only few generic APIs and (cluster-) parallel rendering systems exist which include VR Juggler [Bierbaum et al., 2001] (and its derivatives), Chromium [Humphreys et al., 2002] (an evolution of [Humphreys and Hanrahan, 1999; Humphreys et al., 2000; Humphreys et al., 2001]), Multipipe SDK [Bhaniramka et al., 2005; Jones et al., 2004] and Equalizer [Eilemann et al., 2009].

OpenGL Multipipe SDK OpenGL Multipipe SDK (MPK) [Bhaniramka et al., 2005] implements an effective parallel rendering API for a shared memory multi-CPU/GPU system. It is similar to IRIS Performer [Rohlf and Helman, 1994] in that it handles multi-pipe rendering by a lean abstraction layer via a conceptual callback mechanism, and that it runs different application tasks in parallel. However, MPK is not designed nor meant for rendering nodes separated by a network. MPK focuses on providing a parallel rendering framework for a single application, parts of which are run in parallel on multiple rendering channels, such as the culling, rendering and final image compositing processes.

VR Juggler VR Juggler [Just et al., 1998; Bierbaum et al., 2001] is a graphics framework for virtual reality applications, which shields the application developer from the underlying hardware architecture, devices and operating system. Its main aim is to make virtual reality configurations easy to set up and use without the need to know details about the devices and hardware configuration, but not specifically to provide scalable parallel rendering. Extensions of VR Juggler, such as for example ClusterJuggler [Bierbaum and Cruz-Neira, 2003] and NetJuggler [Allard et al., 2002], are typically based on the replication of application and data on each cluster node and basically take care of synchronization issues, but fail to provide a flexible and powerful configuration mechanism that efficiently supports scalable rendering as also noted in [Staadt et al., 2003].

Chromium While Chromium [Humphreys et al., 2002] provides a powerful and transparent abstraction of the OpenGL API, that allows a flexible configuration of display resources, its main limitation with respect to scalable rendering is that it is focused on streaming OpenGL commands through a network of nodes, often initiated from a single source. This has also been observed in [Staadt et al., 2003]. The problem comes in when the OpenGL stream is large in size, due to not only containing OpenGL calls but also the rendered data such as geometry and image data. Only if the geometry and textures are mostly static and can be kept in GPU memory on the graphics card, no significant bottleneck can be expected as then the OpenGL stream is composed of a relatively small number of rendering instructions. However, as it is typical in real-world visualization applications, display and object settings are interactively manipulated, data and parameters may change dynamically, and large data sets do not fit statically in GPU memory but are often dynamically loaded from out-of-core and/or multiresolution data structures. This can lead to frequent updates not only of commands and parameters which have to be distributed but also of the rendered data itself (geometry and texture), thus causing the OpenGL stream to expand dramatically. Furthermore, this stream of function calls and data must be packaged and broadcast in real-time over the network to multiple nodes for each rendered frame. This makes CPU performance and network bandwidth a more likely limiting factor.

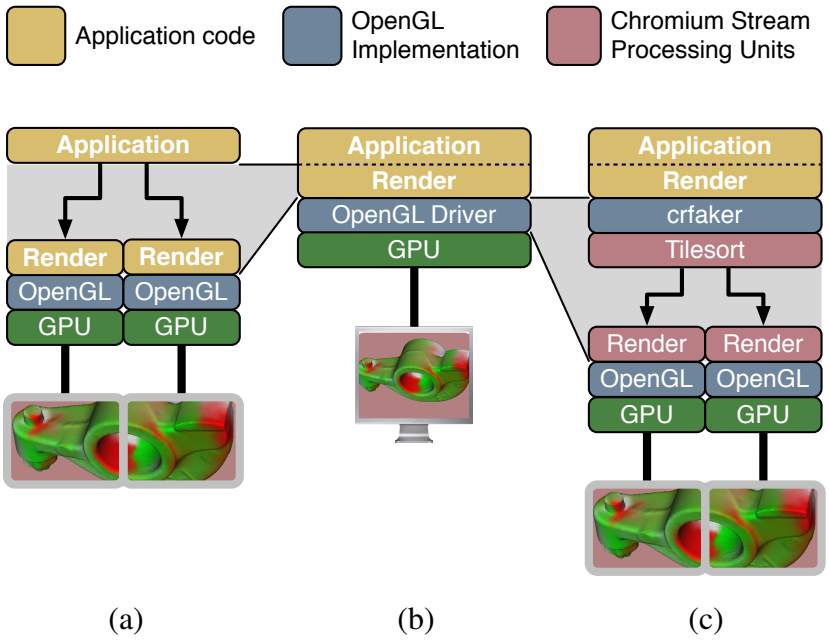


Figure 2.2: *A traditional OpenGL application (b) and its equivalents when using CGLX or Equalizer (a) and Chromium (c).*

The performance experiments in [Humphreys et al., 2002] indicate that Chromium is working quite well when the rendering problem is fill-rate limited. This is due to the fact that the OpenGL commands and a non-critical amount of rendering data can be distributed to multiple nodes without significant problems and since the critical fill-rate work is then performed locally on the graphics hardware.

Chromium also provides some facilities for parallel application development, namely a sort-last, Binary-Swap compositing SPU and an OpenGL extension providing synchronization primitives, such as a barrier and semaphore. It leaves other problems, such as configuration, task decomposition as well as process and thread management unaddressed. Parallel Chromium applications tend to be written for one specific parallel rendering use case, such as for example the sort-first distributed memory volume renderer [Bethel et al., 2003] or the sort-last parallel volume renderer Raptor [2].

CGLX This framework [Doerr and Kuester, 2011] is designed to run applications on display walls. It is trying to overcome performance limitations of Chromium by executing rendering tasks in a distributed way. Each node, that has displays connected to it, is running separate application to render images locally, avoiding transmitting of OpenGL stream of image data over the network. For example, one can setup a multi-screen display-wall with Chromium, streaming the OpenGL calls to a number of render nodes assigned to screen tiles of the display-wall, as illustrated in Figure 2.2(c), where only single instance of the application is running. In contrast, CGLX runs parts of the application in parallel on multiple rendering channels as illustrated in Figure 2.2(a). Only frustum, transformation matrixes and user interaction are communicated over the network for synchronous rendering.

The framework explores master-slave paradigm, where all events are sent to a master node, which then emits correct commands for renderers. A separate daemon process has to be running on each machine that is responsible for handling connections, rendering application startup and on the fly reconfiguration. It is possible to run multiple applications in parallel using different displays and different ports for communication.

Compared to Chromium the reported performance is much higher in geometry heavy scenarios, as a downside, applications have to be ported to the framework, where Chromium is able to run unmodified binaries. The porting effort, however, is minimal, since CGLX doesn't handle any compositing operations or sort-last rendering. CGLX package additionally provides convenient visual tools for setting up a large display wall, which also includes simulation mode for easier testing on a single machine.

[2] http://graphics.stanford.edu/projects/raptor/

Equalizer Parallel Rendering Framework A major strength of Equalizer is its flexible and scalable configuration of the parallel rendering tasks, which takes the notion of a compound tree introduced in MPK [Bhaniramka et al., 2005] to a distributed cluster environment. Hence different parallel rendering task decomposition and image compositing configurations can easily be specified. For example, efficient Direct Send sort-last image compositing has been demonstrated in [Eilemann and Pajarola, 2007].

One of the essential differences of Equalizer to Chromium is that it is fully distributed and runs the application code in parallel. While preserving a minimally invasive API, Equalizer system is better aimed at scalability as the actual data access is decentralized in the distributed rendering clients. Equalizer provides very flexible task decomposition configuration, and while Chromium's infrastructure is primarily the compositing stage, applications written once for Equalizer can easily be run in any different task decomposition mode and for any physical display configuration without any changes to the application itself.

Equalizer takes care of distributed execution, synchronization and final image compositing, while the application programmer identifies and encapsulates critical parts of the application, such as culling and rendering. This approach is considered to be *minimally invasive* since the existing and proprietary rendering code can basically be retained. All rendering is executed directly to an OpenGL context, and at no point are OpenGL commands sent over the network.

This minimally invasive approach allows the application to retain its OpenGL rendering code, but structures the implementation to allow for optimal performance. The network bandwidth is freed from unnecessary transmission of excessive graphics commands and data since only the basic rendering parameters are exchanged between nodes. Only for the unavoidable final image compositing step in scalable rendering, frame buffer data between the nodes must be exchanged. The application can implement efficient dynamic database updates based on distributed objects or message passing as these distributed systems primitives are provided by Equalizer.

The Equalizer framework does not impose any constraints on how the application handles and accesses the data to be visualized. As such, Equalizer does not provide a solution to the parallel data access and distribution problem, which has to be addressed by the application itself, as it is an orthogonal issue. It does address some fundamental problems to help application developers to distribute their data effectively in the context of parallel rendering. The Equalizer networking layer supports message passing and the creation of distributed objects. By sub-classing a distributed object class, static and versioned objects can be created. Objects are addressed on the cluster using a unique identifier, which allows the remote mapping of the object. Versioned objects are typically used for frame-specific data, where a new version for each new frame is created. Equalizer passes

this version information correctly to the application's rendering code. This mechanism allows simple distribution and multi-buffering of data.

Compared to MPK, Equalizer supports fully distributed parallel rendering paradigm and features more flexible task decomposition approach. CGLX system was build with similar principles to Equalizer, while specializing on display wall configuration and having no support for other use-case scenarios. It is different from VR Juggler in that it fully supports scalable parallel rendering such as sort-first and sort-last task decomposition and image compositing, it provides more flexible node configurations which for example allow specifying arbitrary task decomposition and image compositing combinations as simple compound layouts. Furthermore, it is fully distributed which includes support for network swap barriers (synchronization), distributed objects as well as image compression and transmission, also it supports multiple rendering threads per process, which is important for multi-GPU systems.

Due to advantages described above, Equalizer was chosen as a main tool of this thesis for detailed investigation and further enhancements of parallel rendering applications.

2.3 Structure of Parallel Visualization System

A general scheme of a distributed rendering system is presented on Figure 2.3. The essential components are: *Data source*, *Rendering nodes*, *Visualization nodes*, and a *Server node*. It is important to note that any of the components can be combined in a single device within a particular rendering system; that is, for example, the whole system can be represented by a single PC with multiple GPU cards.

Data source Prior to visualization, data has to be simulated or acquired by other means (e.g., scanning). The *Data source* in the Figure 2.3 is therefore related to a file, database, Network Attached Storage (NAS), supercomputer, PC cluster or any other device that can store or produce data for visualization.

The data flow from/to the *Data source* and from/to *Rendering nodes* means that rendering nodes themselves can produce some data (for example acceleration structures for more efficient rendering) or they can exchange portion of the data between each other without involving the original *Data source* (due to caching or internal data management strategy it could be more efficient to transfer already available portions of data from one rendering node to another directly, rather then loading this data from the original source as shown by [Castanie et al., 2006]).

Rendering nodes The data has to be rendered first in order to be visualized. In general, the rendering device doesn't have to be able to actually display

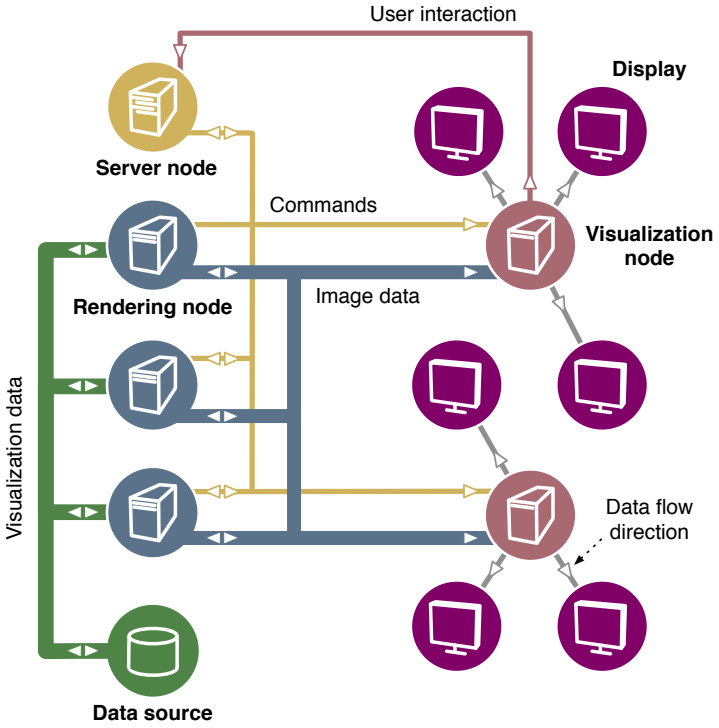


Figure 2.3: *A general scheme of a distributed parallel visualization system. Various components are presented as pictograms; the data flow and type is illustrated with lines; finally, data flow directions are marked with arrows.*

anything. In case of CPU-based rendering on a supercomputer there might be no displays connected to the rendering system at all, it would be forced to use another device to display produced images, or save them to a storage device, from where they would be read by *Visualization nodes*.

Often, *Rendering nodes* have to exchange *Image data* between each other to obtain final pixel values, this happens when multiple machines are rendering the same pixels in parallel (this stage is called *Compositing* and is explained in detail later in the Section 3.1), hence the bidirectional arrows to/from the *Rendering nodes* on the *Image data* flow.

As it was explained earlier the *Rendering node* is combined with the data source if it produces its own data to render (CPU-based rendering on a supercomputer), or when the data is initially copied to the rendering nodes itself (e.g., when distributed file system is used on a rendering cluster to store the data on the same devices, or the data is manually pre-divided between nodes).

Visualization nodes A computer that has connection to a display is called *Visualization node* in Figure 2.3. Generally, such device would be used to display

obtained images and is not producing any renderings itself (remote visualization is one example, where low cost PC is used to display images that are produced by a remote rendering service), therefore the *Image data* flow has only an incoming direction.

The *Visualization node* itself is a device for user interaction in the presented drawings. It is used to send user input commands back to the system to steer visualization. There could be a separate device for capturing user input, but such case is omitted here for simplicity.

In a common setup some of the *Rendering nodes* are combined with the *Visualization node*, such installation is most often used with local rendering on PC clusters. In this case displays are directly connected to some of the rendering machines, therefore the rendering and visualization roles are combined. The advantage is in the reduced *Image data* overhead, since the *Image data* doesn't have to be sent elsewhere for final displaying.

Server node In order to schedule visualization tasks and synchronize rendering a centralized *Server node* is often used. In order to schedule visualization process, user input, received from a *Visualization node*, is considered, additionally, the information such as rendering timings and loaded data state is collected from the *Rendering nodes*. Based on the available input *Server node* is deciding for every frame what exactly is going to be rendered and what *Image data* is going to be exchanged. *Server node* can be as well responsible for starting, initial configuration and shut down of a visualization system. In a typical setup it is often combined with one of the *Rendering nodes*, being represented by a separate process.

C H A P T E R 3

IMAGE COMPOSITING AND IMAGE DATA COMPRESSION

3.1 Primitives Sorting Strategies

Rendering on a modern hardware could be roughly divided into two stages: geometry processing (transformation, clipping, lighting, etc.), and rasterization (scan-conversion, shading and visibility determination). In the *Fully Parallel* scenario (Figure 3.1 (a)) all geometry is processed in parallel, followed by processing of fragments in parallel. Geometry processing is parallelized through assigning subsets of primitives to different processors, while during rasterization step assignment is done through separation of various pixel groups. Essentially, the result of rendering is a combined effect of all primitives on all pixels. Since primitives can fall anywhere on the screen the task of rendering is viewed as a primitive to screen sorting problem. There are three places where sorting can take place, depending on hardware and software setup, according to [Molnar et al., 1994]. Figure 3.1 illustrate those options: *Sort-First*, where primitives are sorted (distributed) early in the rendering pipeline - during or before geometry processing, when their screen-space parameters are not yet known; *Sort-Last*, where sorting is done after of during rasterization (usually meaning that values of screen pixels are already known); *Sort-Middle*, when sorting happens between geometry processing and rasterization (redistributing screen-space primitives).

Only *Sort-First*, *Sort-Last* and combinations of two are used in modern parallel

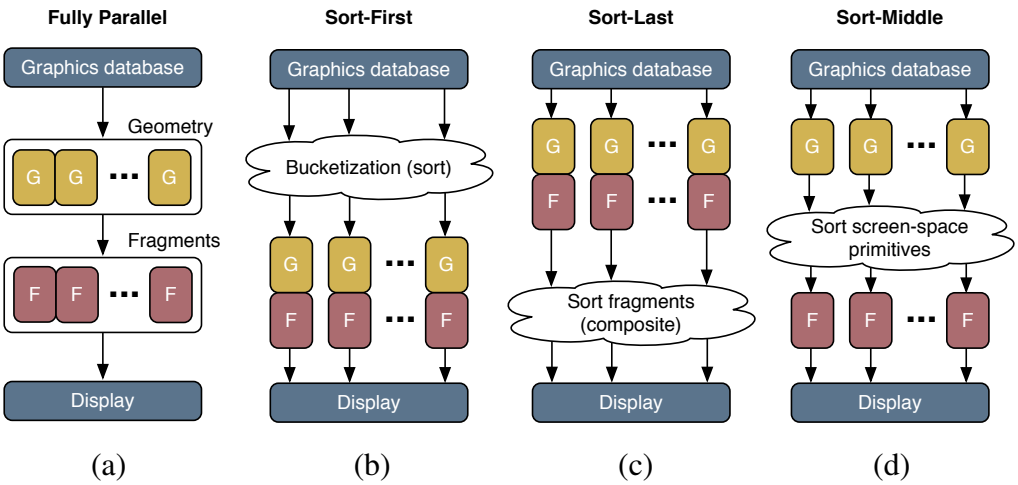


Figure 3.1: *Different sorting strategies for rendering parallelization. Fully parallel geometry and fragment processing (a); popular screen-space (Sort-First) and data-space (Sort-Last) distribution schemes (b) and (c); rarely mentioned with respect to modern hardware, screen-space primitives redistribution (d).*

rendering setups due to their conceptual simplicity and natural support in modern hardware.

Sort-first As originally suggested by [Molnar et al., 1994], the screen is split into regions and each region is assigned to different renderer; each primitive of the scene is then transformed only to the extent when it is possible to determine in which screen region it will fall. Once a primitive is assigned to its region, it is transferred to the appropriate renderer for further evaluation. After the rendering of all primitives for a certain screen region is done, the color values of rendered pixels are sent to display. Figure 3.2 illustrates this process: four renderers (on the left) are evaluating four parts of the screen independently, when finished, this parts are copied to the display (on the right) to expose a complete frame.

For efficiency, a hierarchical structure is often employed and screen region evaluation is done through bounding volumes of groups of primitives, using this hierarchy, therefore no actual geometry processing over separate primitives is needed. This approach lead, however, to data duplication, since same bounding volumes will fall into different screen regions, and not only separate primitives would have to be duplicated but groups of primitives; by adjusting size of the group it is possible to find a balance between screen tile classification overhead and data duplication.

If a certain renderer didn't yet have the primitives it supposed to render, it

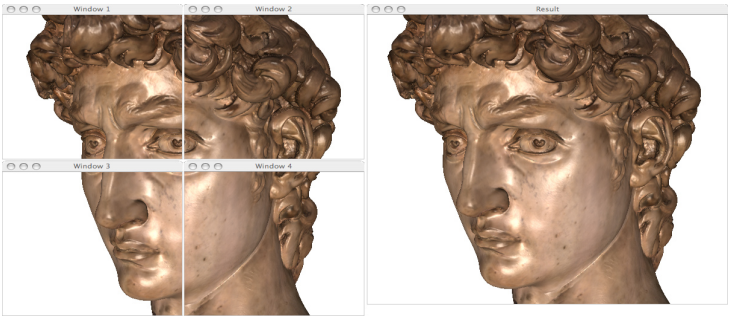


Figure 3.2: *Tiled sort-first parallel rendering using four channels, and showing the final assembled image on the right.*

has to load them from an external source causing data redistribution, possibly on every frame. Since the camera position and models orientations within a scene are usually change over time significantly, the problem of redistributing primitives can become a bottleneck, unless the data is small enough and fits to every renderer's memory entirely.

Another issue is in optimal and consistent screen-to-regions split estimation. For efficient load balancing the amount of information being processed by each renderer on every frame should be as similar as possible, which could be difficult to achieve in *Sort First*, using only rough estimates as it is usually done to avoid performance penalty of a full geometry processing. On the other hand, the screen regions consistency is necessary to reduce significant per-frame data redistribution.

On the bright side, the compositing in *Sort First* is trivial. It requires minimal amount of information (e.g., only RGB values of pixels of a single screen) to be transferred to the destination display; furthermore the amount of image data is a constant and only depends on the screen size and, unlike with other strategies, it is independent of the number of renderers or the nature of the rendering itself.

When it comes to porting of existing applications to perform rendering in parallel, *Sort-First* is very straightforward. Since rendering could be done, within each screen region in the exactly same way as it is done by non-parallel applications, it is possible to distribute applications having only binary executables, without changing a single line of the original code or any code recompiling at all, as it is done by Chromium framework [Humphreys et al., 2002].

Sort-last When primitives are distributed between renderers beforehand, without considering their screen space position, each renderer in the simplest case will perform evaluation of fragment information for the whole screen. When each renderer has finished the evaluation, different devices will have pixel values for similar positions on the screen, therefore screen pixel values have to be commu-

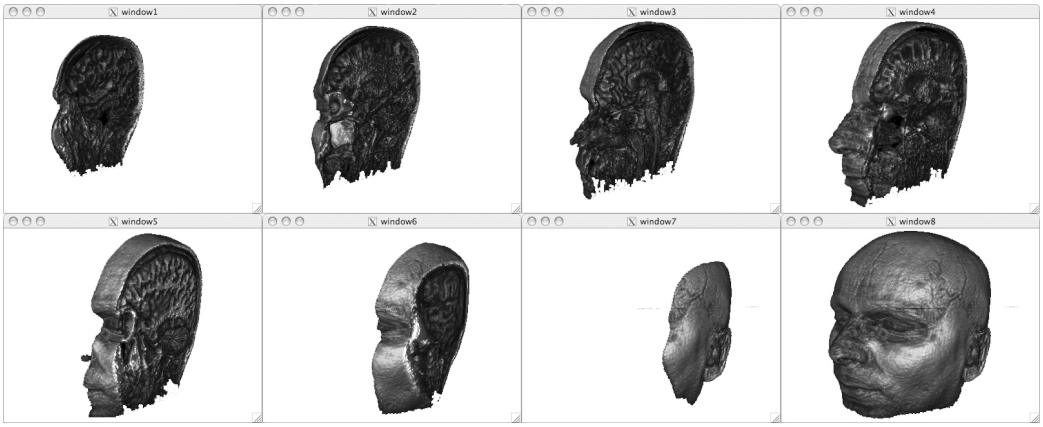


Figure 3.3: *Sort-last parallel rendering of a large volume data set divided uniformly into slabs. Lower-right window shows final destination channel with back-to-front α-blended slab images.*

nicated between renderers in order to, based on pixels' visibility, establish final values for the display.

Compared to sort-first, more data is potentially rendered, and more rasterization is done by fragment processors, since it is usually impossible for one renderer to determine if its pixel data will be occluded by pixel values produced in parallel by some other renderer during compositing stage, while in case of sort-first fully occluded data could be determined early.

Compositing overhead itself is dependent on the number of renderers, and, while it is possible to equalize amount of data being communicated within each node, the total number of messages and the image data size have linear dependency from the number of rendering resources. Additionally, actual compositing have to be done through pixel depth and/or transparency comparisons with possible blending of values, compared to simple pixels values copy of sort-first, thus even more data have to be sent over network (z-value for depth compositing, opacity values for blending or both).

Different portions of the data are assigned to different rendering resources and this information is typically known before each new frame starts. Data subdivision can be static, which will avoid sending additional information over the network, but can lead to load imbalance, or dynamic, which requires more complicated data management scheme. In order to balance the load, system would have to redistribute portions of rendering primitives, or have certain data redundancy, as shown by [Samanta et al., 2001].

Figure 3.3 illustrates process of *Sort-Last* rendering and final image compositing: seven windows are featuring different parts of the volumetric model produced by seven rendering resources; lower-right image shows final composited view that

is used for displaying. This simple example shows how mush data can be unnecessary rendered, however if the volume would be semitransparent, rendering resources would not be wasted.

In order to create a parallel rendering application that allows *Sort-Last* rendering, significantly larger effort have to be invested, comparing to sort-first, not only regarding correct portions of data redistribution, but also with respect to compositing of the final images. Transparency effects have to be treated with great care, and many algorithms that work in screen-space domain would fail or have to be significantly adjusted (screen-space anti-aliasing is one of the examples).

Sort-middle Since geometry transformation of primitives can be done in parallel as well as rasterization, the natural moment to sort and redistribute work would seem to occur between geometry and rasterization stages. Once screen coordinates of primitives are determined and per-vertex information is evaluated by geometry processors, screen-space primitives could be reassigned to corresponding screen regions and redistributed among rasterization processors. Rasterization units could use information about primitives to determine visibility early and perform very little work on rasterization, while the amount data required to be redistributed on each frame is potentially large.

With respect to hardware, there are significant difficulties in implementing *Sort-Middle* approach for parallel rendering on distributed systems. It can be only implemented if intercepting of projected geometry before rasterization is supported and on-the-fly reconfiguration of rasterization units is available for load-balancing purposes. Due to tight coupling of geometry and rasterization units in modern high performance visualization hardware, there was minimal effort in building *Sort-Middle* cluster-based rendering solutions and very limited area of their application.

Current work is focused on GPU-based distributed rendering solutions, therefore *Sort-Middle* is not is the scope of interest and is not evaluated in the following text.

3.2 Image Compositing in Polygonal and Volume Rendering

In sort-first screen is divided into non-overlapping tiles, thus only color information has to be sent for final display, which is simply copied to the desired location. There is no compositing as such and therefore it doesn't depend on the rendering method itself. For sort-last, however, the difference for various applications is significant.

In the current document only polygonal and volume rendering are studied in detail. Polygonal rendering is applied for CAD (or 3D-laser scanned) and terrain types of models, featuring no transparency, where volumetric data rendering is essentially transparency-based. The conceptual difference between polygonal and volume rendering in sort-last is in the way the data is being split and the amount and type of the information required for final image compositing.

Polygonal rendering A standard approach to determine visibility of a pixel, when opaque geometry is being rendered, is through its depth value. Together with screen coordinates, third coordinate of a primitive, representing the distance (depth) of the primitive from the camera, is computed and stored. When primitives are rasterized, their screen coordinates and their depth values are interpolated, thus for every pixel of every primitive on the screen it is possible to compare corresponding depth values in order to figure out which one is closer to the viewer, if it happens that primitives overlap. In case of GPU rendering, this evaluation of visibility is extremely optimized, and there is very little to take care of. Once the rendering is done it is then possible to copy depth values from GPU to RAM (and vice versa) in the similar manner to how color values are copied.

When different parts of the same data in sort-last are rendered on different GPUs, most likely that same screen areas will be rendered by different GPUs more than once (otherwise it could be reduced to sort-first rendering decomposition), similarly to the single renderer, the depth information would be required to establish correct order and visibility of the pixels. For each pixel there are extra 24 to 32 bits of depth information, which has to be downloaded form the GPU, sent over the network along with color information, optionally uploaded to another GPU, and, finally, GPU or CPU depth-based composited.

The attractive property of polygonal rendering with opaque geometry is that primitives can be drawn in any order, creating multiple overlaps over different renderers, but in the end it will all be handled correctly through the depth-based compositing. For distributed rendering the developer has to take care of depth buffer (or z-buffer) precision, as it is normally done in case of a single-renderer (usually handled through setting correct near and far clipping planes), additionally, for correct compositing across multiple sources, same depth buffer settings should be used by all renderers. The latter property is easier to satisfy if a global z-buffer settings are evaluated for the whole data and then distributed among the renderers before every new frame is rendered. Figure 3.4 demonstrates sort-last decomposition of a polygonal model, where first four images feature different parts of the model rendered separately, and then composited to the final destination window on the right, which is itself was rendering fifth part of the data.

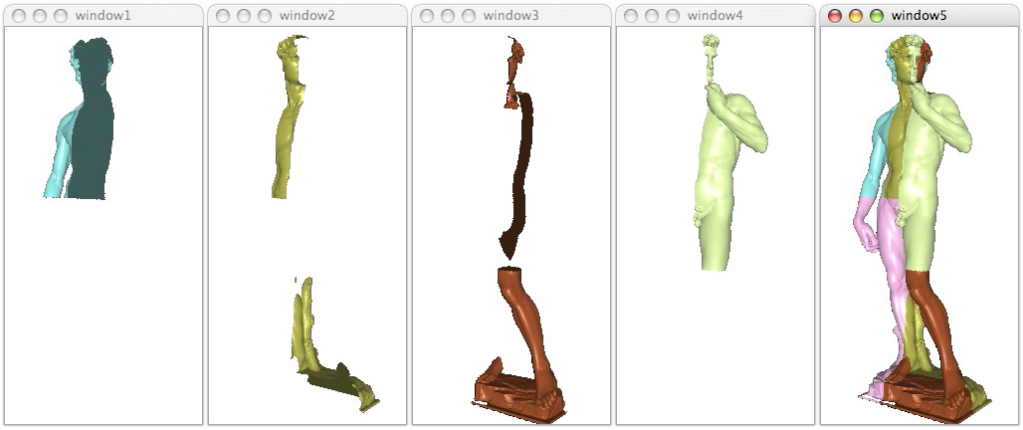


Figure 3.4: *Sort-last parallel rendering of a large polygonal model. Image on the right is a z-value compositing result of five contributing parts. Different parts of the data are color-coded for demonstration purpose only.*

Volume rendering Transparency-based rendering has to blend overlapping pixel values rather than discarding occluded information as it is done with opaque geometry, therefore the order, in which rendering and compositing is done, matters. In case of volume rendering the data is traversed in front to back or back to front order, accumulating opacity and color values. If the data is split between rendering resources, one has to make sure final compositing would be at all possible; this can be guaranteed, for example, by building a binary split tree, where volume iteratively divided on each iteration into two parts by an intersecting plane, therefore during bottom-up compositing only two children have to be merged.

In the simplest case, the volume can be divided into slabs along one dimension as illustrated in Figure 3.5. Each device renders one slab into a partial image, and final image assembly is performed by perspective-correct back-to-front α-compositing of the partial frame data, based on the relative positions of the slabs with respect to the viewer.

Figure 3.3 demonstrates scalable sort-last rendering using an eight-to-one node compound setup. In this example, final α-compositing of the rendered volume slabs is performed on the destination display.

As could be seen, compared to opaque polygonal rendering, compositing is more complicated. The order in which compositing is performed is important and even if the final display happens on one of the GPUs that was used for rendering, the image data on that GPU might be out of order and would have to be moved. When the order can be established per data unit, depth buffer doesn't require to be extracted, however, transparency values of pixels have to be read to the RAM and passed along with the color information anyway. The α-value of a pixel can

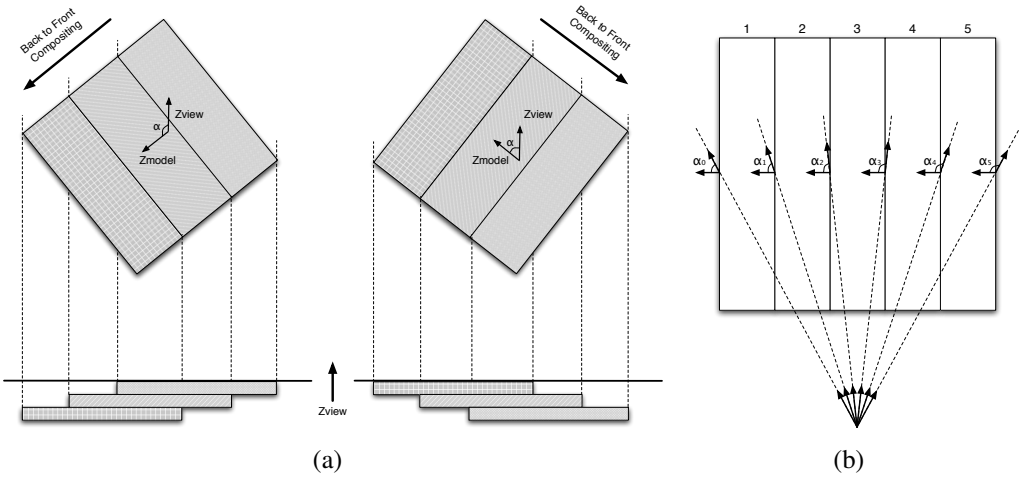


Figure 3.5: *Basic back-to-front compositing order of parallel volume slabs in case of (a) parallel projection and (b) perspective projection. Perspective compositing orders are 5-4-1-2-3 or 1-2-5-4-3, as well as, for this particular example: 1-5-2-4-3, 1-5-4-2-3, 5-1-2-4-3, and finally 5-1-4-2-3.*

occupy between 8 and 32 bits, but even with 8 bits decent results can be achieved, which makes α-compositing much faster than z-order compositing of polygonal rendering, despite more time consuming blending.

3.3 Advanced Compositing Methods

Overall performance of a parallel rendering system depends on two key aspects: rendering performance, and image compositing stage. Rendering performance is mostly influenced by the chosen rendering algorithm and efficiency of data fetching, where developer of a generic parallel rendering framework has very little influence on, and where most of the optimizations are left to the application's developer himself. To a certain extent, the efficiency of the compositing stage is influenced by the rendering output (as it will be shown in the Chapter 5, the lower amount of pixels on the screen was changed, the more efficient compositing can be), however, compositing performance can be studied and improved even beforehand for the case of a general renderer, assuming the worst rendering outcome (the entire available viewport is changed by every participating renderer).

Regardless of the rendering type it can be assumed that the entire viewport consist of c pixels, which occupy C bytes if only color information is considered. The number of machines participating in rendering will be denoted as N. Thus, in case of sort-first rendering, where final image compositing is performed by one of the renderers and rendering area is equally divided among the nodes, it will

be necessary to transmit $(N-1)(C/N)$, or $C(1-1/N)$ bytes, since different renderers change independent screen regions, and every pixel on the screen has to be transferred at most only once. With the growing number of machines N, the theoretical limit of transmission cost will approach C, which is a constant. Therefore assembling stage in case of sort-first doesn't impose any significant performance penalties and can be often implemented as serial $N-1$ to 1 color assembling for moderate N.

If the sort-last decomposition is chosen, every renderer can potentially change pixel data over the whole screen, plus additional depth or/and transparency information has to be communicated to insure correct compositing. If the amount of depth information per screen is D, the total cost of serial $N-1$ to 1 compositing, where one of the renderer is a destination, would result in $(N-1)(C+D)$ bytes. This linear dependency of the number of nodes leads to network saturation, and if this assembling strategy is used, compositing, as well as overall rendering, doesn't scale beyond couple of nodes (if at all). Therefore decentralized compositing methods are essential for efficient sort-last rendering.

Further, regardless of which compositing strategy is used, it is possible to reduce compositing time by applying various compression and screen-space optimization techniques, to ensure fast image/transparency/depth information delivery. Only general methods that do not depend on the rendering method or compositing strategy is of the interest in this work, since only those can be applied in the general parallel rendering framework.

A number of parallel compositing algorithms and their improvements have been proposed in the past. The major aim usually is in equalizing communication and the amount of computation between the nodes during compositing. The most popular generic methods are "Binary-Swap" (BS) [Ma et al., 1994] and "Direct Send" (DS) [Eilemann and Pajarola, 2007], of which only the DS will be evaluated in the further chapters.

3.3.1 Binary-Swap Compositing

Direct $N-1$ to 1 compositing can be considered as a one-stage method. When all of the nodes are done with rendering of a particular frame, they send partial image results, possibly at the same time, to the destination node for compositing (which eventually creates a network and compositing bottleneck). BS is solving this problem with a multistage distributed compositing approach, where nodes exchange partial image results among each other, and only during the final step partial viewports are sent to the destination channel in a sort-first manner.

The partial image exchange and intermediate image compositing is done in the following way (it is assumed here for simplicity that the data is split equally among the renderers and that each renderer updates the whole screen): after the

rendering is done, each of the nodes has rendered $1/N$ of the data that occupies the whole screen and consists of $C + D$ bytes; nodes are split into pairs and each pair exchange half of the color and data information on the screen, and performs compositing for the other half it receives, therefore each node sends $(C + D)/2$ bytes and after compositing each node has $2/N$ of composited data for its half of the screen; on the second step composted part of the screen is split again into two parts, and nodes exchange $(C + D)/4$ amount of information between newly defined partners, this process is repeated $log_2(N)$ times. After $log_2(N)$ iterations each node has fully composited tile of the screen of the size $(C + D)/N$, and up to this point each node would exchange and compose $(C + D)(1 - 1/N)$ bytes. Finally, each node would send its fully composited tile to the destination node in a sort-first manner, resulting in additional C/N bytes of transmission cost per node. In total $(C + D)(1 - 1/N) + C/N$ bytes are transmitted by additional nodes and $(2C + D)(1 - 1/N)$ bytes by the destination node.

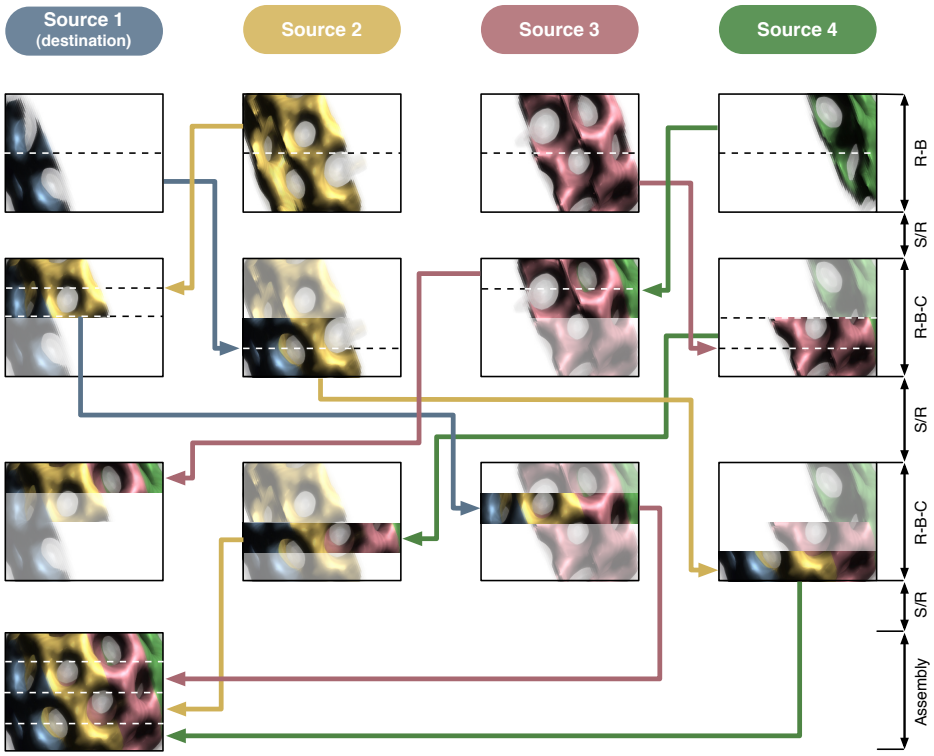


Figure 3.6: *Binary-swap sort-last compositing. The time line on the right is explaining different stages of the algorithm, such as: Read-Back (R-B); Send/Receive (S/R); Read-Back, Compositing (R-B-C), and, finally, sort-first-like Assembly. Colored arrows are showing the corresponding data transfer.*

With the increasing number of nodes, per-node communication and compositing cost approaches $C + D$ and $2C + D$ bytes for additional and destination nodes

accordingly, which are constants. The total number of bytes sent over the network $(C + D)(N - 1) + C$ is slightly larger than the one from direct $N - 1$ to 1 approach $(C + D)(N - 1)$, and the number of communication and synchronization points is increased significantly, however the benefit of a nearly constant per-node compositing cost is much more important.

Figure 3.6 demonstrates BS algorithm using four rendering sources, where *Source 1* is also a destination. The volume rendering example is presented here with the different source images shaded into different colors for visualization purposes; horizontal lines of frames correspond to different processing stages, where operations on different sources happen in parallel. After the initial rendering is done (first raw of images), first and second pair of nodes read-back, exchange and composite color and opacity information for corresponding halves of the screen. During the second iteration first and third sources (as well as second and fourth) read-back, exchange and composite quarters of the partially completed image; after this step $log_2(4) = 2$ iterations are finished and each source has fully composited $1/4$ of the full screen. Last stage is sending of these tiles to the final destination, which is *Source 1*.

3.3.2 Direct Send Compositing

There are a few disadvantages of the BS method: the original algorithm works only with the power of two sources; there are multiple stages of communication, which leads to additional synchronization points; and the compositing itself is somewhat overcomplicated for a larger number of nodes. The DS method was made in an effort to simplify compositing for moderate size visualization clusters, allowing at the same time for a non-power of two number of nodes.

Conceptually it is much simpler than BS. The idea of DS is to split entire viewport into the number of tiles equal to the number of nodes, after which each node is responsible for compositing of only one tile. There is only one stage during which every node sends parts of the screen to other nodes, at the same time receiving from every other node the same spatial tile of the screen for which it is responsible for. If N nodes participate in the exchange, and the screen is divided into N equally sized regions (tiles), each node would send $(C + D)/N$ bytes to every other node and receive $(C + D)(N - 1)/N$ bytes to assemble. After compositing each node would have completed $1/N$ of the screen which is sort-first assembled on to the destination channel, in the same way it is done by BS algorithm.

The amount of information being transmitted and composited by each node is equal to those of BS method. Having only one exchange stage it is much simpler to configure, the amount of the messages sent per compositing round per node however might impose issues when this type of compositing is applied in

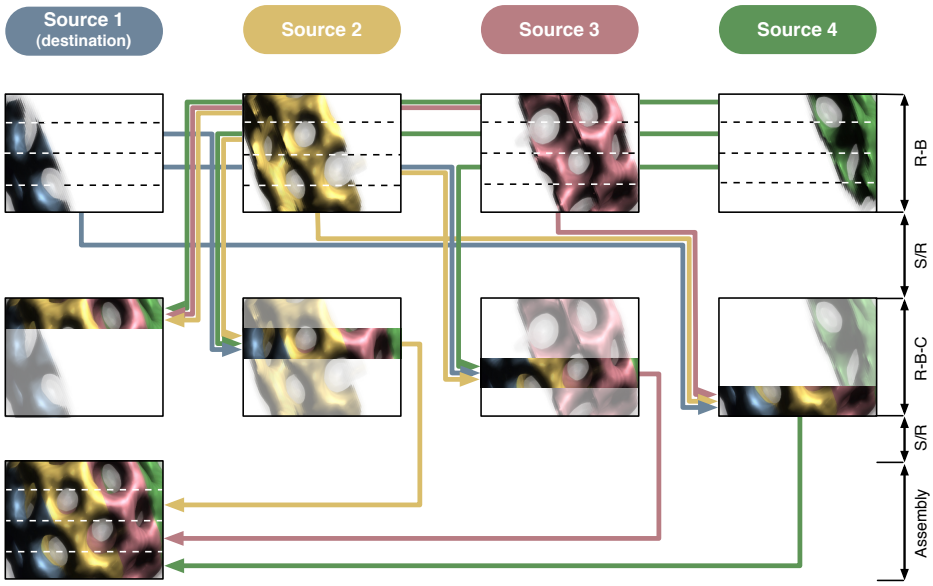


Figure 3.7: *Direct send sort-last compositing equivalent for the binary-swap example given in the Figure 3.6.*

a massively parallel setup with thousands of nodes, as was demonstrated by [Yu et al., 2008]. In the moderately sized setups (10-20 nodes), which are usually used for interactive visualization, this limitation doesn't produce any issues, since the number of rendering sources is relatively low.

Figure 3.7 illustrates DS compositing for the same setup used in the Figure 3.6 for BS. Four nodes, with one of the nodes being a destination, are performing volume rendering of $1/4$ of the data, the screen is then split into four equally sized horizontal stripes. During the single compositing step, each node reads-back three of those stripes and sends them to the other three renderers. Each source performs compositing of four parts corresponding to the same screen tile, that it receives from other sources, and then sends color information of a complete tile to the destination node.

Figure 3.8 illustrates in detail operations performed by each source and the data flow for the example given in Figure 3.7. In the beginning of each frame, each source has to *clear* the viewport and *draw* a new portion of the data (these operations are represented with black blocks), after which the compositing operations are done (colored blocks). Each channel reads-back $3\times$ color and opacity information (depth and color buffers in case of z-order compositing for polygonal rendering examples), receives and composites the same amount of data, and, finally, every node, except the destination, reads-back and transmits a portion of the corresponding color buffer, while destination receives and assembles this data

in a sort-first fashion.

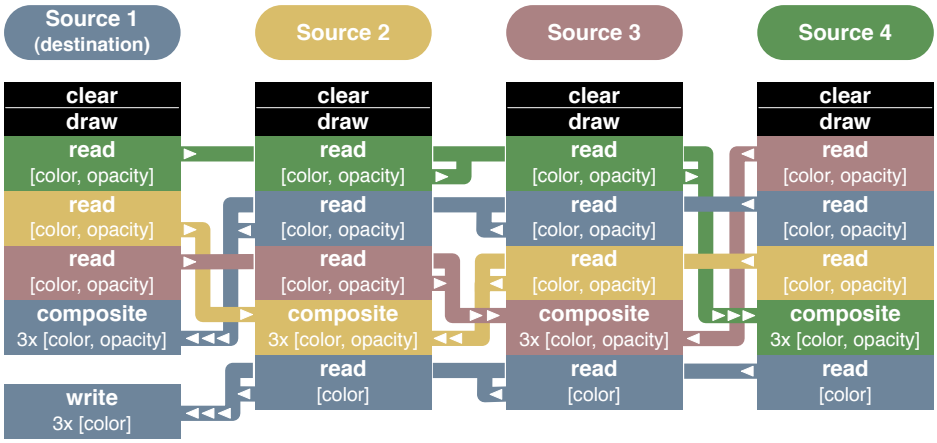


Figure 3.8: *Operations performed by each source and the data flow in case of Direct Send compositing presented on Figure 3.7.*

3.3.3 2-3 Swap Image Compositing

Due to various communication bottlenecks it is not currently possible to implement interactive rendering, where hundreds of nodes contribute to the final image, therefore for interactive visualization smaller number of nodes is usually used, thus DS is typically suitable in such setups. However, for non-interactive rendering with large number of processors, DS produces too many communication messages at the single compositing stage, introducing a significant bottleneck in the switching and compositing hardware. In such scenarios methods like BS is more suitable, due to limited number of communications within the nodes on each stage of compositing. The *2-3 Swap Image Compositing*, proposed by [Yu et al., 2008] is a generalization of the BS method, where the limitation of the power of two nodes from the classical BS is removed.

The observation exploited in *2-3 Swap Compositing* method is that on each stage BS essentially performs DS compositing within pairs of nodes, therefore it is possible to group nodes not only in pairs but also in triplets, where local DS is then performed. This allows for limiting of the communication on each stage to up to four nodes. The method is not described here in detail due to its specific area of application (i.e. supercomputers); more information can be found in the original publication by [Yu et al., 2008].

3.4 Image Data Compression

To reduce transmission cost of pixel data, image compression [Ahrens and Painter, 1998; Yang et al., 2001; Takeuchi et al., 2003; Sano et al., 2004] and screen-space bounding rectangles [Ma et al., 1994; Lee et al., 1996; Yang et al., 2001] have been proposed. However, with modern GPUs these concepts do not always translate to improved parallel rendering as increased screen resolutions and faster geometry throughput impose stronger limits under which circumstances these techniques are still useful. A disproportional growth and shift in compositing-cost that increases with the number of parallel nodes can in fact negatively impact the overall performance, as will be shown later. Therefore, care has to be taken when applying image compression or other data reduction techniques in the image compositing stage of parallel rendering systems.

Basic run-length encoding (RLE) has been used as a fast standard to improve network throughput for interactive image transmission. However, it only gives sufficient results in specific rendering contexts and fails to provide a general improvement as will be shown in Chapter 4. RLE only works to compact large empty or uniform color areas but is often useless for non-trivial full frame color results. Two enhancements to improve the situation, per-component RLE compression and *swizzling* of color bits, are analyzed in the following text.

More complex (and lossy) image compression techniques (e.g., such as LZO or EZW [Shapiro, 1993]) may promise better data reduction, however, at the expense of significantly increased compression cost which renders many solutions infeasible in this context (see Figure 3.11). Additionally, lossy compression (e.g., such as used in VNC [Richardson et al., 1998]) may only be tolerable when compression artifacts are masked by motion and high frame rates. In Chapter 4 the benefit of YUV subsampling, also combined with RLE, are studied, as it can provide very fast and effective compression for scenes in motion, and lossless reconstruction can easily be incorporated by incremental transmission of the missing data from the last frame. Hence the focus is on a few provenly simple and very fast techniques such as RLE and YUV subsampling.

3.4.1 Run-Length Encoding

For the basic RLE method a fast 64-bit version is used. It compares two pixels at the same time (8-bit per channel RGBA format). While this method is very fast it shows poor compression results in most practical settings. A general concept in image compression is to treat color components separately, as illustrated in Figure 3.9. Results on the different RLE schemes are reported in the Chapter 4.

A second improvement is *bit-swizzling* of color values before per-component compression. That way the bits are reordered and interleaved as shown in Fig-

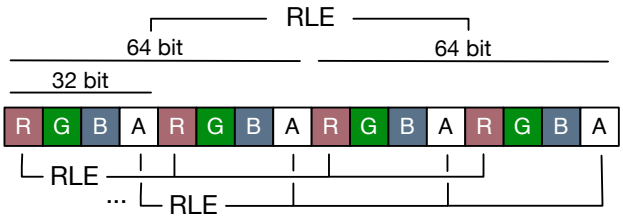


Figure 3.9: *Comparison of 64-bit and per-component RLE.*

ure 3.10. Now per-component RLE compression separately compresses the higher, medium and lower order bits, thus achieving stronger compression for smoothly changing color values.

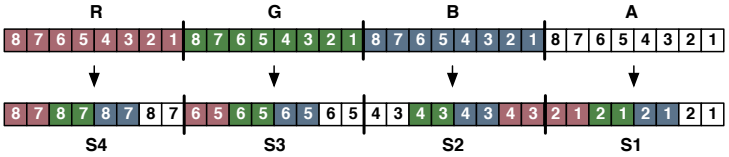


Figure 3.10: *Swizzling scheme for reordering bits of 32-bit RGBA values.*

While LZO is considered to be a very fast general compression technique, Figure 3.11 demonstrates that it performs worse than swizzle RLE when applied in real-time rendering.

3.4.2 YUV Subsampling

Lossy compression in presented study consists of RGB to YUV color transformation and chroma subsampling (4:2:0) since it allows fast computation, good general compression and incremental reconstruction to full chroma color if necessary. Without an additional RLE stage, a compression ratio of $2:1$ can always be achieved this way with good image quality, especially for dynamic motion. Color transformation, subsampling and byte-packing can all be done efficiently in a fragment shader on the GPU such that not only network transmission but also read-back from the frame buffer will be improved.

Figure 3.12 illustrates the YUV subsampling and byte packing. While luminosity values are packed to the left of a 2×2 4-channel pixel block, the chromaticity values are averaged. However, to reduce color distortion at silhouettes only non-zero, non-background color values are averaged. Alpha values are pair-wise averaged on a scan line, and are not needed in the case of sort-first rendering.

The particular color sampling and packing pattern has been chosen to easily support subsequent RLE compression. The above outlined RLE method processes

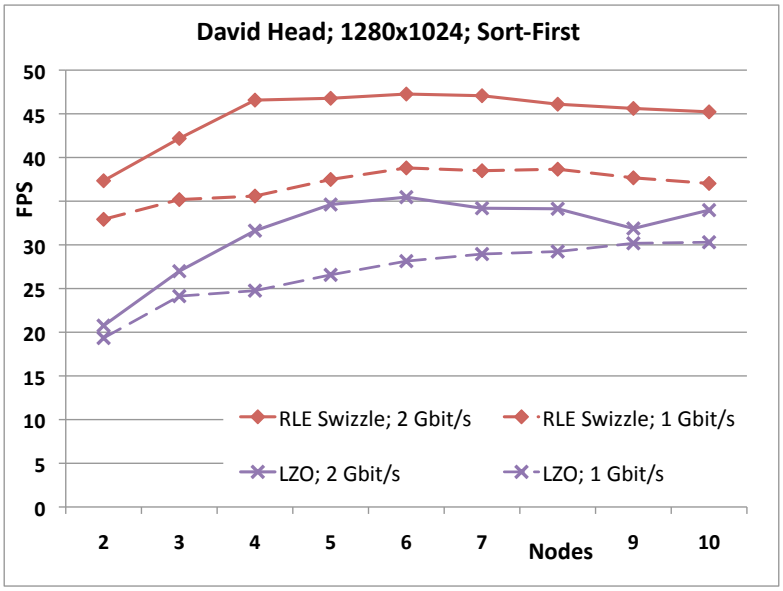


Figure 3.11: *Comparison between LZO and swizzle RLE compression for sort-first rendering of the David Head model. Even though LZO provides better compression, the overall rendering performance is lower due to slower compression times.*

pixels in scan-line order and thus component-wise RLE can directly be applied after YUV transformation and subsampling.

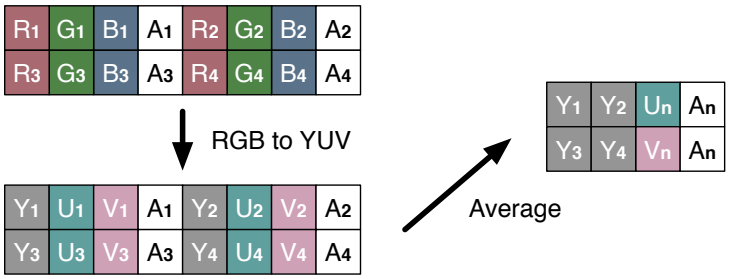


Figure 3.12: *Lossy RGB to YUV transform and subsampling.*

An example of chroma subsampling is presented on Figure 3.13. The artifacts of YUV compression are not visible on the screen, unless significant magnification of the image is performed, the effect is even less noticeable if motion is present in the frame.

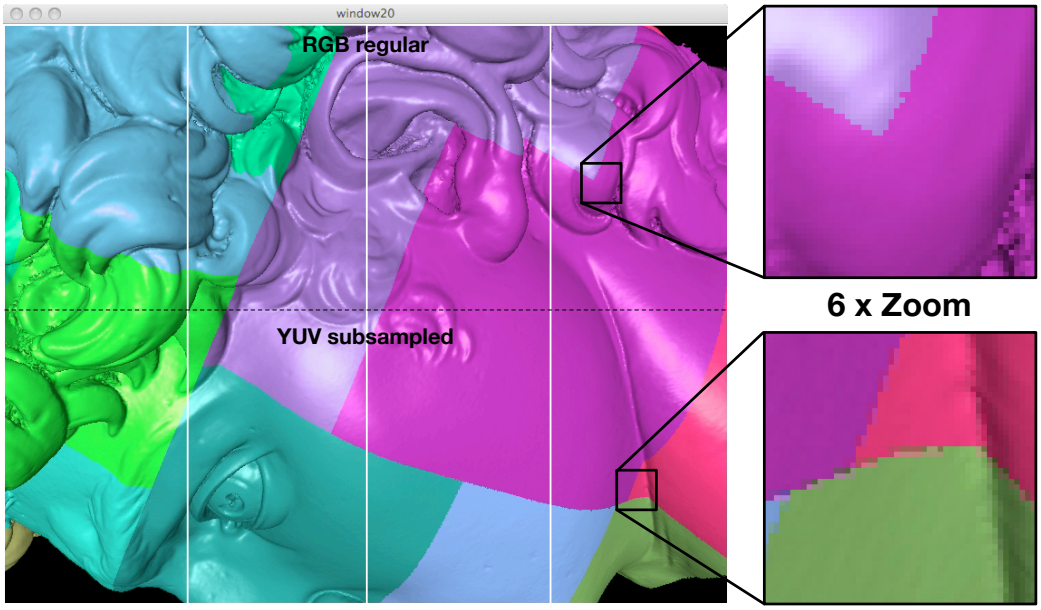


Figure 3.13: *Comparison of YUV compression to the regular uncompressed image. Artifacts can be noticeable in the areas of sharp color edges, only after significant magnification is applied. In this example David Head model is used, painted in pseudo-colors for better effect demonstration.*

CHAPTER 4

PARALLEL RENDERING PERFORMANCE EVALUATION

In this chapter basic performance of Equalizer parallel rendering framework and its various parts are evaluated in order to understand the overhead of the parallel system itself. Additional study is performed to determine which methods are the most efficient for color and depth information compression in case of sort-first and sort-last parallel rendering, and to estimate bottlenecks of polygonal and volume rendering on a PC cluster. Further, some aspects of automatic and manual load balancing for sort-last, and static versus dynamic tiles selection for sort-first, in case of eqRASTeR parallel terrain rendering application, are revealed.

4.1 Hardware Setup and Data Description

Two different PC clusters were used for the evaluation. Most of the tests were carried out on a 10-node cluster, *Hactar*, with the following technical node specifications: dual 2.2GHz AMD Opteron CPUs; 4GB of RAM; one or two GeForce 9800 GX2 GPUs and a high-resolution 2560×1600 pixel LCD panel; 2Gbit Myrinet and switch, as well as 1Gbit Ethernet network. Unless stated otherwise, it is assumed that *Hactar* cluster was used for the test. The second configuration, *Horus*, consisted of 16 nodes with the following technical details: dual 2.4GHz AMD Opteron CPUs, 4GB of RAM (one node has 2 dual-Core 2GHz AMD Opterons and 32GB RAM), Quadro FX4500 PCIe graphics; 1Gbit Ethernet network and

switch as well as local Infiniband 4x networks.

A number of different polygonal, volumetric and elevation (terrain) models of different sizes were used in the study. Table 4.1 lists used models according to their type and size; reference to a screenshot figure (if available) is given in the last column.

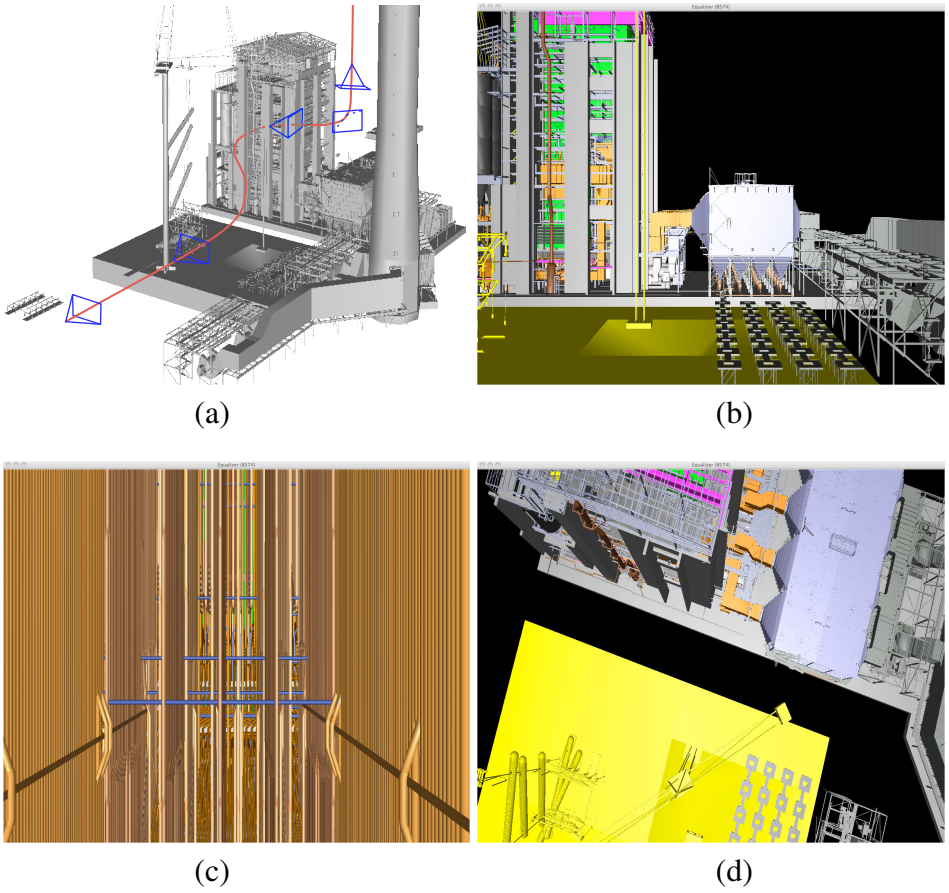
	Model Name	Size	Figures
P	Hip	$1 \cdot 10^6$	Fig. 5.13
P	David head	$4 \cdot 10^6$	Fig. 3.2
P	Power Plant	$13 \cdot 10^6$	Fig. 4.1
P	David 1mm	$56 \cdot 10^6$	Fig. 3.4
V	Bucky Ball	32^3	Fig. 3.6
V	Neghip	64^3	Fig. 6.9
V	MRI Head	256^3	Fig. 3.3
V	Engine	256^3	Fig. 1.1 (c)
V	Femur	$21.6 \cdot 10^9$	Fig. 6.5
T	Zurich	$4^2 \cdot 10^6$	Fig. 1.1 (b)
T	Puget Sound	$16^2 \cdot 10^6$	Fig. 5.12
T	SRTM	$32^2 \cdot 10^6$	Fig. 4.10

Table 4.1: *A list of various models used in this thesis for performance study and visualization purposes. First column stands for the data type:* P - *Polygonal,* V - *Volume, and* T - *DEM (Terrain); size is measured in polygons, voxels and vertices accordingly. Last column references figures containing screenshots of the models if available.*

The tests were typically performed in the following way: in case of compact polygonal or volume models (like *David head* or *Skull*), model was placed in the center of the screen occupying roughly the whole available viewport and rotating around its x and y axes; in case when one dimension was significantly larger (*David 1mm* model), the object would be placed horizontally or vertically in the middle of the screen and rotated around its longest axis (see Figure 4.9). Two additional testing setups were used with *Power Plant* model, where camera path was set to circle around or to pass through the model, they are referenced as *Power Plant Around* and *Power Plant Through* accordingly (or *PPLA* and *PPLT* for short), the screenshots of the *Power Plant* model and corresponding *PPLT* path are given in the Figure 4.1.

For terrain rendering two main tests are conducted: first, where camera swipes above the terrain and another, where camera is zooming into the terrain, they are referenced to as *Turn* and *Zoom* respectively (Figure 4.10 features screenshots form these tests along the aforementioned trajectories); these test are performed using *SRTM* and *Puget Sound* models, also called *32K* and *16K* due to their size.

Additional *Fly Over* setup, with the camera making a larger tour over the terrain is used once with the *Puget Sound* model in the next chapter (Figure 5.12).



(a) (b)

(c) (d)

Figure 4.1: *Typical view of the Power Plant fly-through path(a), and screen shots along the path (b,c,d).*

4.2 Equalizer Framework Performance

First series of tests related to the performance of the Equalizer itself, when no or very little rendering is actually done. These tests are used to prove that the framework doesn't introduce much of the overhead on its own, and to eliminate the possibility of the framework's influence on other, more specific, tests.

Execution loop in Equalizer The application, written using Equalizer framework, solely drives the rendering, that is, it carries out the main rendering loop only, but does not actually execute any rendering. Although depending on the configuration, the application process may also host one or more render

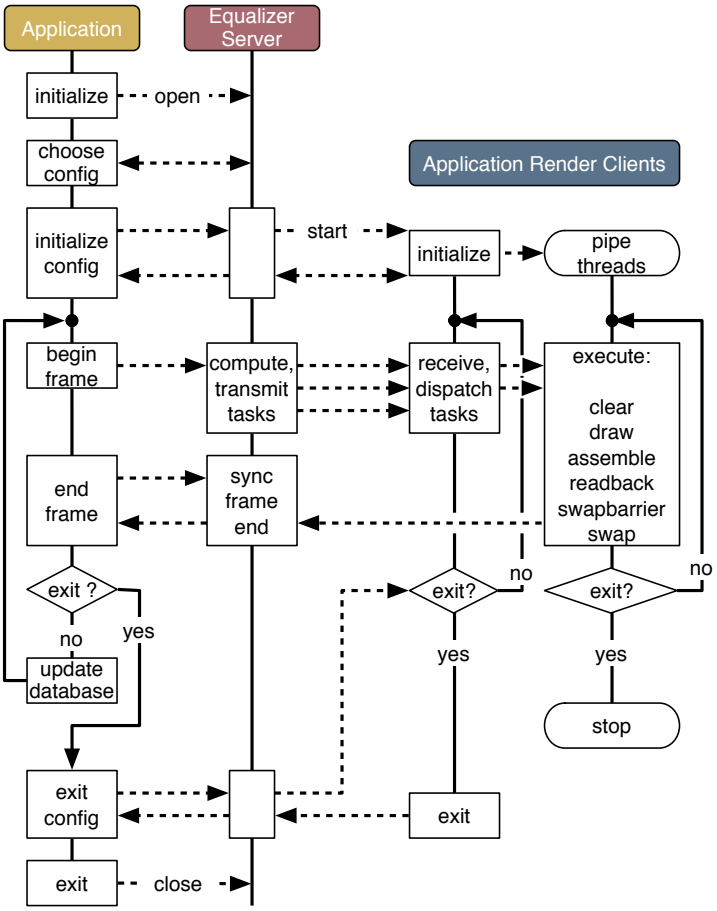


Figure 4.2: *Simplified execution flow of an Equalizer application, omitting event handling and application-node rendering threads.*

client threads, as described below. When a configuration has no additional nodes besides the application node, all application code is executed in the same process, and no network data distribution has to be implemented.

During initialization of the server, the application provides a rendering client. The rendering client is often, especially for simple applications, the same executable as the application. However, in more sophisticated implementations the rendering client may be a thin renderer which only contains the application-specific rendering code. The server deploys this rendering client on all nodes specified in the configuration. The main rendering loop is quite simple: the application requests a new frame to be rendered, synchronizes on the completion of a frame and processes events received from the render clients. Figure 4.2 shows a simplified execution model of an Equalizer application.

Sort-first decomposition Equalizer supports arbitrary rectangular viewport tiling. When multiple machines are combined with a sort-first configuration, a user has to specify which part of the viewport is going to be rendered on which resource. Configuration supports hierarchical screen decompositions and, in case of static tile distribution, screen tiles can be simply assigned in a one level compound; when dynamic load balancing is used, Equalizer can automatically create a *k*-d tree-like partitioning of the screen and then for every frame adjust the size of created tiles based on their rendering timings from previous frames.

Sort-last range specification When sort-last decomposition is used, Equalizer operates on so-called "*data ranges*". In this mode the whole data is mapped to a one-dimensional range of $[0..1.0]$. This means that Equalizer can request different renderers to evaluate various parts of the data by assigning different intervals from this "*complete*" data range; if the different ranges cover the whole $[0..1.0]$ interval, then after compositing final viewport will have complete data, as if it would be rendered on a single machine. In terms of Equalizer configuration, range $[0..0.5]$, for example, would mean "*first half*" of the data and $[0.5..1.0]$ - "*second half*", if rendered by two machines the total amount would be 100%. Mapping of this range to actual data values is left to an application developer.

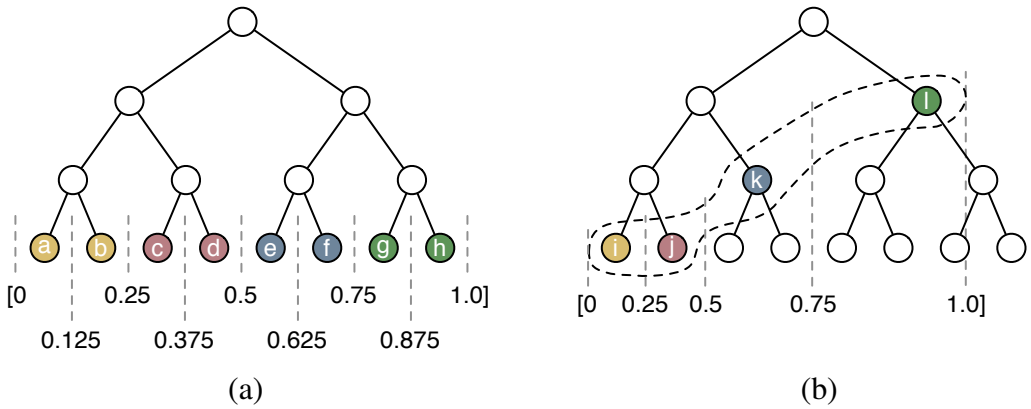


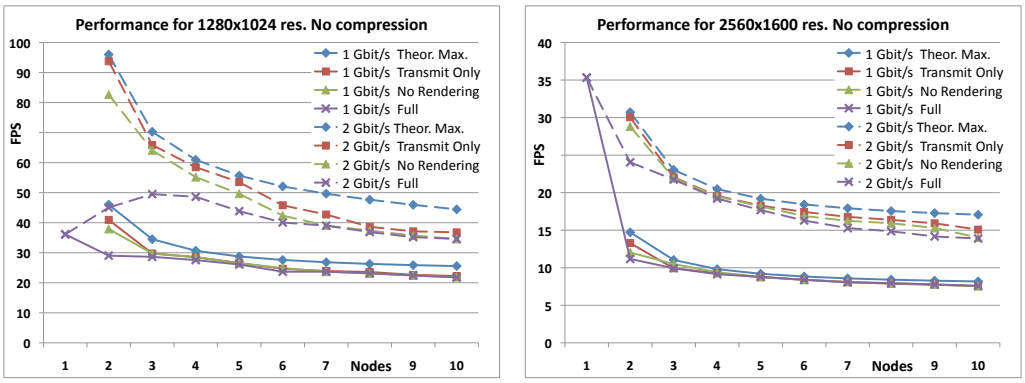
Figure 4.3: *Difference in sort-last range selection between eqPly (a) and eqRASTeR (b) applications. In case of eqPly the whole tree is mapped to the [0..1] range, while eqRASTeR maps the same range to the rendering data selected for the current frame.*

For the eVolve volume rendering application one-dimensional data range is directly mapped to one of the volume's dimensions, creating volume slabs as demonstrated in the Figure 3.3, where full range was split into seven consecutive sub-ranges of equal size, and compositing was done according to explanations given in Section 3.2. In case of eqPly polygonal renderer, the range is mapped to a *k*-d tree structure that contains polygonal data, the mapping is done accord-

ing to the number of triangles contained in each subtree; the final data is stored only within leaf nodes, since no LOD technique is applied. In eqRASTeR the strategy for range mapping is different. For every frame eqRASTeR selects a number of tree nodes for rendering, then range mapping is done over these selected elements. Figure 4.3 demonstrates this difference: no matter what data is visible within final viewport, if four renderers are assigned equal ranges $[0..0.25]$, $[0.25..0.5]$, $[0.5..0.75]$, and $[0.75..1.0]$, renderers in eqPly will always render tree nodes *ab*, *cd*, *ef*, and *gh* accordingly, where eqRASTeR first would select a subset of the data, depending on estimated LOD, then split the work between four renderers equally, *i*, *j*, *k* and *l* is just one example of the outcome.

Rendering examples of equal range subdivision for eqPly are presented on Figures 3.4 and 3.13, where different colors correspond to different data ranges. An example of eqRASTeR sort-last decomposition can be found in Figure 5.12.

Basic evaluation First, the typical distributed image compositing and thus parallel rendering performance limits are identified, when image compression is not used. This will also demonstrate that the used parallel rendering framework is in fact very resource efficient and achieves the expected limits.



(a) (b)

Figure 4.4: *Maximal theoretical and real image throughputs for (a) 1280x1024px and (b) 2560x1600px resolutions.*

Figure 4.4 shows different image throughput limits for two different frame buffer resolutions, without using image compression, in the context of sort-first rendering. In sort-first rendering, given N rendering nodes (one of which is also the display node), the network will be loaded with the cost of transmitting $(N - 1)/N$ of the full frame buffer image data to the final display node. Hence with increasing N, the image transmission throughput and thus compositing speed is expected to decrease. In the limit ($N \to \infty$) a maximal frame rate of μ/s_{FB} can

be expected for a given network bandwidth μ and frame buffer size s_{FB}. Using *netperf*, a realistic achievable data transmission rate of $\mu = 115MB/s$ and $\mu \leq 240MB/s$ for 1Gbit and 2Gbit networks is evaluated. Thus for a frame buffer size s_{FB} of 1280×1024 ($5MB$) up to 23 fps or 48 fps is expected, and for 2560×1600 ($16MB$) 7 fps or 15 fps respectively for the different network speeds. The maximal achievable frame rates limited by the bandwidth as described above are indicated in Figure 4.4 with Theor. Max..

Ignoring any rendering costs but only focusing on image throughput and compositing, the experiments show that chosen parallel rendering setup is efficient and approaches the expected limits. The Transmit Only graphs in Figure 4.4 indicate the system's limit simply for transmitting the partial frame buffer results to the destination, which closely follows the expected limits in particular for the larger frame buffer. The No Rendering curves for the entire sort-first compositing task, but still not accounting for actual rendering itself, indicate the hard limits of the rendering system if no image compression is used. These results still closely follow the bandwidth constraints and theoretical expected limits, and thus demonstrate that the sort-first compositing stage does not introduce any significant overhead.

A full rendering test, labeled as Full in Figure 4.4, with only a small polygonal object, further confirms the limits and resource efficiency of the system. The curves show that the frame rate is quickly limited by the image throughput and decreases accordingly for larger N. Only for a small frame buffer and slow network configuration the frame rate initially increases when adding rendering nodes until being dominated by the image throughput constraints. Hence apparently there is no notable overhead introduced in the parallel rendering system.

Basic scalability for larger models is confirmed in Figures 4.5(a), 4.5(c), and 4.5(e), showing what can be reached in the best case just from rendering, this time not taking any image transmission and compositing into account. Both sort-first and sort-last parallel rendering improve rendering speed almost linearly for uniformly distributed geometry (Figure 4.5(a)), However, sort-first scalability starts to flatten out at some point as expected due to smaller viewports but constant per-node culling costs. For the large Power Plant model, the results in Figures 4.5(c) and 4.5(e) show that sort-last rendering can scale superlinearly due to GPU caching effects. It also shows that for uneven distributed geometry, a regular sort-first screen decomposition cannot improve rendering speed, which is expected as well.

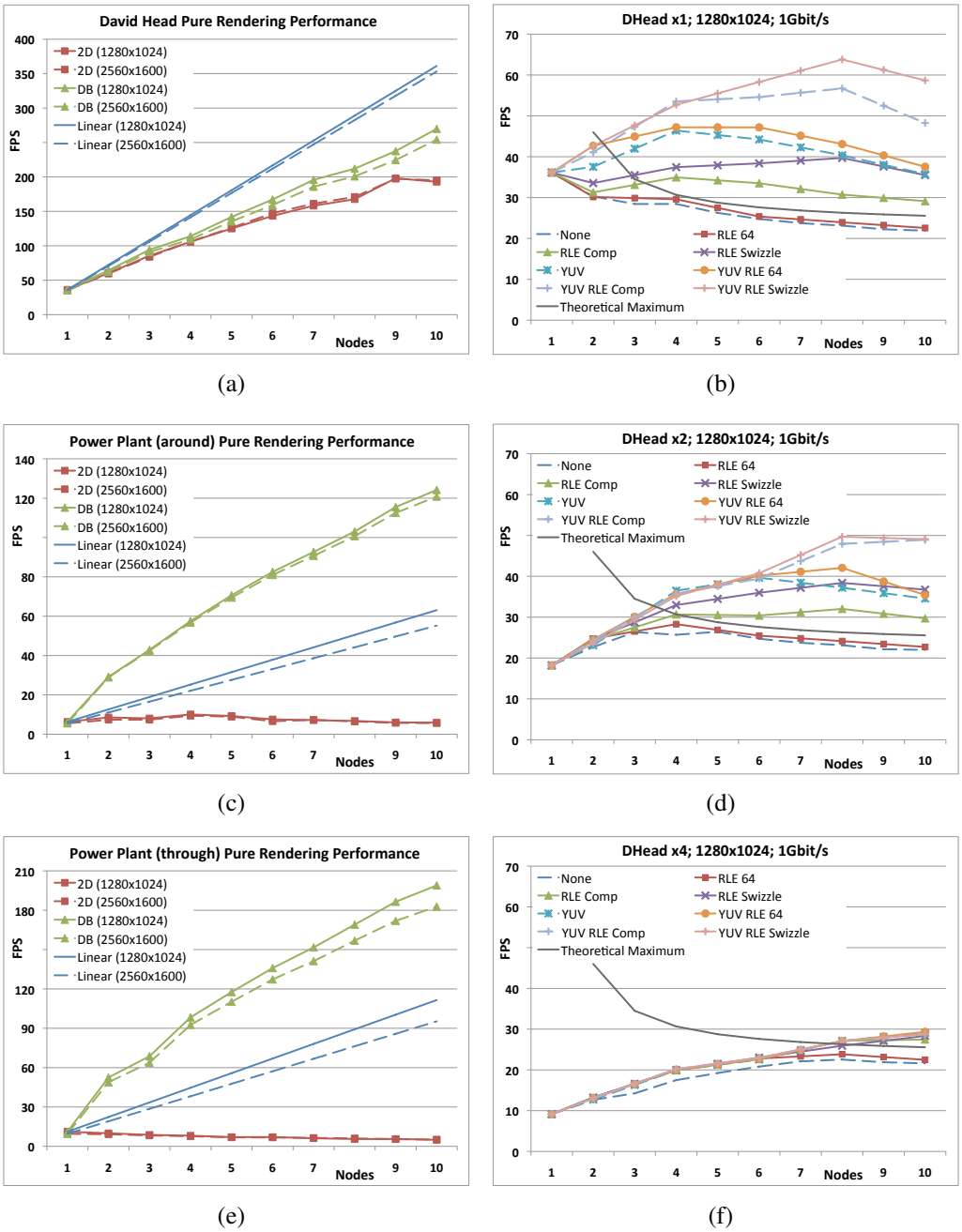


Figure 4.5: *Rendering-only performance: (a) David Head, (c) Power Plant (fly-around), and (e) Power Plant (fly-through). Different color compression for David Head model: (b) normal rendering speed, (d) double rendering load, (f) four times rendering load (same model rendered one, two and four times respectively). Theoretical maximum lines correspond to the best speed possible when only considering the time required for uncompressed image transmission over the given network.*

4.3 Color Compression in Sort-First

To evaluate color compression benefits in relation to the basic image throughput RLE, per-component RLE, swizzle RLE and YUV subsampling have been analyzed. While YUV is lossy, it can often be used without noticeable loss in visual quality (see also Figure 3.13), and it can be combined with RLE. Figures 4.5(b), 4.5(d), and 4.5(f) show the overall frame rate due to image compression for varying geometric model complexity. It shows that basic RLE does not help for non-uniform color images. While per-component or swizzle RLE are more costly, they can achieve an improvement. Swizzle RLE works reasonably well as it can improve image transmission more significantly.

YUV subsampling reduces the chromatic color components by a factor of 4 and thus the total image size by 2 at minimal extra image processing cost. This data reduction shows immediate effects on the frame rate as shown in Figures 4.5(b) and 4.5(d). It is also confirmed that basic RLE does not improve upon YUV (compare curves YUV with YUV RLE 64). However, per-component or swizzle RLE based compression on top of YUV subsampling can further improve the image throughput and overall frame rate (see curves YUV RLE Comp with YUV RLE Swizzle).

Figures 4.5(d) and 4.5(f) show the effect of pure rendering limits for larger models and consequently achievable parallel speedup. If the 3D data complexity is so high that rendering itself is the bottleneck, adding more nodes improves sort-first parallel rendering until the image throughput limit is reached (see curves None with RLE 64). Only after that intersection point, image compression has a noticeable effect.

Figure 4.5 furthermore confirms the $23fps$ limit for $N \to \infty$, with $s_{\mathrm{FB}} = 1280 \times 1024$ screen and bandwidth $\mu = 115MB/s$ since all measures converge to that, unless compression is applied. These results will scale appropriately with screen resolution and network bandwidth.

The complete composing performance is defined by:

1. Read-back;
2. Compression (if compression is used);
3. Transmit;
4. Decompression (if compression is used);
5. Per-image compositing.

The results in Figure 4.5 help to further understand the influence of different approaches to sort-first rendering.

In the tests above the per-vertex colored David Head model was placed in the center and nearly covering the full screen. The animation rotated the model

around x and y. The results confirm that image transmission has the most significant influence compared to read-back and compositing which cause very little overhead. Basic RLE compression proves to have poor performance and only swizzle RLE can sufficiently compensate extra compression cost with increased image throughput, outperforming in total other RLE versions.

YUV subsampling alone enhances performance due to the fixed data reduction, which can further be improved in combination with per-component or swizzle RLE.

RLE compression is implemented using OpenMP on the CPU, however, the parallel framework is already running four threads and the CPUs are fully used. If more than four cores are available, one could expect improved performance of RLE (doubling of RLE speed on 2 cores was observed, when tested without the rest of the framework running).

4.4 Sort-Last Performance

Sort-last parallel polygon rendering uses z-buffer in order to perform z-value compositing of partial rendering results, as it was explained in the Section 3.2. Thus the z-depth buffer data also has to be compressed and sent over the network. The compression methods in case of sort-last are determined for depth buffer first, and then for color component, to ensure optimal results.

Depth component compression To determine the best depth component compression color compression was set to RLE and serial sort-last compositing was used, which imposes a network image transmission load proportional to the number N of rendering nodes. Color RLE was used since it removes blank screen space effectively which is typical for sort-last rendering. In Figure 4.6 uncompressed depth data is indicated in the graph by None. Measures are shown for different data models (David Head, Power Plant around and fly-through), network bandwidth and for $N = 2, 6, 10$ nodes.

For uniform partitioning of geometry each node is rendering $1/N$ of the data and less screen space is covered with increasing N for sort-last rendering. Hence empty-pixel skipping is more important than actual depth-value compression to transmit the full-frame sized depth buffer to the final destination node. This is supported in Figure 4.6 with simple RLE depth compression performing better than the more complex per-component or swizzle variants.

However, in the case where partitioning of the data does not lead to sparse images for sort-last compositing, but most of the frame buffer is covered, per-component RLE compression shows better results. This experiment is reported

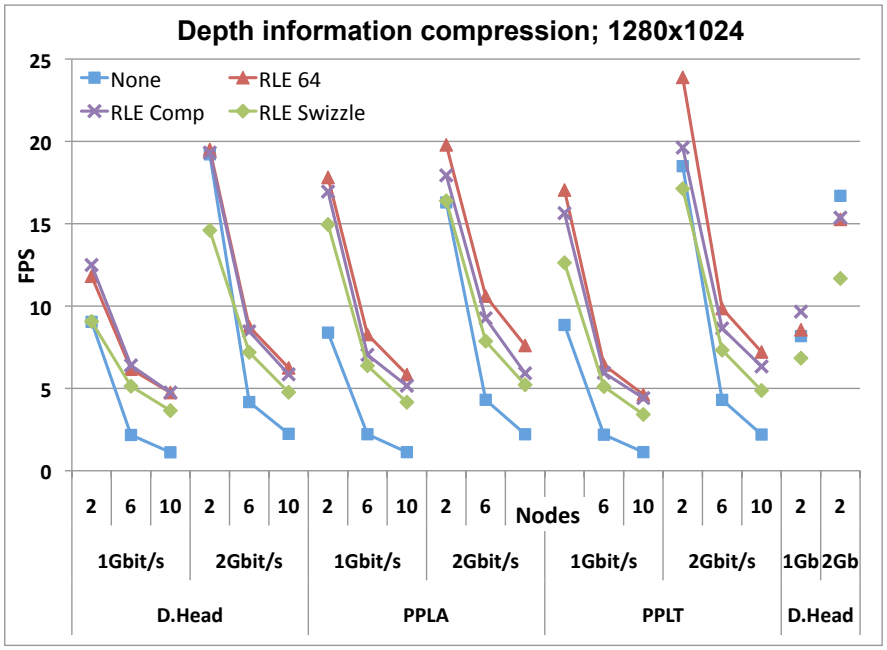


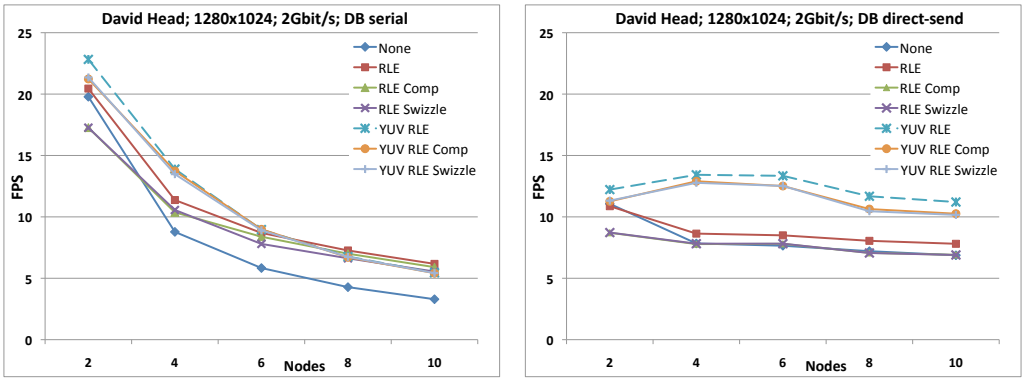
Figure 4.6: *Image throughput performance comparison of different depth-component compression methods.*

in the last two columns of Figure 4.6 where the model is rendered twice on two nodes, covering the entire frame buffer on both nodes.

Color component compression To determine the best color compression depth compression was fixed to RLE. As observed before, the images generated from sort-last rendering are likely to have large empty regions with increasing N. When using Direct Send (DS) compositing (details are given in Section 3.3.2), each node renders a part of the 3D data into a full-sized frame buffer, followed by depth-compositing of a part of the entire viewport. The composited sub-regions are eventually sent to the destination node for final assembly.

Figure 4.7(a) shows that simple RLE compression is effective for color if increasingly larger parts of the partially rendered images are empty. YUV subsampling has much less effect for such sparse image data. Almost identical results were found for rendering of the Power Plant model.

In contrast to serial compositing, DS exhibits a constant amount of image data that has to be transmitted between the parallel rendering nodes, which is also evidenced in Figure 4.7(b). Due to the split-frame compositing and different distribution of rendering and pixel data, YUV subsampling can notably improve image throughput here.



(a) (b)

Figure 4.7: *Image throughput comparison of different color compression methods for (a) serial and (b) Direct Send sort-last compositing.*

The results suggest that for a small number of nodes, depending on the network bandwidth, serial compositing performs better than DS. DS will be beneficial for a larger number of rendering nodes and large complex 3D data sets.

4.5 Distributed vs. Streaming Systems

Despite Equalizer and Chromium having slightly different main targets, flexible configuration and scalability on one side and transparent abstraction of the OpenGL API on the other side, only a limited experimental evaluation is provided here.

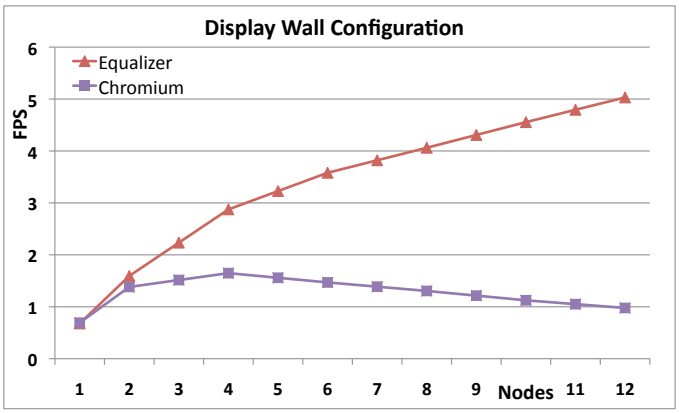


Figure 4.8: *Frame rate performance comparison between Chromium and Equalizer for tiled display wall configurations of up to 12 screens and nodes.* (Horus)

For this test a simple display wall configuration was used, as shown in Figure 4.9, with a static model, rotating about its vertical axis, placed such that it sufficiently covers the different screens. A standard tile-sort Chromium configuration has been compared to a simple Equalizer display-wall compound setup. The polygonal model is rendered using eqPly and uses display lists for the static geometry. Using display lists allows Chromium to send geometry and texture data once to the rendering nodes (retained mode rendering) and display them repeatedly using *glCallLists()* which is inexpensive in terms of network overhead [Bethel et al., 2003].

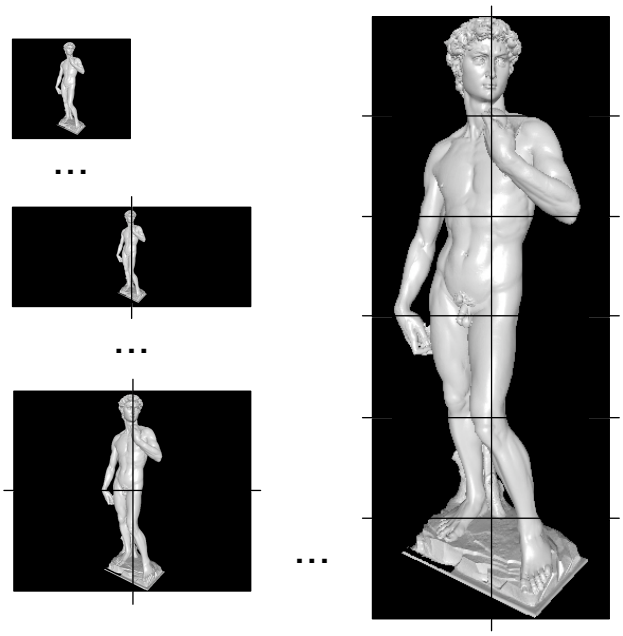


Figure 4.9: *Display wall configurations to compare Equalizer and Chromium using 1, 2, 4, . . . and 12 screens and rendering nodes.*

According to [Humphreys et al., 2002; Bethel et al., 2003; Staadt et al., 2003], a tile-sort display-wall setup with static geometry rendered in retained mode should be reasonably favorable for Chromium because the display lists have to be transmitted only once over the network, and only simple display calls will be processed and distributed by Chromium for each rendered frame. Figure 4.8 shows the experimental results of the display-wall comparison between Chromium and Equalizer. One can clearly observe that while Chromium initially increases performance when adding nodes, it quickly stagnates and even decreases when more nodes are added. In contrast, Equalizer continually improves performance with more added nodes and only exhibits a smooth drop-off in speedup, due to the expected synchronization and network overhead as the rendered data gets negligible in size per node. This performance difference may also be due to

the fact that Equalizer can benefit from distributed parallel view-frustum culling.

Similar results were demonstrated by [Doerr and Kuester, 2011], where CGLX framework was used to manage display wall installation. These tests also confirm that Chromium is suitable only in case of low geometry count, and distributed rendering solutions are necessary otherwise.

4.6 Terrain Rendering Application Performance

Porting standalone applications to parallel rendering frameworks like Equalizer is relatively simple, in case when no special care about data distribution has to be worried about. In fact, in order to support sort-first decomposition, very little work is required. Any rendering application has a viewport and a camera setup for using a single window, developer has to replace those initializations with the information provided by the framework, after which sort-first, stereo, time multiplex, display wall and any combinations of these setups are immediately become available, if Equalizer is used.

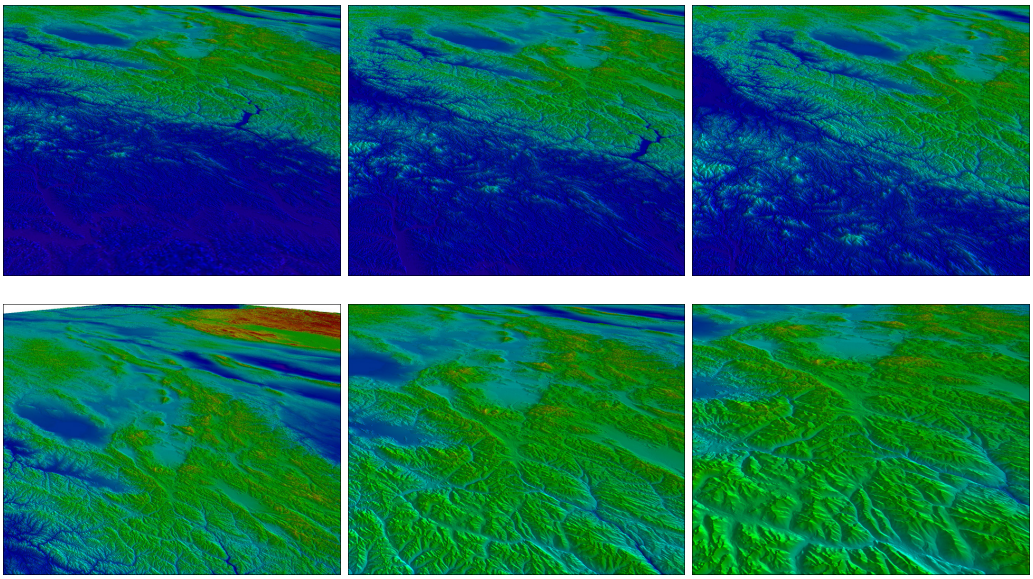
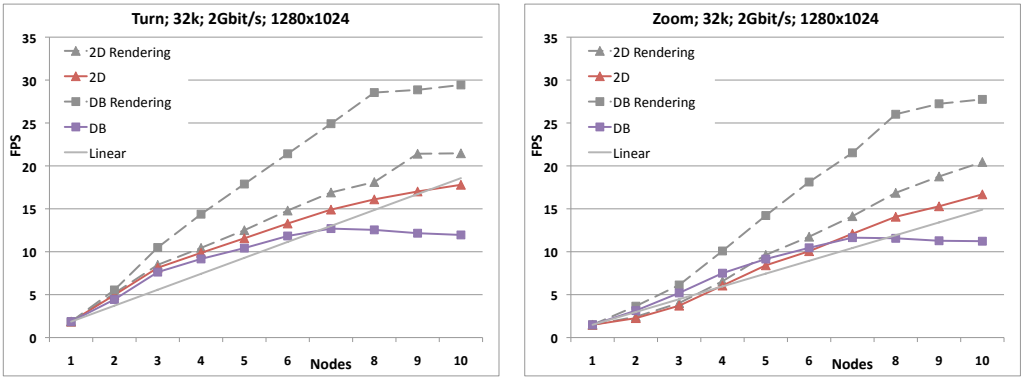


Figure 4.10: *Frames from two test sequences of eqRASTeR terrain renderer. Top row:* Turn *(camera is flying and turning in horizontal plane); bottom row:* Zoom *(camera zooms into the terrain).*

In order to support sort-last decomposition, more work is necessary. There are two major tasks to be solved: data range interpretation and partial image compositing. One-dimensional data range of $[0..1]$ provided by Equalizer has to be mapped to the actual data, as explained in the Section 4.2, and if any transparency

was used during the rendering, the developer has to take care about correct compositing order. Equalizer provides default implementation of z-buffer-based compositing of opaque geometry, therefore no special care is required if transparency was not used (one option, in case of small number of semitransparent data within the frame, is to draw all semitransparent objects in a second pass on the destination viewport only, this way all z-buffer-based compositing would be finished before and will not affect semitransparent regions).

Following these guidelines RASTeR terrain rendering application was ported to Equalizer (eqRASTeR) in order to support parallel rendering capabilities. Standalone RASTeR employs out-of-core rendering, thus data management was not an issue, further, RASTeR only performs opaque geometry rendering, therefore default compositing of Equalizer was suitable. Details about RASTeR and its parallel implementation can be found in [Goswami et al., 2010].



(a) (b)

Figure 4.11: *Graphs showing rendering performance on 10 machines in parallel using DEM models of 32k×32k SRTM grids with camera in turning and zooming trajectories respectively. 2D - Rendering refers to sort-first rendering, 2D to sort-first rendering with compositing, DB - Rendering to sort-last rendering, DB to sort-last rendering with compositing.*

eqRASTeR application was then evaluated on the Hactar cluster. Basic scalability tests were performed using sort-first and sort-last decomposition modes. Static vertical tiles were used for sort-first, as they provide nearly equal triangle per-renderer distribution in this particular application, and *data range* to *rendering set* mapping was done in case of sort-last, as explained earlier in this chapter. Additionally, Regions Of Interest method was used to accelerate read-back transmission and compositing (the method is explained later in the Chapter 5).

Figure 4.11(a) present frame rate graph for moving forward and turning camera trajectory, while Figure 4.11(b) presents one for the camera zooming into the terrain (screenshots of the corresponding test sequences are presented on Fig-

ure 4.10). As could be seen from these graphs, in both sort-first (2D) and sort-last (DB) parallel rendering modes, pure drawing performance (labeled as 2D - Rendering and DB - Rendering respectively) scales at least linearly. Super-linear performance of rendering can be explained by reduced data fetching since each machine fetches only those terrain blocks that are not already cached locally in main memory. The reduced size of the rendering set on a single machine allows it to be cached more efficiently in GPU and main memory, hence avoiding repeated data fetching. Pure sort-last rendering scales better than sort-first because each machine renders a similar number of triangles, thus data is well distributed among them. However, this cannot be ensured in the case of sort-first task decomposition.

Overall rendering performance depends largely on the compositing stage of the parallel rendering framework, which includes reading of partial images back from GPUs, transmitting them to the destination node and assembling final frames for display. The decrease of the final performance (labeled as 2D and DB) with increasing number of nodes on Figure 4.11 happens due to the image throughput bottleneck. The amount of data that has to be sent over the network in case of sort-last rendering and compositing is roughly twice larger than for sort-first, thus network saturation happens earlier despite the rendering itself being faster. In presented case, sort-last network saturation happens at around $15fps$, which is independent of the drawing speed. That means if the initial rendering on one node is already fast, overall performance will not scale well with more rendering nodes.

The basic scalability tests demonstrate that performance of distributed RASTeR rendering scales very well. Overall performance, however, is mostly limited by network throughput and by the compression used for partial images.

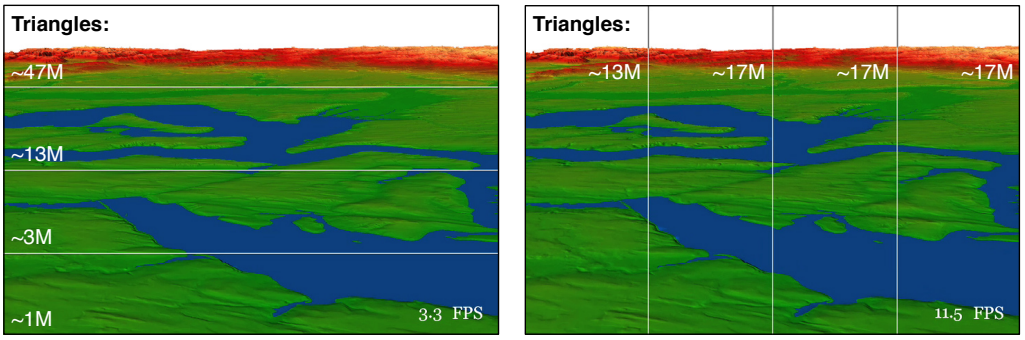
4.7 Automatic vs. Manual Load Balancing

Static distribution of rendering tasks is hardly desirable in an interactive rendering environment. Changes in the viewport and camera setup would usually lead to unequal load on rendering devices, therefore screen tiles or data selection have to be adjusted on the fly. Equalizer solves this problem by measuring time that was spent on rendering of different tiles or data intervals, and making an adjustment of tile or data range sizes dynamically. Despite these measurements only accommodating for previous time information and not performing any forward correction, automatic LB works fairly well in eqPly and eVolve, where the whole rendering data fits to RAM and GPU memory. In case of eqRASTeR the situation is different because out-of-core rendering require data fetching from slow local HDD, or even slower network storage.

The following two experiments were performed in order to understand if automatic load balancing can improve rendering performance of a reasonably sophis-

ticated application which uses its own data management strategy, where default tile and data split are replaced with load balanced solution.

It was shown by [Goswami et al., 2010] that vertical tiles are preferable over horizontal once in sort-first decomposition mode. The horizon of the terrain, rendered by eqRASTeR, contains more polygons than the foreground, since more geometry is visible within the same screen area there, and, due to regular nature of the elevation data, vertical tiles of similar width result in almost even triangle count per stripe (because elevation shifts happen within the same stripe). Figure 4.12 demonstrates this effect clearly.



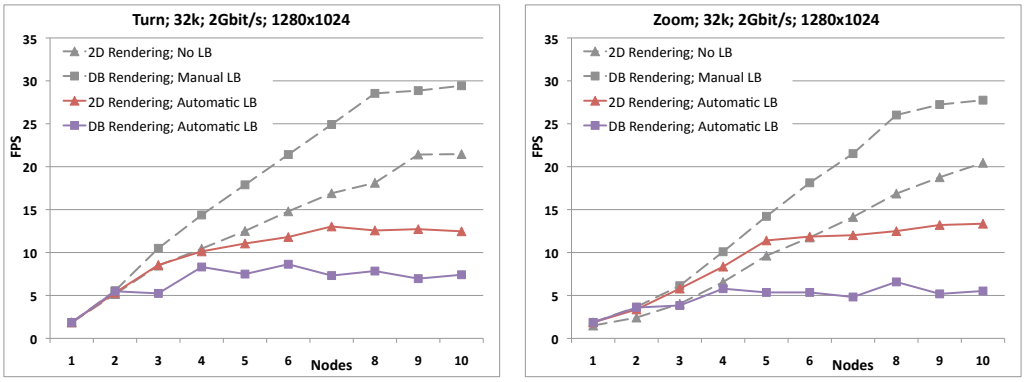
(a) (b)

Figure 4.12: *Triangle count (millions) in horizontal (a) and vertical (b) tiles in terrain rendering with eqRASTeR. Vertical tiles have closer to even triangle count per tile, and therefore more suitable for rendering in parallel.*

In case of sort-last, data range was already mapped in such a way, that equally sized range intervals would result in similar number of triangles rendered by each node, therefore data was already manually load-balanced in this sense. Performance results were satisfying since the homogeneous system with equal rendering resources on each node was used for these tests, however such manual balancing might not work that well in a heterogeneous system. Enabling load balancing over data ranges allows potential inequality in the triangle count, and therefore allows processing of more data by faster renderers. Evaluating dynamic load balancing in case of eqRASTeR will show if automatic range adjustment can at least maintain similar results to manually assigned decomposition in heterogeneous setup.

Figure 4.13 shows the obtained result of the rendering only evaluation. The setup was similar to the one used earlier in Section 4.6 for basic performance evaluation. Lines 2D - Rendering; No LB and DB - Rendering; No LB correspond to 2D - Rendering and DB - Rendering for sort-first and sort-last decompositions on the Figure 4.11 accordingly. New data is represented by 2D - Rendering; Automatic LB and DB - Rendering; Automatic LB.

The curves, where automatic load balancing was used, are flattens out very



(a) (b)

Figure 4.13: *Rendering performance comparison of static tile/data split and dynamic load balancing for eqRASTeR application. Sort-first and sort-last (2D and DB on the graphs respectively) decomposition modes are considered, however compositing is omitted in order to ignore network overhead.*

quickly, performance is only improving when few first nodes are added, and performance of sort-last balanced rendering is particularly bad. The observed behavior is explained in the following way: since only rendering time is considered, not taking into account data management and data loading costs, Equalizer framework tends to shift load to few machines which were able to load more data faster than the other. These few machines would have most of the data cached in their RAMs and are spending time performing most of the rendering, being awarded by load balancer with even more data to render, while the rest of the nodes are busy updating their caches, resulting in poor overall performance.

CHAPTER 5

COMPOSITING OPTIMIZATION

Distributed cluster-parallel image compositing, even if only for image assembly as in sort-first rendering, is fundamentally limited by the network throughput that bounds the amount of image data that can be exchanged between nodes, as was shown in the Chapter 4. Hence efficient image analysis, compression and transmission techniques must be considered in this context. The performance aspects of any optimization has to be carefully studied within the fully functioning rendering system in order to understand how these changes affect overall efficiency of the framework. Furthermore, only generic methods should be considered, when implemented without prior knowledge about the rendering algorithm, as it is usually the case when general parallel rendering library is developed.

5.1 Compositing Loop

Slightly simplified execution flow of parallel rendering is presented on Figure 5.1(a). On every frame, after all of the initialization is done, every renderer sequentially performs drawing (*clear*, *draw*), compositing and synchronization (*swap*) operations, where in the simplest case application developer has to implement only drawing functionality. During the compositing stage (Figure 5.1(b)) several iteration of image compositing and assembly can occur, depending on the compositing configuration.

Initially pixel information is contained in the *Frame Buffer* on the GPU (in case of GPU-based rendering), in the general case, this data has to be read-back, and

transmitted to other nodes, as well as used for in-place compositing. Some data processing and possibly compression can occur on the GPU itself (denoted on the Figure 5.1(b) as *Processing 1*), where inherent parallelism of GPUs is exploited. Certain information is then *Read-Back* to the main memory, possibly processed in the RAM (*Processing 2*), compressed and sent over the network. The same node would simultaneously receive already processed data from other nodes (*Get Data*), decompress and use it for compositing (*Compose*). Since compositing can happen both on GPU and CPU, the composited data will end up either in the RAM or in the VRAM again, after which the compositing loop is closed, and the whole process can be repeated again.

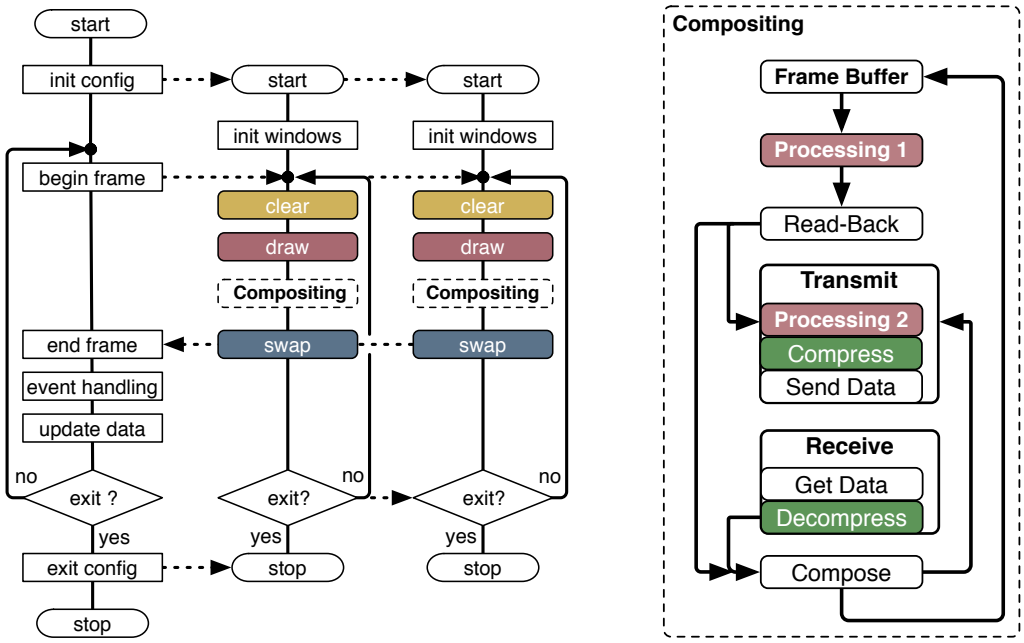


Figure 5.1: *Simplified execution flow of parallel rendering (a) and compositing loop of Equalizer framework (b).*

There is limited room for optimization, with additional constrains of strict performance requirements. In case of the unknown general rendering algorithm, very few assumptions could be made by the parallel rendering framework developer. Rendered output can be analyzed and some types of image data compression could be evaluated on the GPU itself, during *Processing 1* stage. For example YUV compression and decompression, presented in the Section 3.4.2 and evaluated in the Chapter 4, are fully implemented on the GPU. Other compression methods, like RLE are not ”GPU friendly” due to their linear nature and have to be exe-

cuted by CPU in the RAM, after the *Read-Back* is done. The compression itself has to be lossless or introduce minimum amount of artifacts, especially for the depth buffer, where compression errors can result in significant artifacts during compositing stage.

5.2 ROI-based Compositing

Distributed parallel image compositing cost is directly dependent on how much data has to be sent over the network, which in turn is related to how much screen space is actively covered. Additionally, read-back and transmission times are also affected by image color and compression formats.

In the following a generic sparse-image representation approach that can be used in any sort-first or sort-last parallel rendering configuration, is introduced. Further data reduction can be gained with image compression. However, this must meet demanding requirements, as its overhead has to be strictly smaller than any transmission gainings, which can be difficult to achieve, as explained earlier in the Chapter 4.

In sort-last rendering, every node potentially renders into the entire frame buffer. With an increasing number of nodes the set of affected pixels typically decreases, leaving blank areas that can be omitted for transmission and compositing.

The proposed Region-Of-Interest (ROI) algorithm splits the frame buffer into parts with active pixels and excluded blank areas. The active ROI is less or equal to the frame buffer size and depends on how many pixels have actually been generated. Thus the maximal benefit will be reached when each node renders only to a compact region in the frame buffer. This assumption largely holds for hierarchically structured data, which is often used to accelerate culling and rendering.

The ROI algorithm is called when all rendering is finished right before read-back. Identified subregions are individually treated for Read-Back, Compression and Transmission (RBCT) as well as compositing, any of which is reused from the original parallel rendering framework.

5.2.1 ROI Selection

For an efficient RBCT process, there are several desired features that final regions should exhibit:

- Compact rectangular shape;
- Coverage of all generated pixels in the frame buffer;
- No region overlap;
- Smallest possible area for limited number of regions.

Since the number of regions must be limited to avoid excessive GPU read-back requests, the last feature is the most difficult to achieve. The proposed ROI method reformulates the last criterion as follows: the aim is to exclude as much of the blank frame buffer area as possible with as little effort as possible. This solution preserves the other criteria while effectively removing the undesired blank spaces.

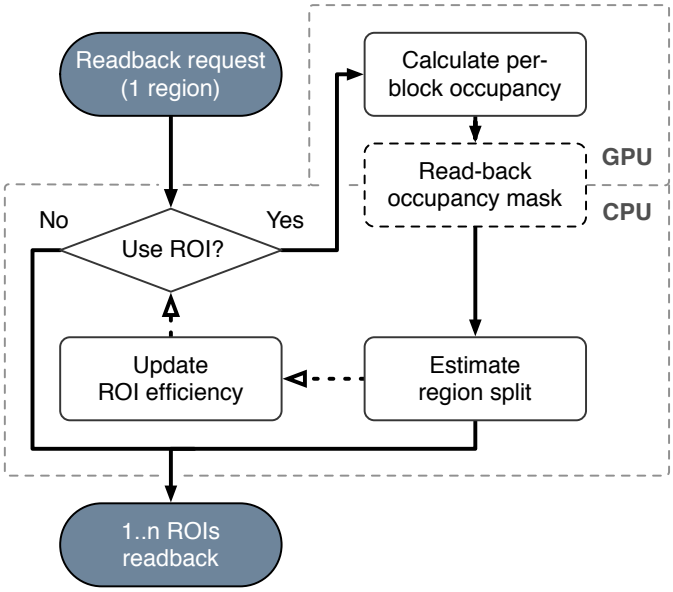


Figure 5.2: *ROI selection algorithm.*

ROI selection itself requires time and if its area is not significantly smaller than the entire frame buffer it may affect performance negatively due to overhead. The decision whether to use ROIs or not is made dynamically while identifying blank regions. If the blank area is too small with respect to the frame buffer size, ROI selection *fails* and is turned off, delayed for one frame, meaning the whole screen will be read-back in the next frame. Every consecutive ROI failure causes an increasing delay to the next ROI estimation up to a certain limit at which ROI estimation is done at a regular periodic rate. Currently this limit is at 64 frames, equaling to a few seconds of real-time interaction and minimizing overhead of failed ROI estimation.

Since rendering framework can request different areas from the same window, rather than the entire frame buffer, and these viewports can change dynamically due to automatic load balancing, requested regions have to be tracked within ROI method in order to update ROI efficiency correctly. Proposed method solves region tracking problem by storing information about all requested regions from previous frame. When read-back requests are passed to the ROI algorithm for a new frame, it compares first areas of these new requests to the once that are already available. If overlap of the new and one of the previously defined regions

exceeds $2/3$ of the old region's area, the match considered to be positive, new area replaces the matched one and inherits its ROI performance statistics. If no proper match is found, the area considered to be completely new and ROI algorithm is executed without the use of performance statistics.

The algorithm itself consists of two main parts: per-block occupancy calculation performed on the GPU and region split estimation executed on the CPU as indicated in Figure 5.2.

To speed up the region split process, a block-based occupancy mask is used. This mask is computed using a fragment shader on the GPU. It provides a flag for each grid block indicating whether it contains rendered pixels or not. For ROI selection, the reduced-size bitmask is transferred to the main memory. On the CPU a block-based region split is computed aligning the ROIs to the regular grid blocks. If enabled, the necessary ROIs will then be read back from the GPU for further compression, transmission and final compositing. Empty areas are detected using either a specified solid background color or by analyzing z-buffer values. 16×16 blocks are used for the occupancy mask, which provides a good trade-off between speed and precision.

The split algorithm is based on recursive region subdivision. Each iteration consists of two steps: finding the largest rectangular blank area (a hole); and best split determination based on hole position and size. Figure 5.3 illustrates the recursive per-block occupancy hole detection and split process. Depending on a hole position within a region, there are several possible split options. For this particular hole's configuration, there are two ways to obtain rectangular sub-areas that do not include the hole itself but only the rest of the image, one of these is shown in each subsequent step.

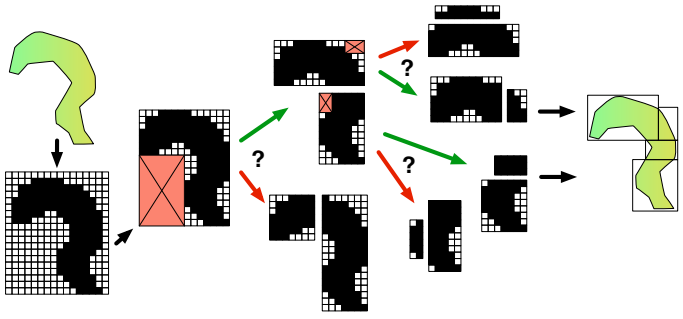


Figure 5.3: *Recursive largest hole detection and subtraction for recursive ROI selection.*

5.2.2 Empty Space Search

Identifying the largest unused rectangular region within the block-based occupancy mask is efficiently performed using a *summed area table* (SAT). The oc-

cupancy map is transformed into a SAT with entries indicating the number of empty blocks up to the given index. Thus emptiness of any rectangular region can quickly be verified by four SAT lookups and comparison to the block size of the region.

A maximal empty-area search algorithm is defined as a sequence of requests to the SAT, and executed in the following way over the block-based occupancy map SAT:

1. Search in scan-line order bottom-up for empty blocks.
2. In each step, find the *intermediate largest hole*, a rectangular hole that includes the current block and is bounded by the upper right corner of the map.
3. Update the current largest hole if the new one is bigger.

The *intermediate largest hole* search in Step 2 is based on a three-fold region growing strategy as shown in Figure 5.4. The strategy is to first grow an empty square region diagonally, followed by growing a tall empty rectangle vertically with subsequently reduced width on every empty-region test failure. The same is then performed analogously horizontally. In each growth step an empty-region test corresponds to a query of the SAT occupancy map.

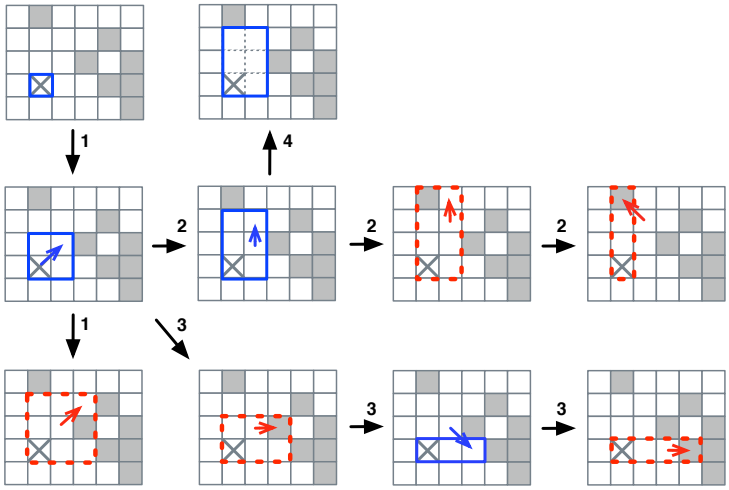


Figure 5.4: *Largest hole searching first proceeding diagonally upwards (1), then vertically with decreasing width (2) and last horizontally with decreasing height (3). Eventually the largest hole found (4) is reported.*

To avoid identifying too small empty areas, the hole search procedure is terminated if either a minimal absolute or relative empty region size threshold is reached. In that case a zero-sized hole is reported to avoid further recursion. These values can be set to quite high values (e.g., 200 blocks and 2%) and still give acceptable cuts.

5.2.3 Split Estimation

A region split is executed once after the largest hole in a current frame buffer region has been found. There are four categories of hole positions as shown in Figure 5.5, of which only the first one leads to a simple split into two new vertical or horizontal rectangular regions.

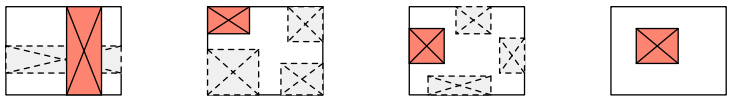


Figure 5.5: *Different categories of hole positions: (a) through, (b) corner, (c) side and (d) center.*

The symmetry classes of possible region splits are shown in Figure 5.6. For a corner hole only two variants are possible, with one either vertical or horizontal line aligned to one of the hole's edges. A side hole has four different options, and a center hole has two unique configurations as well as four, which reduce it to a side hole.

To find the best split, the algorithm maximizes the area that can be removed in the next subdivision. That is, a hole search is performed for the subregions of every possible split and the accumulated size of all holes is considered. For the best split the determined hole positions are forwarded to the recursive split processes for each subregion.

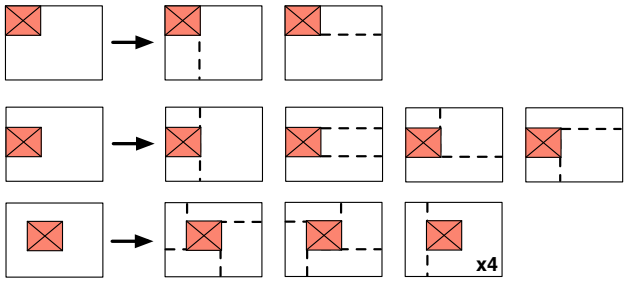


Figure 5.6: *Symmetry categories of region splits.*

In order to avoid excessive and repeated hole searches, information from common subregions is shared. There is a maximum of 16 different subregions that have to be checked depending on the hole position, as depicted in Figure 5.7. For corner and side holes, disappearing subregions are considered to have a zero hole area.

Figure 5.13 demonstrates examples of the ROI algorithm results. Rectangle areas are final regions after ROI algorithm is finished, where hashed areas are not

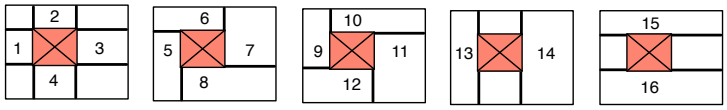


Figure 5.7: *Different subregions for split calculation.*

read back to the RAM at all. On the right-bottom image it could be seen that ROI can't reduce any space and therefore disabled; the whole window is read back in this case.

5.3 Performance Evaluation

Performance of the ROI algorithm was evaluated using the same setup and data sets that were described earlier in the Chapter 4. *David Head*, *David 1mm* and who different setups *fly-around* and *fly-through* for *Power Plant* model were used with eqPly, as well as *Turn* and *Zoom* paths for *32K×32K SRTM* terrain data rendered with eqRASTeR application. Additional *fly-over* setup with a smaller *Puget Sound* model for eqRASTeR is evaluated in order to demonstrate per-frame ROI performance in detail.

ROI compression According to the Chapter 4 it can be concluded that for sort-last rendering the efficient encoding of the sparse image data is important, i.e. removal of blank pixel data. Furthermore, the experiments on sort-first rendering have shown that for large non-uniform color regions YUV subsampling and optionally swizzle RLE coding achieve significant improvements in image throughput. In the following the influence of proposed ROI selection algorithm is evaluated. Serial compositing up to 5 nodes and afterwards Direct Send (DS) compositing is used as a reference point, when ROI is executed in serial fashion only. RLE compression used for simple tests, and RLE per-component for ROI tests (since blank areas are already excluded).

Figure 5.8 shows that the ROI algorithm clearly outperforms RLE for empty pixel removal. Despite the additional cost to detect ROIs (from 0.79ms to 2.5ms for 1280×1024) and splitting an image into multiple regions, the ROI selection method quickly and effectively identifies and removes blank frame buffer areas. Figure 5.9 further highlights advantages of the ROI method, featuring relative speedup that was achieved for different scenarios and data compression. In Figure 5.9(a) RLE and YUV compression methods are compared with and without ROI enabled, while Figure 5.9(b) demonstrates that due to combination of ROI and YUV subsampling significant speedups can be achieved over pure RLE method. The decrease of frame rates is due to the fundamental image throughput

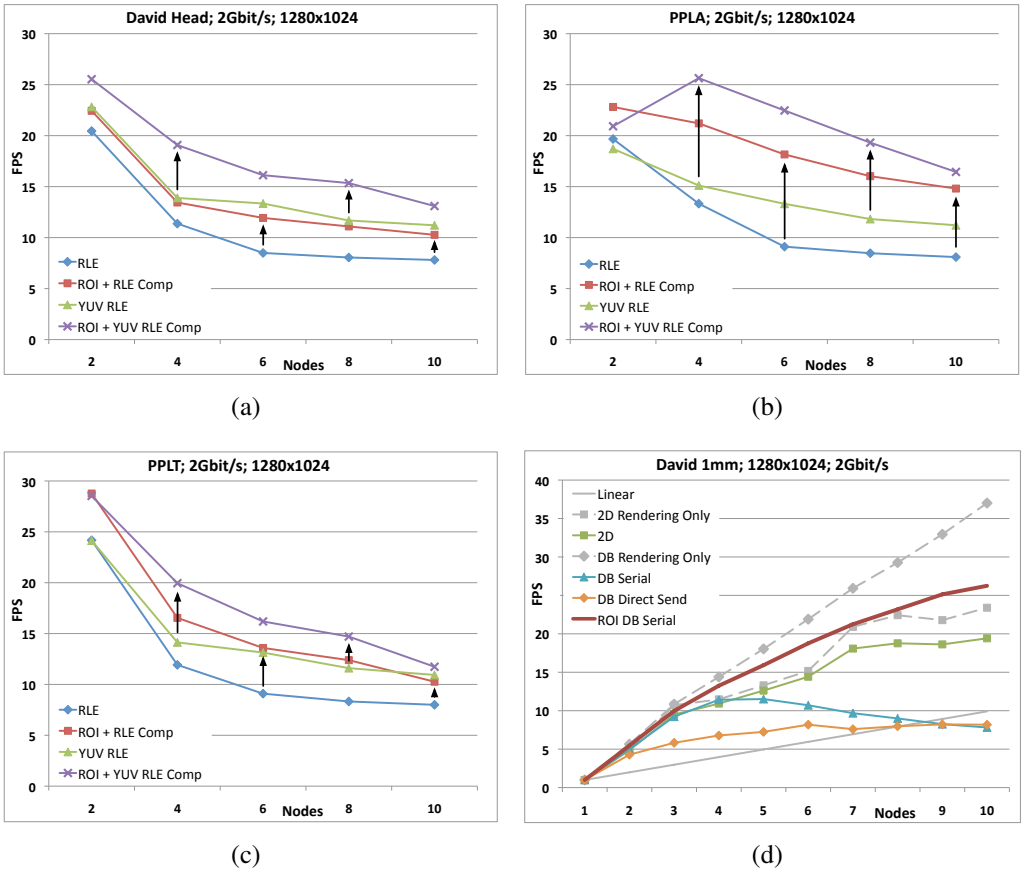
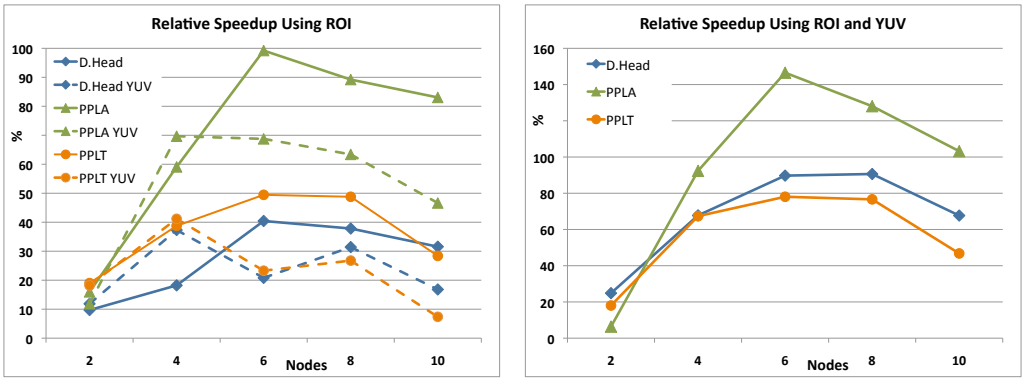


Figure 5.8: *Effect of the ROI method on image throughput: (a) David Head, (b) Power Plant (fly-around) and (c) Power Plant (fly-through) model; ROI selection impact on sort-first (2D) as well as sort-last (DB) serial and Direct Send compositing modes (d).*

limits outlined at earlier.

Even better improvement is achieved in eqRASTeR, when ROI is used. Figure 5.10 shows comparison of eqRASTeR performance with and without ROI enabled. It can be seen that for sort-first setup (2D ROI and 2D No ROI) the performance is the same. Since vertical screen tiles are fully covered with image data, ROI is not able to find any empty regions and therefore is automatically disabled (it is clear from the results that ROI doesn't introduce any overhead in this case). Further, sort-last rendering with serial and DS compositing methods without ROI detection (DB No ROI and DS No ROI) are compared to serial ROI-enabled compositing (DB ROI). Presented results show that, while overall sort-last performance is still limited at around 12 fps due to network saturation, ROI method significantly outperform non-ROI compositing.



(a) (b)

Figure 5.9: *Relative speedup of ROI algorithm compared to the same methods without ROI enabled (a); combined ROI + YUV speedup compared to RLE compression only (b).*

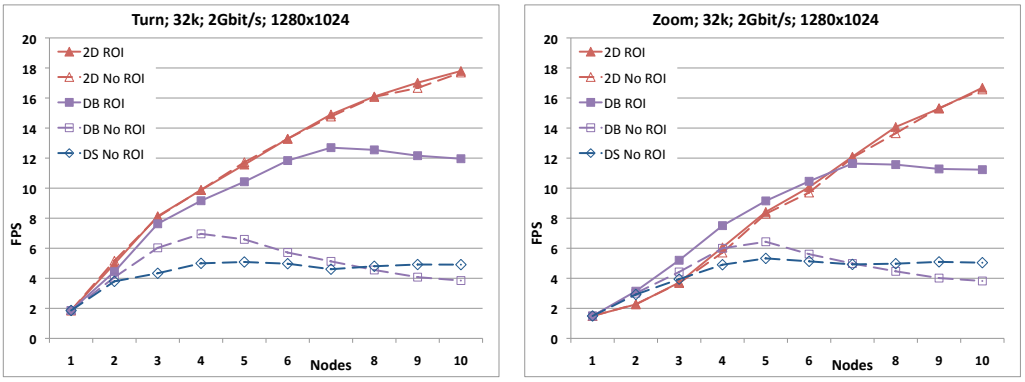
ROI scalability Without optimization of the image throughput, scalability is quickly limited due to the distributed image compositing stage, i.e. network transmission as reported in [Eilemann et al., 2009]. The positive impact of ROI selection on the compositing stage is demonstrated below. Experiments are conducted with the large *David 1mm* model displayed horizontally. For sort-first rendering the screen was divided into N equal vertical tiles, for sort-last the data is split into N equal chunks.

In Figure 5.8(d) the theoretically possible performance is indicated with Rendering only for sort-first (2D) and sort-last rendering (DB), without accounting for any image transmission and compositing. The actually achieved frame rates of sort-last rendering are shown for Serial and Direct Send compositing. Superlinear performance of rendering can be explained by caching effects in main memory and GPU. The negative performance impact of the limited image throughput is obvious as the speedup quickly approaches the expected frame rates and flattens out. In contrast, the ROI enhanced sort-last rendering keeps up scalability much longer and performs superlinearly up to the maximum number of nodes tested.

While sort-first rendering in general shows less impact due to image throughput limits than sort-last methods for large complex 3D data, ROI enhanced sort-last rendering nevertheless outperforms it considerably.

Per-frame ROI Performance It is clear that ROI optimization is more efficient when the occupied screen area is smaller, the following two graphs will show how much performance per-frame can be gained exactly.

Figure 5.11 illustrates per-frame timings of the serial 6-to-1 sort-last compositing. These results correspond to the data points for 6 nodes on the Figure 5.10(a),



(a) (b)

Figure 5.10: *Performance results of eqRASTeR application with and without ROI optimization enabled.*

where average frame rate of 5.7 fps and 11.8 fps is archived by DB No ROI and DB ROI accordingly (the small difference in the average timings compared to the Figure 5.11 is due to the time measurement error since these tests were performed at different time).

The amount of the in-place data, rendered by the destination window is shown below the graph (white digits in the lower-left corners represent number of pixels relative to the whole frame), where white space corresponds to the composited area (with black digits in the upper-right corners). Due to camera movement the ratio of in-place to composited pixels slowly changes, this result in slight increase in overall ROI performance (Linear(DB ROI)), since less data has to be processed, on the contrary, the performance of the non-optimized version stays the same (Linear(DB No ROI)).

To demonstrate the influence of ROI even better, another experiment with different camera path and smaller dataset was conducted. Puget Sound (PS) model and much faster camera motion were used. This experiment was evaluated using only 4 nodes in order to visualize corresponding data regions better. Figure 5.12 contains the findings.

The chart features per-frame performance of eqRASTeR with and without ROI method enabled (DB ROI (FPS) and DB No ROI (FPS)), as well as number of composited pixels (% of new pixels)). First row of the screenshots below the chart, displays data distribution between different renderers; colors red, green, blue and pink correspond to 4 different contributing pixel sources, where the red color is assigned to the data rendered on the destination channel itself (the numerical amounts of in-place information are additionally given in the lower-left corners and the amounts of composited information in the upper-right corners, similar to

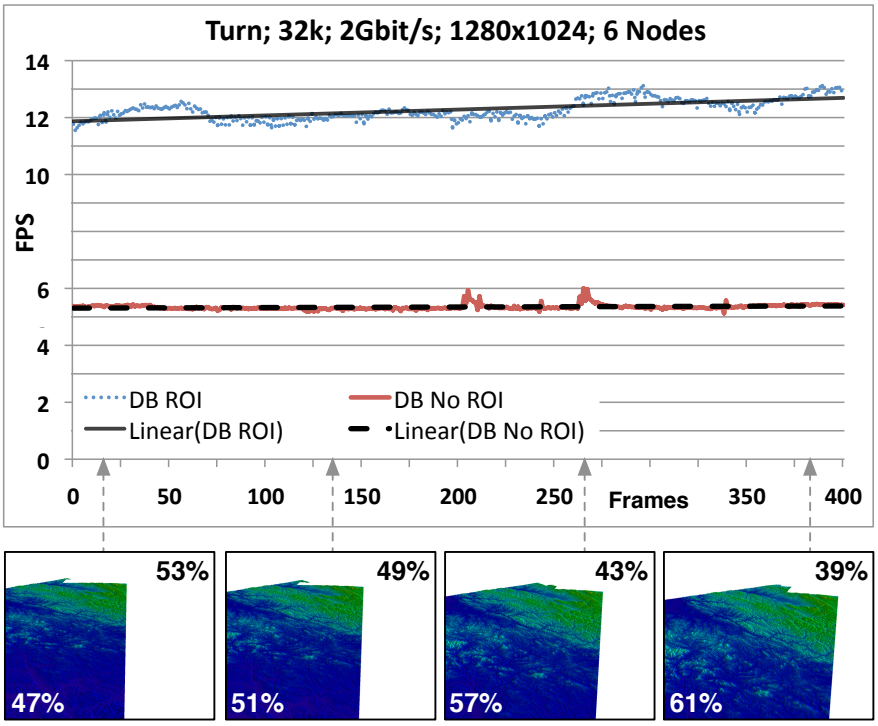


Figure 5.11: *Per-frame performance of ROI algorithm. The screenshots below the chart are featuring screen area space that was drown by the destination channel (white numbers), white space correspond to composited area, rendered by 5 other channels (black numbers).*

the Figure 5.11). Second row of the screenshots features complete frames as they appear on the destination viewport after compositing.

First, second and fourth screenshots correspond to, around, 70 percent of the in-place data where only 13 to 15 percent of the screen is read-back, sent over the network and finally composited, hence the high frame rate of 28 to 32 fps for these frames. The third screenshot correspond to a global maximum of 75 percent of the screen of composited information. The overall frame-drop appear with a small delay as expected, when the amount of composited information reduces again, the frame rate increases back to almost 30 fps. At the same time the performance of non-optimized version (DB No ROI (FPS)) is nearly a constant for the whole sequence.

It has to be noted, that during frames 150 to 300, large portions of the data being loaded by all renderers, due to significant change of the camera position, which lead to significant change in the rendering set selection and therefore significant changes in data-range assignments. Nevertheless the optimized version of the

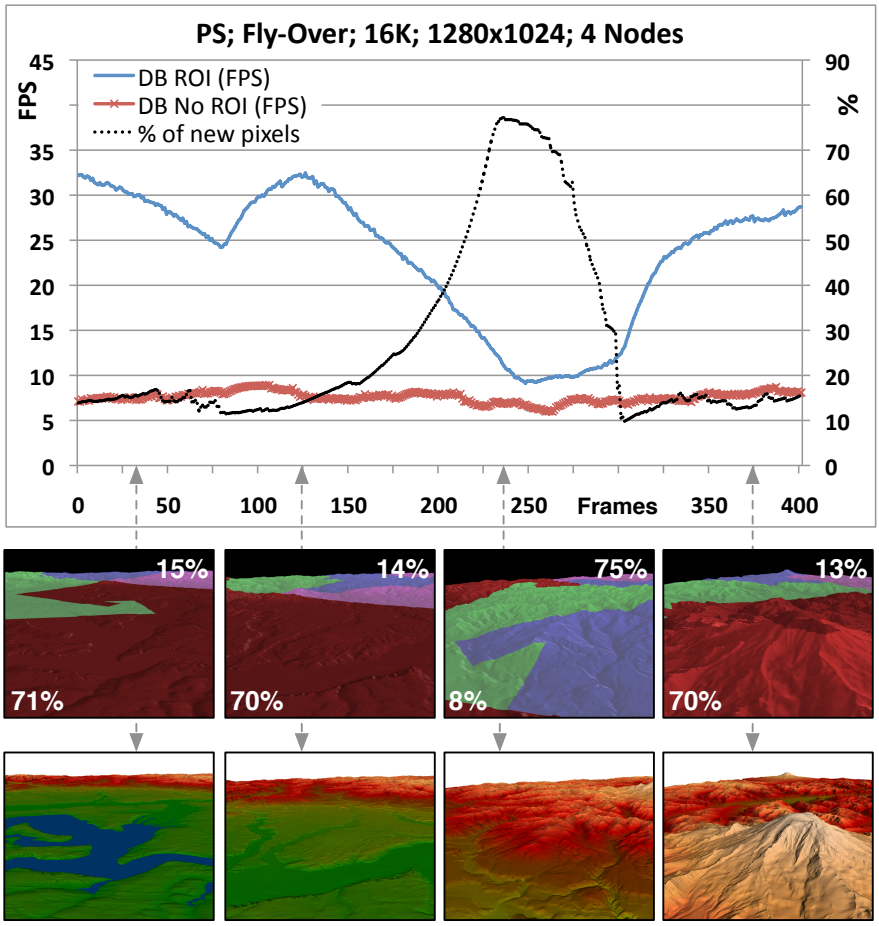


Figure 5.12: *Per-frame performance evaluation of a Puget Sound data set. Screenshots are featuring different data sources as well as final images after compositing.*

compositing algorithm outperforms non-optimized version at all times, providing up to 4 times frame rate increase in the best case.

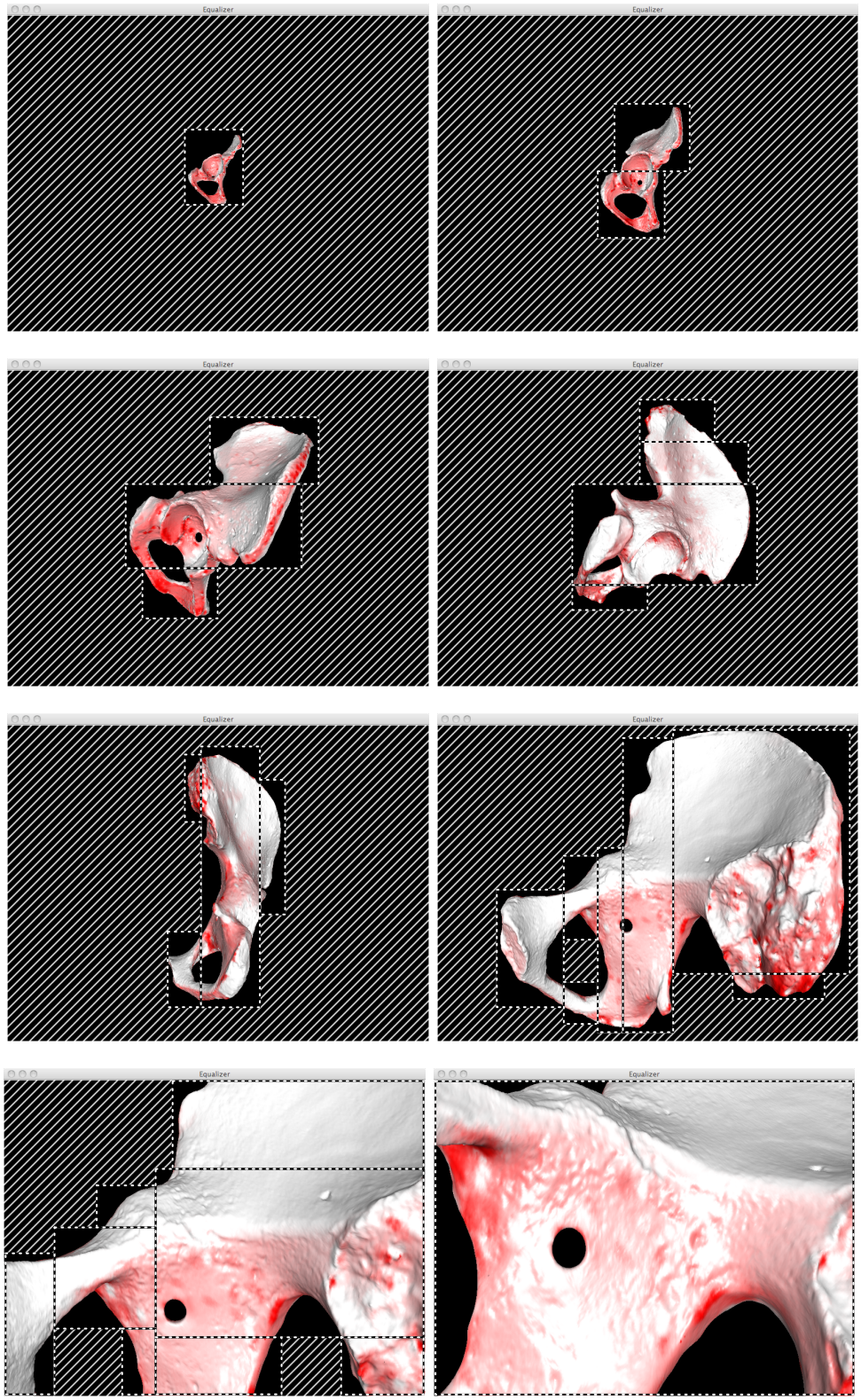


Figure 5.13: *Examples of results of the ROI algorithm (rectangular areas are read back separately, where hashed areas are not copied to RAM at all).*

C H A P T E R 6

DATA MANAGEMENT AND LARGE VOLUMES RENDERING SYSTEM

Equalizer parallel rendering framework in its current state, provides a great deal of compositing and synchronization support. Several methods for data synchronization and exchange are also available, however application developer is still forced to invest significant amount of time in creating of parallel out-of-core rendering solutions due to missing data management layer. Out-of-core methods exploit similar strategies that are widely used in general-purpose computation optimizations, namely two level cache, these methods should be introduced to the framework to make application development easier.

6.1 System Overview

Equalizer framework provides basic implementations of the entire parallel rendering application, where an application developer can extend default behavior if needed. Figure 6.1 (a) shows the most important classes of the framework that application developer has to deal with (they belong to the *eq::* namespace). *eq::Client* and *eq::Server* are used to run application and handle configuration tasks (parsing of configuration files by server, starting of the applications, and calling rendering, compositing and synchronization commands). Further classes represent resources of the rendering system: *eq::Node* correspond to a single computer; *eq::Pipe* is an abstraction layer for GPUs; *eq::Window* responsible

for windows; and, finally, *eq::Channel* represents viewports within a window. *eq::Pipe* and eq::Window provide necessary interfaces for multi-platform execution (Equalizer runs on Windows, Linux and Mac OS), hiding system specific code from the user. If new windowing system has to be introduced, mainly these two classes have to be adjusted.

Since this structure is mapped to actual resources, it has similar properties: one configuration can have many Nodes (computers), each node can have multiple Pipes (GPUs), where each GPU can host multiple Windows, and rendering would happen within several Channels (viewports) of each window.

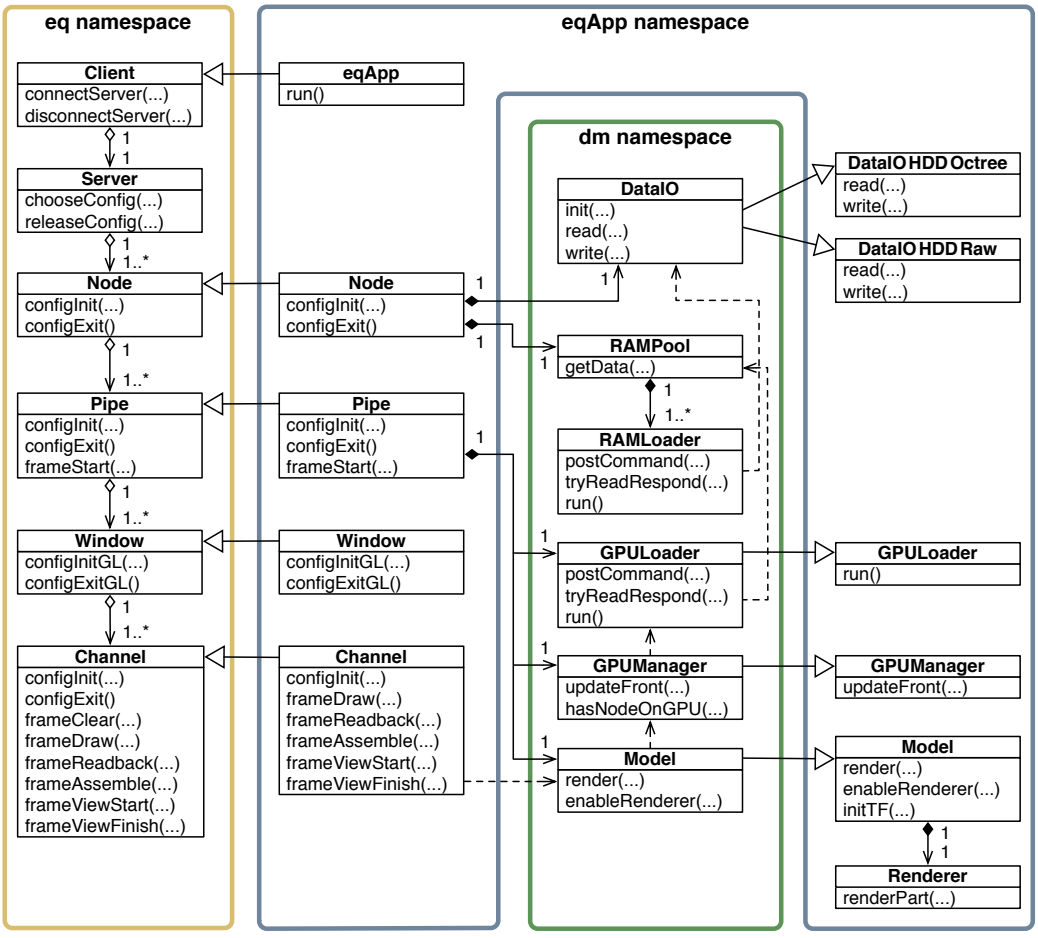


Figure 6.1: *Basic classes of sample applications (b) with corresponding Equalizer classes (a); data management functionality (c) and its application specific implementation (d).*

Creating of a parallel rendering application with Equalizer can be as easy as subclassing *eq::Channel* class and overriding its *frameDraw()* function only, the

default implementation of the framework will take care of the rest. This, however, allows very little flexibility in the way the data can be managed. If any type of data sharing is required, further overriding of build-in functionality is necessary. For example, in order to share data between windows of the same GPU, *eq::Pipe* class has to be subclassed and modified; when rendering parameters should be changed during OpenGL context initialization - *eq::Window* class and its *configInitGL()* function needs to be overridden; further, when some information has to be cached in the RAM and be accessible to multiple GPUs - *eq::Node* would be the right place to look at.

Generally, any default behavior of Equalizer can be extended or replaced with custom implementations; for instance, eVolve volume rendering example overrides compositing functionality of *eq::Channel* to achieve custom back-to-front α-blending based image compositing; eqRASTeR extends *eq::Pipe* and *eq::Window* with additional hidden window that shares context of current GPU in order to perform asynchronous data loading to this GPU. The according classes that most often are subclassed by applications are presented on the Figure 6.1 (b).

There is a number of additional classes (they are not described in detail here) that are used for data synchronization between nodes, and suppose to help with data management, while the generic caching and data to GPU uploading mechanisms are still missing. The presented, in the following, system is build upon standard Equalizer functionality and is designed to help with creation of distributed out-of-core rendering solutions.

6.2 Data Management

The focus of proposed system is on interactive large data visualization. When the whole data, even after distribution between nodes, can't be fitted into the RAM or VRAM, out-of-core methods are required. There are two basic strategies to render large data: split data into parts and render one part after another, until the whole data is rendered, or create level of detail structure and for each frame estimate appropriate levels of detail based on the camera position and the number of resources available. Although being more complicated, only the second method can provide interactive rendering performance, where time spent on a single frame is fixed, meaning that the amount of information rendered per frame has to be limited as well.

In the following it is assumed that the data is initially stored outside of the rendering cluster's RAM (it can be hard drives of the cluster, external NAS, or even memory of a separate supercomputer that runs a simulation, which has to be visualized), and that rendering is performed on a small GPU based cluster (recent reports [Suss et al., 2010; Howison et al., 2010; Makhinya et al., 2010] suggest

that this type of setup is the most suitable for interactive visualization in terms of overall rendering performance).

Naturally, the data, being visualized, has to appear in the VRAM; since it is common that amount of RAM greatly exceeds amount of VRAM, a two level cache can be constructed. An example of such caching system is presented on the Figure 6.2. Initially the data (*DATA*) is stored outside of RAM and VRAM (*RAM CACHE* and *GPU CACHE*); in the beginning of each frame, algorithm responsible for LOD computation (*Model*) evaluates parameters of the camera, available rendering time and VRAM size, and requests from the *GPU Manager* appropriate portions of information to be loaded to the VRAM. At this point of time *Model* doesn't know whether portions of the data were uploaded to the GPU already or not, it is task of the *GPU Manager* to figure out what data blocks is already on the GPU, which blocks are not, and what portion of information has to be replaced on the GPU in case there is not enough space. Once the missing blocks are determined, a list of requests is sent to the *GPU Loader* which is a separate process running in parallel. *GPU Loader* requests data blocks from the RAM manager (*RAM Pool*) and if the data is available in the RAM, *GPU Loader* uploads the data to the *GPU Cache*. In case when requested information is not in the RAM, *RAM Pool* sends loading requests to *RAM Loader* which is responsible for loading of the data from external source to the RAM and is implemented as a separate process that runs asynchronously as well.

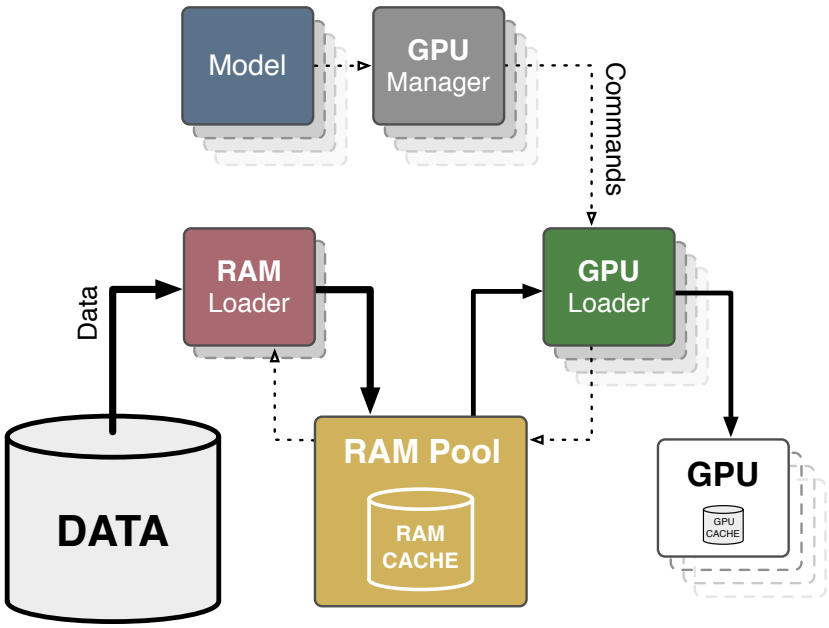


Figure 6.2: *Overview of data flow, and management.*

Since there could be multiple GPUs in the system it is necessary to have mul-

tiple *GPU loaders* each of which shares OpenGL context of one GPU, also the number of *GPU Managers* and *Models* has to be adjusted accordingly, to reduce overheads when rendering is done in parallel on multiple GPUs. The number of *RAM Loaders* can be arbitrary, depending on the external storage characteristics, since it make sense to maintain only single *RAM Pool* so that the data in the RAM could be shared between *GPU Loaders*.

The roles of different parts are summarized in the following:

Model

- Maintains hierarchical data structure representation;
- Estimates LOD selection;
- Renders data.

GPU Manager

- Maintains state of a single GPU data cache;
- Provides GPU's data accessibility information to the *Model*;
- Communicates with a single *GPU loader*.

RAM Pool

- Maintains state of the RAM data cache;
- Maintains queues for the data requested by *GPU Loader*s;
- Makes sure data is loaded only once if it is required by multiple GPUs.

RAM Loader

- Reads loading requests stored in the *RAM Pool*;
- Asynchronously fetches data from the Data storage to the *RAM Pool*.

GPU Loader

- Asynchronously fetches data from the *RAM Pool* to the GPU memory;
- Maintains queues for requested by *GPU Manager* data;
- Communicates with the *RAM Pool*.

The mapping of the proposed cache structure to the Equalizer's class hierarchy is presented on the Figure 6.1 (c). *dm::RAMPool* is initialized from the *eqApp::Node* class, since it has to be shared among GPUs; *dm::RAMPool* initializes as many *dm::RAMLoader*s as necessary; where *dm::RAMLoader*s are sharing *dm::DataIO* that is responsible for reading of particular data types and is also initialized once for each *eqApp::Node*. Each *eqApp::Node* creates as

many *eqApp::Pipes* as required by the configuration, consequently, creating one *dm::GPUManager*, *dm::GPULoader* and *dm::Model* per GPU.

In order to implement application specific behavior several classes have to be subclassed and extended, these classes are shown on Figure 6.1 (d). For example, *eqApp::DataIOHDDOctree* and *eqApp::DataIOHDDRaw* are implementing data read and write functionality for two different volume formats; *eqApp::GPULoader::run()* implements particular data format to GPU uploading; *eqApp::GPUManager* can be used to change GPU memory allocation and data uploading strategies; *eqApp::Model* contains application specific LOD selection functionality and managing different rendering techniques through *eqApp::Render*.

Figure 6.3 illustrates data loading and caching algorithms and related auxiliary data structures that are used to implement fully asynchronous *DATA* to *RAM CACHE* and *RAM CACHE* to *GPU CACHE* data loading. Current state of the *GPU CACHE* is maintained by the *GPU Manager* in the *GPU Info*, and the state of *RAM CACHE* is handled by *RAM Pool* with *RAM Info* structure accordingly. Asynchronous behavior is achieved by maintaining several queues for exchanging with commands, requests and replies, which are periodically checked by their owners. Two queues are maintained by *GPU Loader*: *GPU Loading Queue*, where *GPU Manager* writes loading requests to and *GPU Loader* reads and processes them, and *GPU Respond Queue*, through which *GPU Loader* informs *GPU Manager* about successful loadings to *GPU CACHE*. Based on the information received through *GPU Respond Queue*, *GPU Manager* updates *GPU Info* structure to maintain current status of VRAM.

Separate *RAM Loading Queue* is maintained by *RAM Pool* for each *GPU Loader*, this way request from each *GPU Manager* and *GPU Loader* can be handled independently. When a request for *DATA* to *RAM Cache* loading is coming from *GPU Loader*, it is associated with an appropriate queue based on the unique identifier of the *GPU Loader* or new queue is created on a very first request, this way it is possible, for example, to easily cancel requests of different GPUs without affecting performance of other rendering units running in parallel.

*RAM Loading Queue*s are checked periodically by *RAM Loader*s and available requests are handled. It is responsibility of the *RAM Pool* to provide appropriate memory slot, based on Least Recently Used (LRU) statistics, when new data is loaded from an external source, and to make sure that there is only one copy of every data block in the RAM. The uniqueness of the loaded memory blocks is important since multiple GPUs can request the identical information at the same time through different *RAM Loading Queue*s (or a request can be potentially duplicated in the same queue) and multiple *RAM Loaders* can try to execute those requests in parallel as well.

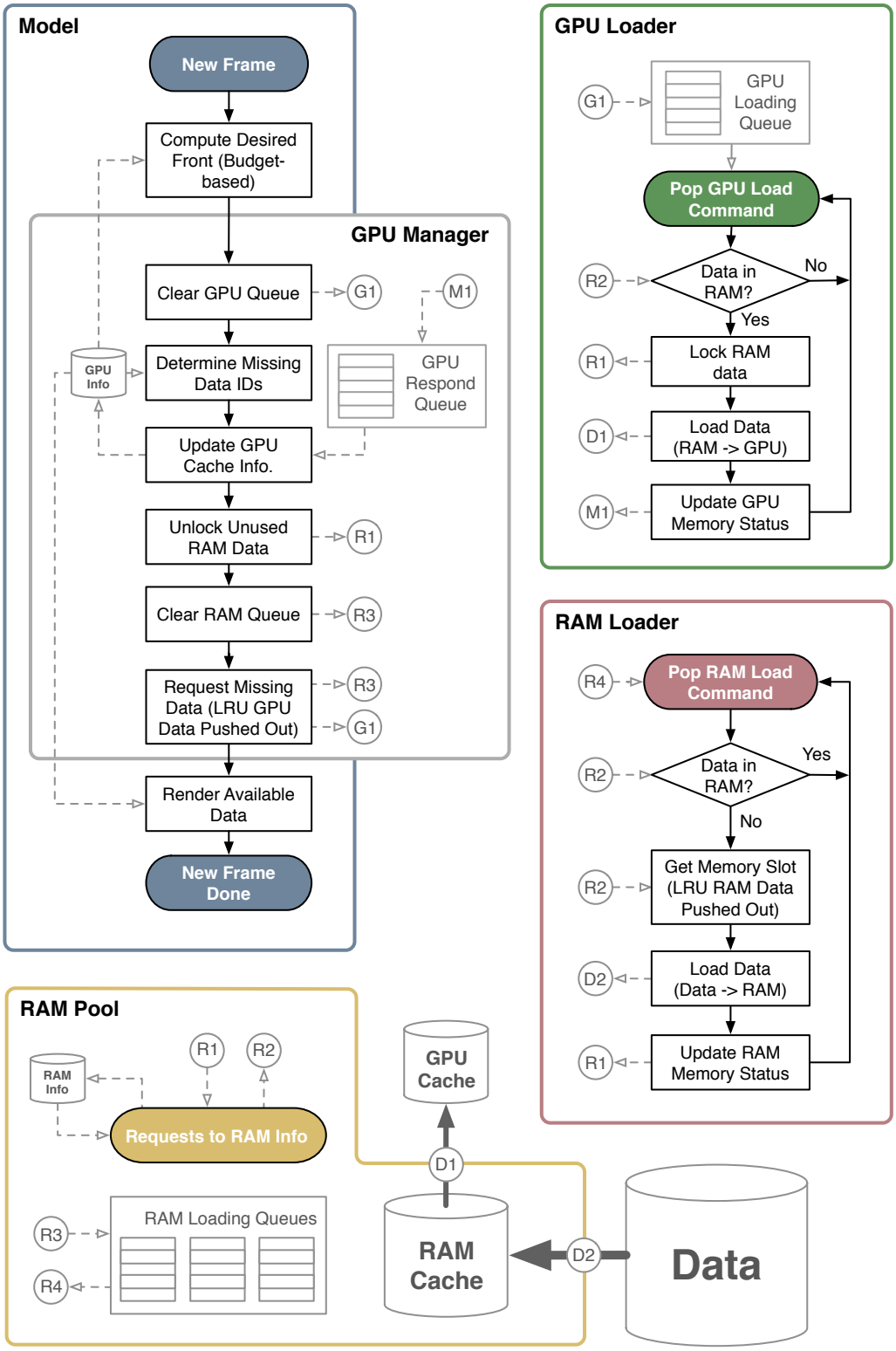


Figure 6.3: *Data management algorithms and communications.*

During rendering of a new frame *eqApp::Channel* (Figure 6.1) calls *dm::Model::render(...)* function, providing camera and viewport parameters, *Model* computes new set of data blocks, based on the data availability provided by *GPU Manager* through *GPU Info* (Figure 6.3), then calling *dm::GPUManager::updateFront(...)*, which updates *GPU Info*, computes and submits requests to *GPU Loader*, after which *Model* can proceed immediately with rendering of available data (*Render Available Data*) based on the updated information, while *GPU Loader* and *RAM Loader*s are fetching missing data blocks in the background. If viewport coherency is maintained there is a very high chance of previously requested data being needed again over several frames, therefore after possibly few frames of the delay, newly uploaded data can be used by the *Model*.

Although only data fetching from HDD is implemented at the moment, the system can be generalized to the shared memory setup (similarly to the method proposed by [Castanie et al., 2006]), through extending of the *RAM Loader*'s or data input (*dm::DataIO*) functionality. Shared memory is such case means that data blocks are shared between nodes, and when a data block was loaded from a slow source once, it can be copied to other computers much faster directly from node to node, without involving slow storage. This however assumes maintaining of a global data block availability cache or broadcasting data loading requests to other nodes, either way increasing intra-node communication.

6.3 Volume Rendering System

Interactive rendering of very large volume data sets become extremely popular demand nowadays. It is common to obtain tens and even hundreds of gigabytes of volume data from a single CT scan; these amounts of information can't be interactively visualized directly, and usually require offline preprocessing and out-of-core rendering techniques. In order to demonstrate functionality and apply it to solving of a real world problems, the proposed data management system was used to build an out-of-core parallel volume visualization application.

6.3.1 Data Import

Medical data is usually produced as a set of independent scans, each residing in its own file (typically in a DICOM file format). This set of files has to be converted into an appropriate format for efficient out-of-core visualization, which means that a set of levels of detail has to be created for faster data loading and interactive visualization.

Currently, a data set of $2600 \times 1600 \times 5196$ voxels, occupying about $42\ GB$ of disk space was tested with the proposed system. Figure 6.4 illustrates the volume

data processing chain. A set of DICOM files was converted to the RAW data representation during the first step. RAW data is then aligned with a regular grid and split into smaller cubes (64^3), this creates a lowest level of the octree (*Level N* on the Figure 6.4), where each block is numbered and can be loaded independently by the data management system. The final size of this lowest level is chosen in such a way that the number of cubes in each direction is equal to the next largest power of 2 number and, in general case, final size is larger than the size of original data, however empty blocks are only used in the further data reduction step and are not stored on the HDD.

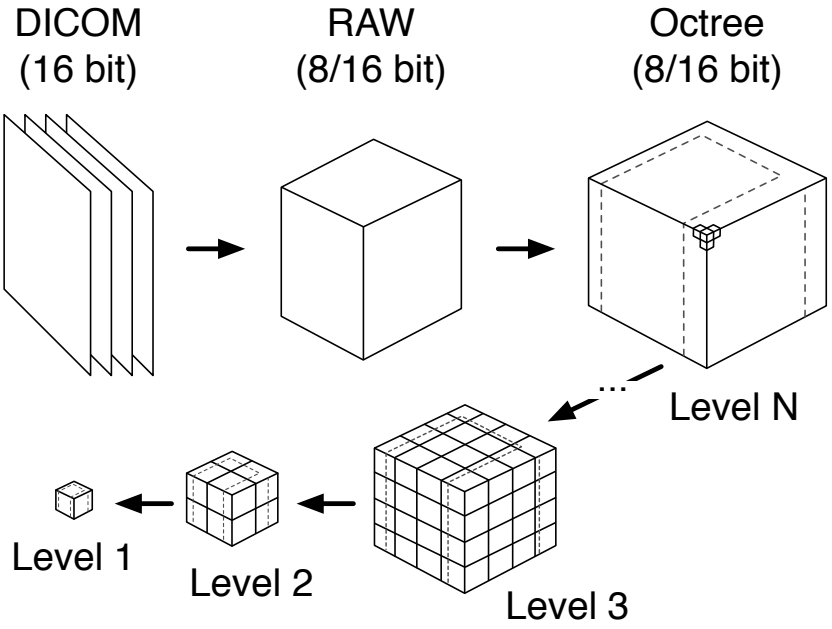


Figure 6.4: *Volume data preprocessing, where original DICOM image files are converted into an octree representation for efficient loading and LOD-based visualization.*

On every next step a higher level of the octree is created by averaging every eight neighboring blocks of a previous level. Since the number of blocks in each direction is equal and is a power of two number, in the end such data reduction produces a single block (*Level 1*), which is a rough representation of the whole dataset and is associated with a root of the octree data structure.

The whole process of data reduction is additionally visualized on the Figure 6.5, using real data. Initially a set of DICOM files can be viewed slice by slice only: Figure 6.5 (a) demonstrates one of the 5196 slices of 2600×1600 px resolution, painted with pseudo-colors; then separate blocks of the original data are extracted: Figures 6.5 (c) and (d) correspond to two blocks of the lowest octree level, and contain original data before reduction. Finally after the octree hierarchy

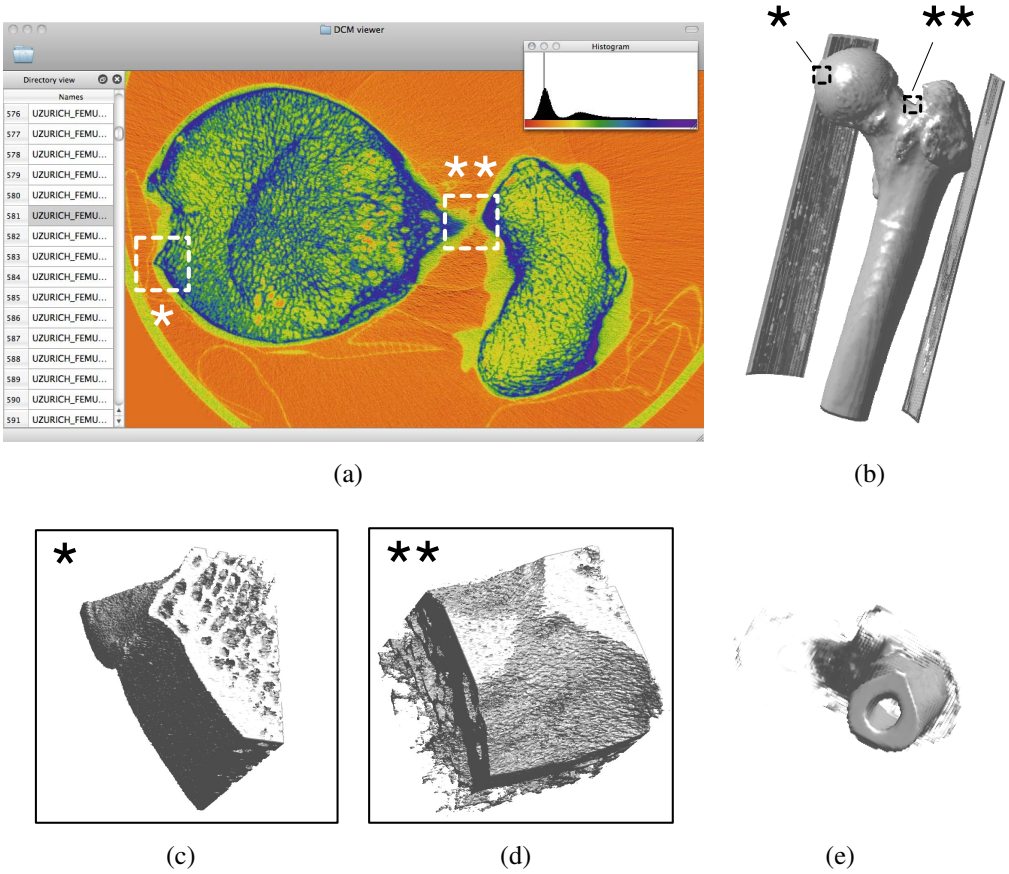


Figure 6.5: *DICOM datasets viewer (a) featuring one scanned slice of 2600×1600px resolution, with the renderings of two corresponding data blocks (c, d), and renderings of full data-set with two different data values interpretations (different transfer-functions) (b, e). Additionally, characters "*" and "**" are showing locations of corresponding data blocks (c) and (d) in the original DICOM data (a) and full data-set (b).*

is created, a top level block can be used to visualize the data on the coarsest level of detail: Figures 6.5 (b) and (e) represent rendering of the same *Level 1* block using two different interpretations of the volume values (two different transfer functions).

6.3.2 Rendering

Once the data was transformed into an octree representation, it is possible to visualize any part of the volume with any available quality, including original values, by rendering individual blocks in the correct order. The octree has to be traversed

according to camera parameters and appropriate blocks have to be scheduled for loading and rendering afterwards.

The main idea of brick-based volume rendering, presented here, is similar to the one used in eVolve: blocks have to be rendered in back to front or front to back order, the main differences from the approach used in eVolve is that the octree is split in three dimensions, rather than in one and that neighboring blocks can have different scaling factors, depending on the tree level. Once the order is established, each block can be rendered using the same rendering algorithm, that was used in eVolve, called for each block sequentially.

GPU memory handling Numerous blocks have to be cached in the VRAM and rendered during each frame, therefore efficient GPU memory allocation strategy has to be used. In the presented application, a single 3D texture memory cache is created. Multiple blocks are stored within this single texture simultaneously; when the system runs out of memory, some parts of this texture are overwritten with new blocks of data, therefore VRAM memory management is greatly simplified since data unloading doesn't have to do anything at all, and only one OpenGL texture is used during the whole rendering process. The *GPU Manager* is dealing with the mapping of data blocks in the *GPU CACHE*, providing necessary information about blocs' location to the *Model* during rendering.

Octree traversal and compositing For each frame octree is traversed by the *Model* twice. First time when it computes what is a desired data to render, before passing these requests to the *GPU Manager*, and the second time during actual rendering of available blocks.

During the first pass *Model* us memory budget, performance budget and quality requirements to evaluate which blocks have to be rendered, based on this information octree traversal order and levels of detail are established.

At each traversal step it is decided whether child link of the octree has to be followed or not, which means all children of the current node will be rendered instead of this node. The traversal continues if the following conditions are met: there is enough time to render all children, desired levels of detail for current node is not yet reached, and all children are already in the VRAM. In case when there is enough time, but children are not available for immediate rendering, *Model* forms a request to the *GPU Manager* for loading of the missing data. The traversal also makes sure that blocks, which are in front are traversed first and have at least the same quality with those, which are behind.

The traversal starts at the highest tree level, consequently deciding whether further traversal has to be done. At the each node that has to be traversed, an order of child blocks is established based on the normals associated with the octree and a

vector that connects center of the camera and center of the block that is associated with currently traversed node.

The process of establishing visibility and traversal order is illustrated on the Figure 6.6. Figure 6.6 (a) represents two levels of hierarchy traversal, where corresponding to this octree structure blocks of data are shown on the Figure 6.6 (b); the camera position is denoted as O and centers of the whole hierarchy and divided parent as O_0 and O_1 accordingly.

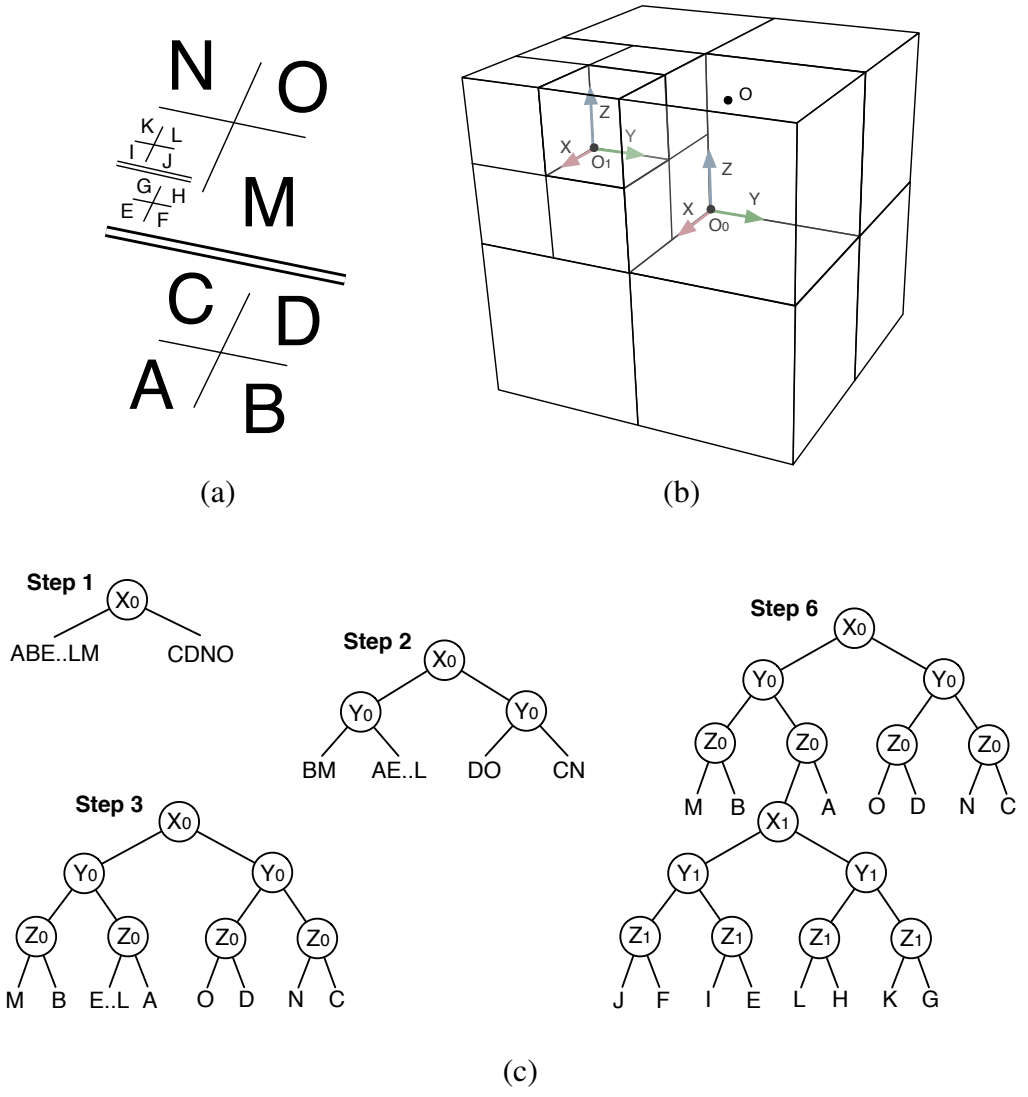


Figure 6.6: *Octree volume hierarchy (a) with corresponding volume data blocks (b) and traversal sequence (c).*

There are seven parent nodes named *A*, *B*, *C*, *D*, *M*, *N*, *O*, and eight children

on the eight's parent *E..L*. The traversal starts from the top level, where an angle between OO_0 and X vectors is analyzed, based on the result the volume is split into two parts by the plane O_0YZ, where blocks on either side are considered to be in front or in the back, Figure 6.6 (c) (*Step 1*) shows this split after the first iteration. During the second step an angle between the vectors OO_0 and Y is checked, depending on the result a plane O_0XZ divides each leaf into two parts again (Figure 6.6 (c), *Step 2*). In case of front to back traversal, after this step, it is possible to say already that nodes B and M are in front, followed by nodes $AE..L$, followed by D and O, and finally concluding with C and N. After the third step, where angle between vectors OO_0 and Z is computed, the top level of the hierarchy is fully evaluated; next, the procedure have to be repeated for children nodes, using vectors OO_1 and X, Y and Z accordingly. In total, for the given example, after six comparisons of the appropriate angles, it is possible to build the whole visibility tree hierarchy; traversing this tree from left to right, counting only leafs, gives front to back order of the blocks: $MBJFIELHKGAODNC$, and from right to left gives the back to front compositing sequence.

Levels of detail selection The levels of detail are selected based on time, memory space and quality requirements. The number of blocks to render depends on the average time spent during previous frame on all rendered blocks. This way it is possible to limit number of blocks in order to achieve required frame rate. Number of blocks is additionally limited by the available GPU memory, however on modern GPUs the amount of memory is usually a less significant constrain comparing to rendering time. Finally, the quality aspect is taken into account, by projecting of the bounding sphere of each block on to the screen and comparing the resolution of the block to the size of the projection. In case of perspective projection the size of the bounding sphere in the screen coordinates will depend on the distance of this sphere to the screen, therefore further away blocks can be visualized using coarser octree level, while maintaining the same per-pixel rasterization quality.

Two examples of such octree traversal and LOD selection are presented on the Figure 6.7. Depending on the camera position two different set of blocks are selected based on the distance to the screen and projected area.

6.3.3 Graphical User Interface

At the moment Equalizer doesn't provide any user interface out of the box, except for creating OpenGL windows, which limits the GUI experience significantly. Event though it is possible to extend Equalizer to support any windowing system (as it is already done for supporting GLUT-based windowing on various operating systems), integrating of advanced windowing toolkits, such as QT currently has

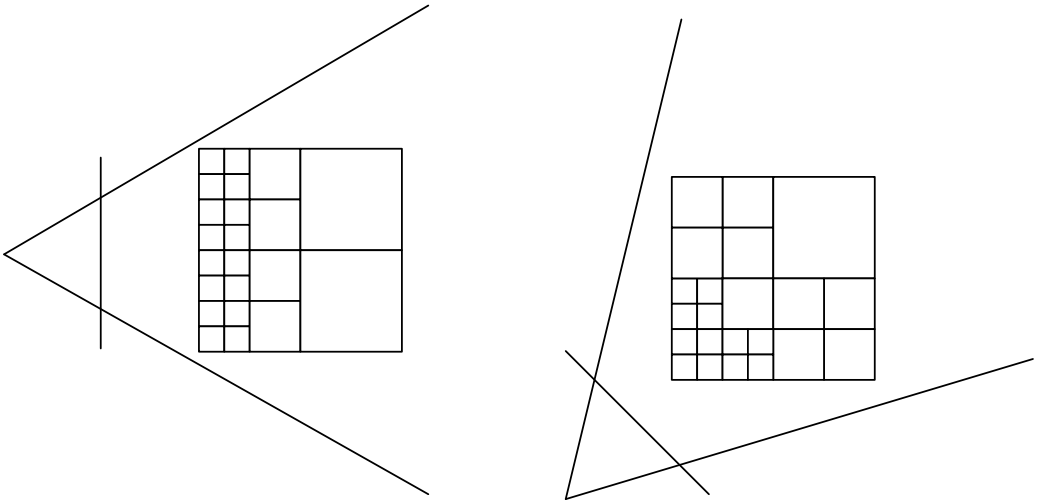


Figure 6.7: *Two different levels of detail selection of the same octree from two different points of view. Depending on the camera position LOD selection can significantly vary, which requires new data to be constantly fetched in the background for every significant change of the viewing parameters.*

some issues. Equalizer is already managing rendering loop, events handling and OpenGL context sharing, while QT framework has its own events processing loop, and straightforward integration causes overheads and OpengGL context conflicts. Solving these problems would require structural changes to the main rendering loop of Equalizer. Hopefully the framework will be extended to fully support QT soon, in the meantime less invasive approach to complete existing applications with a GUI was implemented.

A separate GUI application based on the EQ networking layer is created. It provides GUI features for file selection and TF editing for the volume rendering application by connecting to a running Equalizer process and exchanging user commands. This approach require minimum changes to existing applications and this type of GUI can be run from a separate device and be connected to any application remotely. Integration of existing GUI widgets is reasonably simple, Figure 6.8 features main dialog, connection widget and TF editing widget[1] of the volume renderer.

Two examples of interactive TF editing results are presented on the Figure 6.9, where two different values interpretation result in two different rendering outcomes for the same dataset.

[1]TF editing widget is a part of IVS volume visualization system developed at the Visualization and Multimedia Lab of the University of Zurich by Philipp Schlegel.

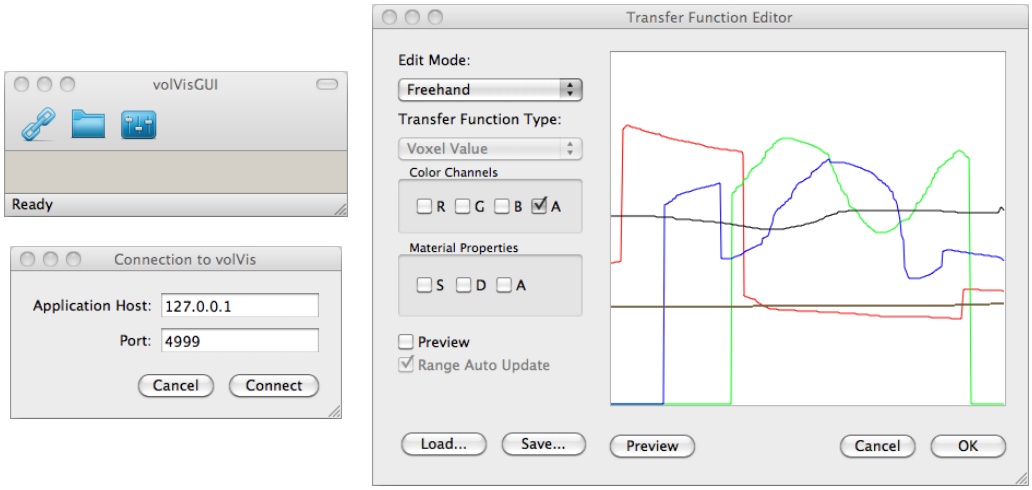


Figure 6.8: *Screenshots of the integrated Transfer Function editor. Provided GUI allows connection to a running Equalizer application and modifying of rendering parameters in real-time.*

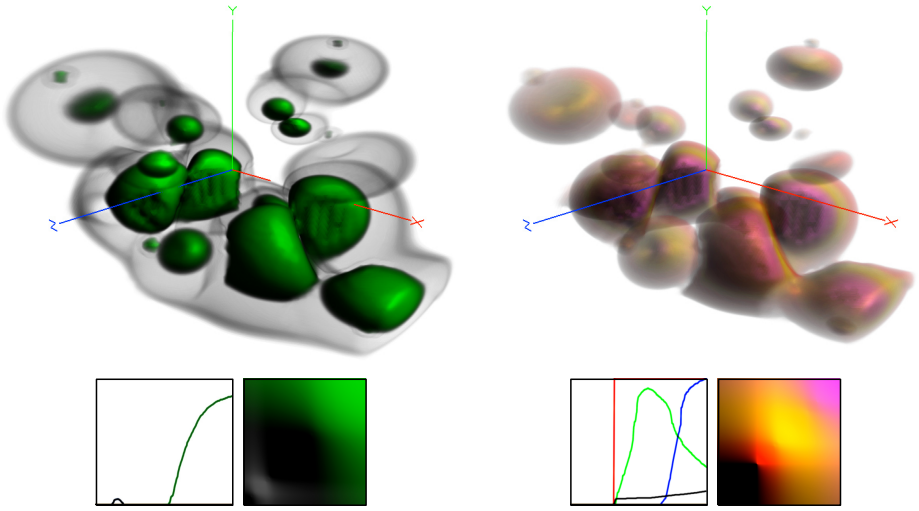


Figure 6.9: *Editing of rendering parameters in real-time is important for interactive data exploration. These two examples of different values interpretations illustrate two different visualization results of the same data. Boxes below the images show graphical representation of mapped colors in the GUI editor and corresponding textures on the GPU used for coloring the data.*

6.3.4 Results

Data Processing Proposed out-of-core volume visualization application operates on equally sized blocks of the octree hierarchy, and requires that the input data is preprocessed and individual blocks are stored as continues portions of data within a file for efficient loading. The initial DICOM images are converted to a 8bit raw format resulting in a single $21\,GB$ file, which is then used to create blocks for the octree. Octree data preprocessing is performed according to the following stages:

1. Storage space is allocated for the entire octree hierarchy;
2. Original data blocks that correspond to the *Level N* on the Figure 6.4 are then extracted and saved as continues portions of data;
3. Using blocks obtained on the step $2.$ the children nodes of the *N-1 Level* are created;
4. Reduction is repeated until *Level 1* (root of the octree hierarchy) is not obtained.

Performance results of the data preprocessing are demonstrated on the Figure 6.10, where time is compared to simple copying of data of the same size. As could be seen from this graph, preprocessing doesn't introduce a significant overhead, taking also into account that it has to be done only once before the actual rendering.

Data Loading The performance of the data fetching is only limited by the data storage reading performance itself. Using local HDD as the data source, proposed system was able to achieve around $140\ MB/s$ in a test where $250\ MB$ of data (1000 blocks of $256\ KB$ each) was loaded to the GPU, which is close to the limits of the HDD reading speed.

In the second experiment the same data was located on a remote storage device with slow network connection. This test was performed in the following way: available GPU memory was restricted to $250\,MB$ and two portions of 1000 blocks of $256\ KB$ each was uploaded to the GPU sequentially. Since only 1000 blocks would fit to the GPU, the entire VRAM was rewritten twice, every time with the new data; it took around 38 seconds to deliver this data to the GPU cache from the slow data source. The test was then executed again, this time both portions of the data were uploaded to the GPU twice in the following sequence: first 1000 of blocks; second 1000 of blocks; first 1000 again; second 1000 for the second time. In this scenario the content of the VRAM was replaced four times. Without RAM cache such data access would result in doubling of the loading time, however it took only 39 seconds for the proposed system to fetch the whole sequence, since

fast RAM cache was used instead of the original storage whenever a block was requested for the second time.

These two experiments confirm that the management layer doesn't introduce any significant overhead and can be efficiently used for caching with slower data sources.

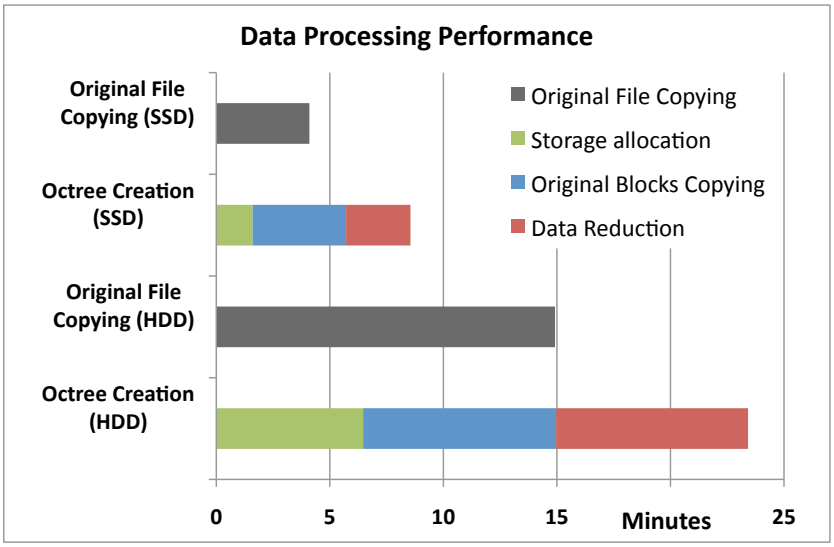


Figure 6.10: *Time required for full octree creation is compared to the time required to copy the original 21GB file. Data processing performance mostly depends on the storage performance, which is demonstrated by using two different data sources: ordinary HDD and fast SSD.*

Rendering Performance Out-of-core rendering has an advantage over in-core, single model rendering in case when more data than fits to the VRAM or even RAM has to be visualized, however the algorithm usually has to work with more rendering primitives since the data is rendered in small portions. More CPU resources are used to estimate the relationships between data parts, order of rendering and parameters of each rendered primitive.

The impact of using multiple bricks to visualize a single model was studies for the proposed volume rendering application in the following way: a 512^3 model that fits entirely on the GPU was rendered using a single 512^3 cube of data for 1000 frames; same model was converted into an octree, consisting of 512 children nodes (585 nodes in total) of 64^3 brick each, and all children nodes were rendered for 1000 frames as well, using the same quality settings. The average frame rate for single model resulted in 8.02 fps, where for bricked data it was 6.74 fps. The overhead is noticeable here since LOD was turned off and no parent nodes were allowed, resulting in rendering of all 512 blocks; the correct sequence of

rendering had to be estimated and extra computation for setting correct per-block rasterization parameters had to be done.

In the second part of the experiment LOD was switched on, and tuned to provide similar visual quality, but allowing parent nodes to be used, resulting in 288 blocks of 64^3 rendered per frame on average and running at 9.44 fps.

Performed tests show how flexible rendering using hierarchical data representations can be. In the provided examples the time spent on rendering was not taken into account by LOD selection and only quality being a limiting factor, however in interactive visualization minimum frame rate restriction is a very important parameter, which could be adjusted by LOD selection on multiple bricks with better granularity and in more flexible ways that can be achieved without hierarchical data representation.

Visual Results Figure 6.11 illustrates final volume visualization. Red wire-like boxes outline data blocks used for rendering. Depending on the resolution and rendering budget different number of blocks from different octree levels is selected, resulting in different quality to performance ratio. Rendering speed can additionally vary depending on the rasterization quality of each block, and not studied in the detail here, more importantly, proposed asynchronous data loading doesn't introduce any overhead to the rendering itself, where initial data fetching only depends on the data storage performance.

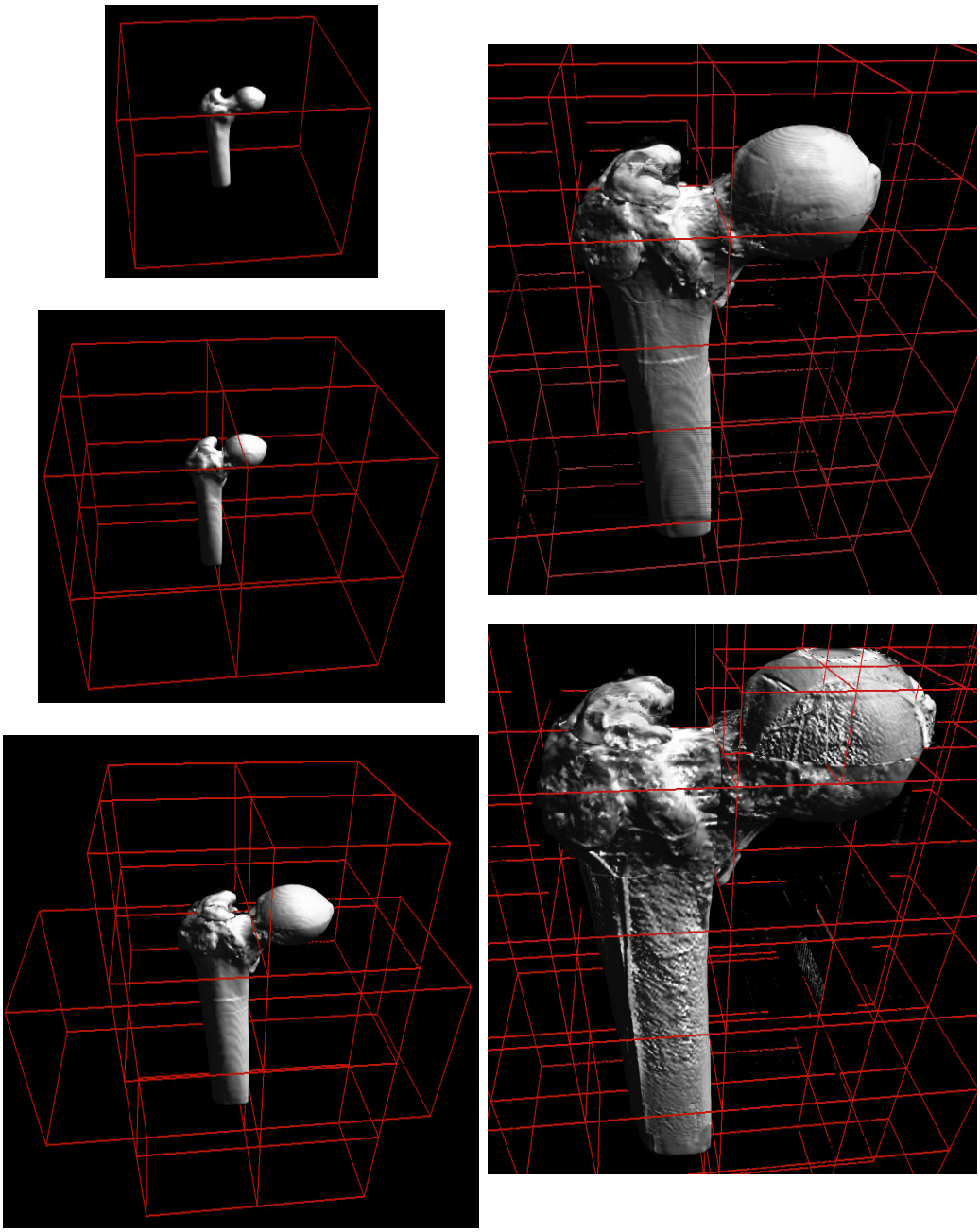


Figure 6.11: *Examples of different LOD selection depending on resolution and rendering budget requirements. Wire-like boxes outline individually rendered data blocks of the octree.*

CHAPTER 7

CONCLUSIONS

7.1 Summary

The initial porting of a stand-alone application to a general-purpose parallel rendering framework is relatively straightforward. In particular, sort-first and display wall types of rendering setups can be achieved easily. The algorithm has to be only extended in order to include the new view-frustum matrices and the windowing system. The sort-last decomposition, however, requires rather significant changes to the initial rendering method. Efficient data range mapping is one specific difficulty. Data synchronization across nodes and, in some cases, correct compositing require additional effort from the application developer. Overall, there are still too many aspects of parallel rendering system that have to be taken into account. A good trade-off between performance and required programming effort on the application side is an open challenge.

Compositing stage is an important part of any parallel rendering system, where low network performance is usually the weakest link of the chain. Selection of a proper compositing algorithm and compression method is essential for the overall performance improvement. Lossy data compression has to be used with a great care, especially in sort-last scenarios, where the boundaries between different parts of the model composited from different sources can have arbitrary shape (not only vertical or horizontal as in sort-first) and therefore reveal cracks in the compositing easily. Moreover, compression methods have to be extremely fast, preferably implemented on the GPU, such that compression and decompression

overheads can be easily covered by gains in transmission performance.

While it is possible to improve compositing speed, even without prior knowledge about the rendering algorithms, few constrains have to be considered by both, the rendering system and the application developers; for example having compact screen regions combined with clever ROI technique can improve compositing performance considerably. This idea mostly affects the data distribution and rendering pattern that has to be taken into account by the application developer.

The evaluation of parallel terrain renderer showed that simple measuring of time spent on a frame is not sufficient for assessing workload, since it doesn't exploit underlying data caching and asynchronous loading mechanisms of the application. This type of balancing can only be used if all the data could fit to the GPU memory or be uploaded to GPU very quickly and is available without any noticeable delay. It limits the application to small data sets, where algorithm can benefit from rendering in parallel only if it has high pixel rasterization cost (e.g., ray tracing).

For an out-of-core application the best option for better scaling, at the moment, is to handle load balancing independently; this can be achieved by enabling static data split in the rendering framework and performing a number of rendered primitives equalization between nodes, based on a per-frame LOD selection for the whole data (Figure 4.3). This approach however assumes that all parameters of the rendering resources are similar, this is required for equal data fetching, caching and rendering performance; in heterogeneous systems proposed approach would have to be corrected, taking into account rendering power of each resource.

Equalizer provides very flexible configuration and parallel execution of the application; accessible abstractions for nodes, GPUs, windows and channels are flexible enough, and several options for handling distributed objects are also available. This however is not sufficient, other building blocks like proposed asynchronous data fetching and better GUI integration are essential for rapid application development.

7.2 Directions for Future Work

Several fundamental issues of parallel rendering on a small visualization cluster were addressed in this thesis. There are still many problems to solve, due to the broad nature of the subject, these challenges are summarized in the following:

- *Efficient automatic load balancing*

  At the moment application developer has to adapt data distribution strategy manually for better performance and LB; many aspects of what parallel

rendering system is doing has to be taken into account as well (for example minimizing changed screen regions), since direct approach for LB through measuring only rendering time fails. Thus new strategies for automatic LB are required, where parallel rendering exploits some better form of a feedback from and to the renderer, which contradicts the idea that tight coupling of parallel system and rendering algorithm is not desired since it reduces reusability of both parts.

Nevertheless, some general clues can be exchanged between rendering system and the application, for example some form of high level data caching map. In case of sort-last rendering, additional hints can be distributed between renderers for better LOD selection, for example relationships between the data and the occupied screen space can be further used to improve data redistribution coherency.

- *Image compression*

  Further performance improvements can be achieved if more sophisticated compression techniques would be used, for instance, methods that exploit not only spatial, but also temporal image redundancy such as video codec based compression. The challenge is to obtain very fast compression, and to adapt compression to dynamically changing viewport. In order to improve performance, motion compensation information could be extracted from the camera and objects' movements. Additionally, video compressor has to support transparency in case of sort-last rendering.

  The framework could also detect currently executed scenario and perform selection of the compression method automatically, rather than forcing user to pick appropriate compression beforehand.

- *Parallel volume rendering system*

  Parallel volume rendering is a large field for research of its own, and further possible improvements of the proposed algorithm are manifold. Volume compression is one option to further improve storage requirements and data fetching performance. Data fetching itself could be done from other rendering nodes in addition to the disk storage (this requires either data request broadcast, or, data occupancy map distribution). Both compression and fetching from different nodes can be easily integrated into proposed data management system.

  Further, the border between levels of detail change can be improved by taking into account this information and performing appropriate interpolation correction. Better LOD selection strategy based on the occlusion approximation is another topic.

Finally, global volume illumination that would span across multiple nodes is still one of the unresolved problems in general.

- *Data management system*

  Currently only frame rate based data management strategy is implemented, since it is the only useful strategy for achieving interactive frame rates; however, for static scenes where no user interaction is performed, more data can be rendered at slower speed. This type of load estimation and balancing, where system would render more data than fits to VRAM, can also be integrated. Such behavior usually causes memory trashing since the amount of data exceeds the amount of cache, and a better asynchronous loading strategy is required.

  So far the developed data management was only applied to the volume rendering application. It has to be sufficiently simple to integrate it with any existing approach like point-based or polygonal rendering. Besides the rendering functionality itself and data fetching routines, a level of detail mechanism has to be available for a better out-of-core performance.

- *Automatic configuration generation*

  Creating the best scalable configuration for arbitrary renderers and hardware is something that was not discussed in this work at all. So far all of the configurations were generated for a specific data decomposition and for certain number of resources. It would be useful to handle automatic configuration of an arbitrary number of resources and arbitrary renderers.

- *Native GUI integration*

  Equalizer supports various operating systems, providing an abstraction of the windowing functionality. This windowing system has very limited GUI support. QT would be a good option to replace existing implementation, however fundamental changes to the framework are required. Current difficulties lie in the combining of the execution loops of QT and Equalizer, where each framework has its own fixed order of execution, and straightforward implementation leads to OpenGL context conflicts and performance penalties. Therefore processing loop of Equalizer has to be substantially changed in order to be more service-oriented.

BIBLIOGRAPHY

[Ahrens and Painter, 1998] Ahrens, J. and Painter, J. (1998). Efficient sort-last rendering using compression-based image compositing. In *Proceedings Eurographics Workshop on Parallel Graphics and Visualization*.

[Allard et al., 2002] Allard, J., Gouranton, V., Lecointre, L., Melin, E., and Raffin, B. (2002). Netjuggler: Running VR Juggler with multiple displays on a commodity component cluster. In *Proceeding IEEE Virtual Reality*, pages 275–276.

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

[Bhaniramka et al., 2005] Bhaniramka, P., Robert, P. C. D., and Eilemann, S. (2005). OpenGL Multipipe SDK: A toolkit for scalable parallel rendering. In *Proceedings IEEE Visualization*, pages 119–126.

[Bierbaum and Cruz-Neira, 2003] Bierbaum, A. and Cruz-Neira, C. (2003). ClusterJuggler: A modular architecture for immersive clustering. In *Proceedings Workshop on Commodity Clusters for Virtual Reality, IEEE Virtual Reality Conference*.

[Bierbaum et al., 2001] Bierbaum, A., Just, C., Hartling, P., Meinert, K., Baker, A., and Cruz-Neira, C. (2001). VR Juggler: A virtual platform for virtual

reality application development. In *Proceedings of IEEE Virtual Reality*, pages 89–96.

[Bittner et al., 2004] Bittner, J., Wimmer, M., Piringer, H., and Purgathofer, W. (2004). Coherent hierarchical culling: Hardware occlusion queries made useful. *Computer Graphics Forum*, 23(3):615–624.

[Blanke et al., 2000] Blanke, W., Bajaj, C., Fussel, D., and Zhang, X. (2000). The metabuffer: A scalable multi-resolution 3-d graphics system using commodity rendering engines. Technical Report TR2000-16, University of Texas at Austin.

[Castanie et al., 2006] Castanie, L., Mion, C., Cavin, X., and Levy, B. (2006). Distributed shared memory for roaming large volumes. *IEEE Transactions on Visualization and Computer Graphics*, 12:1299–1306.

[Cavin and Mion, 2006] Cavin, X. and Mion, C. (2006). Pipelined sort-last rendering: Scalability, performance and beyond. In *Proceedings Eurographics Symposium on Parallel Graphics and Visualization*.

[Cavin et al., 2005] Cavin, X., Mion, C., and Filbois, A. (2005). COTS cluster-based sort-last rendering: Performance evaluation and pipelined implementation. In *Proceedings IEEE Visualization*, pages 111–118. Computer Society Press.

[Chiueh et al., 1997] Chiueh, T.-c., Yang, C.-k., He, T., Pfister, H., and Kaufman, A. E. (1997). Integrated volume compression and visualization. In *Proceedings of the 8th conference on Visualization '97*, VIS '97, pages 329–336, Los Alamitos, CA, USA. IEEE Computer Society Press.

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

[Courbet and Hudelot, 2009] Courbet, C. and Hudelot, C. (2009). Random accessible hierarchical mesh compression for interactive visualization. In *Proceedings of the Symposium on Geometry Processing*, SGP '09, pages 1311–1318, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

[Crockett, 1997] Crockett, T. W. (1997). An introduction to parallel rendering. *Parallel Computing*, 23:819–843.

[Doerr and Kuester, 2011] Doerr, K.-U. and Kuester, F. (2011). CGLX: A scalable, high-performance visualization framework for networked display environments. *IEEE Transactions on Visualization and Computer Graphics*, 17(2):320–332.

[Eilemann et al., 2009] Eilemann, S., Makhinya, M., and Pajarola, R. (2009). Equalizer: A scalable parallel rendering framework. *IEEE Transactions on Visualization and Computer Graphics*, 15(3):436–452.

[Eilemann and Pajarola, 2007] Eilemann, S. and Pajarola, R. (2007). Direct send compositing for parallel sort-last rendering. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 29–36. Eurographics Association.

[Eyles et al., 1997] Eyles, J., Molnar, S., Poulton, J., Greer, T., Lastra, A., England, N., and Westover, L. (1997). PixelFlow: The realization. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics Hardware*, pages 57–68.

[Fout and Ma, 2007] Fout, N. and Ma, K.-L. (2007). Transform coding for hardware-accelerated volume rendering. *IEEE Trans Vis Comput Graph*, 13(6):1600–1607.

[Garcia and Shen, 2002] Garcia, A. and Shen, H.-W. (2002). An interleaved parallel volume renderer with PC-clusters. In *Proceedings Eurographics Workshop on Parallel Graphics and Visualization*, pages 51–60.

[Gobbetti et al., 2006] Gobbetti, E., Marton, F., Cignoni, P., Benedetto, M. D., and Ganovelli, F. (2006). C-BDAM – compressed batched dynamic adaptive meshes for terrain rendering. *Computer Graphics Forum*, 25(3):333–342.

[Goswami et al., 2010] Goswami, P., Makhinya, M., Bösch, J., and Pajarola, R. (2010). Scalable parallel out-of-core terrain rendering. In *Proceedings Eurographics Symposium on Parallel Graphics and Visualization*, pages 63–71.

[Hoppe, 1997] Hoppe, H. (1997). View-dependent refinement of progressive meshes. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '97, pages 189–198, New York, NY, USA. ACM Press/Addison-Wesley Publishing Co.

[Houston, 2004] Houston, M. (2004). Designing graphics clusters. parallel rendering workshop. In *IEEE Visualization Conference*.

[Howison et al., 2010] Howison, M., Bethel, E. W., and Childs, H. (2010). Mpi-hybrid parallelism for volume rendering on large, multi-core systems. In

Ahrens, J. P., Debattista, K., and Pajarola, R., editors, *EGPGV*, pages 1–10. Eurographics Association.

[Hui et al., 2009] Hui, C., Xiaoyong, L., and Shuling, D. (2009). A dynamic load balancing algorithm for sort-first rendering clusters. In *IEEE International Conference on Computer Science and Information Technology*, pages 515 – 519.

[Humphreys et al., 2000] Humphreys, G., Buck, I., Eldridge, M., and Hanrahan, P. (2000). Distributed rendering for scalable displays. *IEEE Supercomputing*.

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

[Humphreys and Hanrahan, 1999] Humphreys, G. and Hanrahan, P. (1999). A distributed graphics system for large tiled displays. *IEEE Visualization 1999*, pages 215–224.

[Humphreys et al., 2002] Humphreys, G., Houston, M., Ng, R., Frank, R., Ahern, S., Kirchner, P. D., and Klosowski, J. T. (2002). Chromium: A stream-processing framework for interactive rendering on clusters. *ACM Transactions on Graphics*, 21(3):693–702.

[Igehy et al., 1998] Igehy, H., Stoll, G., and Hanrahan, P. (1998). The design of a parallel graphics interface. *Proceedings of SIGGRAPH 98*, pages 141–150.

[Johnson et al., 2006] Johnson, A., Leigh, J., Morin, P., and Van Keken, P. (2006). GeoWall: Stereoscopic visualization for geoscience research and education. *IEEE Computer Graphics and Applications*, 26(6):10–14.

[Jones et al., 2004] Jones, K., Danzer, C., Byrnes, J., Jacobson, K., Bouchaud, P., Courvoisier, D., Eilemann, S., and Robert, P. (2004). SGI®OpenGL Multipipe™SDK User's Guide. Technical Report 007-4239-004, Silicon Graphics.

[Just et al., 1998] Just, C., Bierbaum, A., Baker, A., and Cruz-Neira, C. (1998). VR Juggler: A framework for virtual reality development. In *Proceedings Immersive Projection Technology Workshop*.

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

[Kolda and Bader, 2009] Kolda, T. G. and Bader, B. W. (2009). Tensor decompositions and applications. *SIAM Review*, 51:455–500.

[Koltun et al., 2000] Koltun, V., Chrysanthou, Y., and Cohen-Or, D. (2000). Virtual occluders: An efficient intermediate pvs representation. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, pages 59–70, London, UK. Springer-Verlag.

[Lee et al., 1996] Lee, T.-Y., Raghavendra, C., and Nicholas, J. B. (1996). Image composition schemes for sort-last polygon rendering on 2D mesh multicomputers. *IEEE Transactions on Visualization and Computer Graphics*, 2(3):202–217.

[Li et al., 1996] Li, P. P., Duquette, W. H., and Curkendall, D. W. (1996). RIVA: A versatile parallel rendering system for interactive scientific visualization. *IEEE Transactions on Visualization and Computer Graphics*, 2(3):186–201.

[Li et al., 1997] Li, P. P., Whitman, S., Mendoza, R., and Tsiao, J. (1997). ParVox: A parallel splatting volume rendering system for distributed visualization. In *Proceedings IEEE Parallel Rendering Symposium*, pages 7–14.

[Lombeyda et al., 2001] Lombeyda, S., Moll, L., Shand, M., Breen, D., and Heirich, A. (2001). Scalable interactive volume rendering using off-the-shelf components. Technical Report CACR-2001-189, California Institute of Technology.

[Ma et al., 1994] Ma, K.-L., Painter, J. S., Hansen, C. D., and Krogh, M. F. (1994). Parallel volume rendering using binary-swap image compositing. *IEEE Computer Graphics and Applications*, 14:59–68.

[Magallón et al., 2001] Magallón, M., Hopf, M., and Ertl, T. (2001). Parallel volume rendering using pc graphics hardware. In *Proceedings of the 9th Pacific Conference on Computer Graphics and Applications*, PG '01, pages 384–389, Washington, DC, USA. IEEE Computer Society.

[Makhinya et al., 2010] Makhinya, M., Eilemann, S., and Pajarola, R. (2010). Fast compositing for cluster-parallel rendering. In *Proceedings Eurographics Symposium on Parallel Graphics and Visualization*, pages 111–120.

[Marchesin et al., 2006] Marchesin, S., Mongenet, C., and Dischler, J.-M. (2006). Dynamic load balancing for parallel volume rendering. *IEEE Computer Graphics and Applications*, 14(4):41–48.

[Moll et al., 1999] Moll, L., Heirich, A., and Shand, M. (1999). Sepia: scalable 3D compositing using PCI pamette. In *Proceedings IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 146–155.

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

[Molnar et al., 1992] Molnar, S., Eyles, J., and Poulton, J. (1992). PixelFlow: High-speed rendering using image composition. In *Proceedings ACM SIGGRAPH*, pages 231–240.

[Mueller, 1995] Mueller, C. (1995). The sort-frst rendering architecture for high-performance graphics. In *Proceedings Symposium on Interactive 3D Graphics*, pages 75–84. ACM SIGGRAPH.

[Mueller, 1997] Mueller, C. (1997). Hierarchical graphics databases in sort-first. In *Proceedings IEEE Symposium on Parallel Rendering*, pages 49–. Computer Society Press.

[Muraki et al., 2001] Muraki, S., Ogata, M., Ma, K.-L., Koshizuka, K., Kajihara, K., Liu, X., Nagano, Y., and Shimokawa, K. (2001). Next-generation visual supercomputing using PC clusters with volume graphics hardware devices. In *Proceedings ACM/IEEE Conference on Supercomputing*, pages 51–51.

[Nie et al., 2005] Nie, W., Sun, J., Jin, J., Li, X., Yang, J., and Zhang, J. (2005). A dynamic parallel volume rendering computation mode based on cluster. In *Proceedings Computational Science and its Applications*, volume 3482 of *Lecture Notes in Computer Science*, pages 416–425.

[Niski and Cohen, 2007] Niski, K. and Cohen, J. D. (2007). Tile-based level of detail for the parallel age. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1352–1359.

[Osman and Ammar, 2002] Osman, A. and Ammar, H. (2002). Dynamic load balancing strategies for parallel computers. *Scientific Annals Journal of Cuza University, International Symposium on Parallel and Distributed Computing (ISPDC)*, 11:110–120.

[Richardson et al., 1998] Richardson, T., Stafford-Fraser, Q., Wood, K. R., and Hopper, A. (1998). Virtual network computing. *IEEE Internet Computing*, 2:33–38.

[Rohlf and Helman, 1994] Rohlf, J. and Helman, J. (1994). IRIS Performer: A high performance multiprocessing toolkit for real-time 3D graphics. In *Proceedings ACM SIGGRAPH*, pages 381–394. ACM Press.

[Samanta et al., 2001] Samanta, R., Funkhouser, T., and Li, K. (2001). Parallel rendering with K-way replication. In *Proceedings IEEE Symposium on Parallel and Large-Data Visualization and Graphics*. Computer Society Press.

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

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

[Sano et al., 2004] Sano, K., Kobayashi, Y., and Nakamura, T. (2004). Differential coding scheme for efficient parallel image composition on a pc cluster system. *Parallel Computing*, 30(2):285–299.

[Schneider and Westermann, 2003] Schneider, J. and Westermann, R. (2003). Compression domain volume rendering. In Turk, G., van Wijk, J. J., and II, R. J. M., editors, *IEEE Visualization*, pages 293–300. IEEE Computer Society.

[Shapiro, 1993] Shapiro, J. (1993). Embedded image coding using zerotrees of wavelet coefficients. *Signal Processing, IEEE Transactions on*, 41(12):3445 –3462.

[Staadt et al., 2003] Staadt, O. G., Walker, J., Nuber, C., and Hamann, B. (2003). A survey and performance analysis of software platforms for interactive cluster-based multi-screen rendering. In *Proceedings Eurographics Workshop on Virtual Environments*, pages 261–270.

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

[Stompel et al., 2003] Stompel, A., Ma, K.-L., Lum, E. B., Ahrens, J., and Patchett, J. (2003). SLIC: Scheduled linear image compositing for parallel volume rendering. In *Proceedings IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 33–40.

[Strugar, 2009] Strugar, F. (2009). Continuous distance-dependent level of detail for rendering heightmaps. *journal of graphics, gpu, and game tools*, 14(4):57–74.

[Suss et al., 2010] Suss, T., Wiesemann, T., and Fischer, M. (2010). Evaluation of a c-load-collision-protocol for load-balancing in interactive environments. In *Proceedings of the 2010 IEEE Fifth International Conference on Networking, Architecture, and Storage*, NAS '10, pages 448–456, Washington, DC, USA. IEEE Computer Society.

[Takeuchi et al., 2003] Takeuchi, A., Ino, F., and Hagihara, K. (2003). An improved binary-swap compositing for sort-last parallel rendering on distributed memory multiprocessors. *Parallel Computing*, 29(11-12):1745–1762.

[Vezina and Robertson, 1991] Vezina, G. and Robertson, P. K. (1991). Terrain perspectives on a massively parallel SIMD computer. In *Proceedings Computer Graphics International (CGI)*, pages 163–188.

[Wittenbrink, 1998] Wittenbrink, C. M. (1998). Survey of parallel volume rendering algorithms. In *Proceedings Parallel and Distributed Processing Techniques and Applications*, pages 1329–1336.

[Yang et al., 2001] Yang, D.-L., Yu, J.-C., and Chung, Y.-C. (2001). Efficient compositing methods for the sort-last-sparse parallel volume rendering system on distributed memory multicomputers. *Journal of Supercomputing*, 18(2):201–22–.

[Yeo and Liu, 1995] Yeo, B.-L. and Liu, B. (1995). Volume rendering of DCT-based compressed 3D scalar data. *IEEE Transactions on Visualization and Computer Graphics*, 1(1):29–43.

[Yu et al., 2008] Yu, H., Wang, C., and Ma, K.-L. (2008). Massively parallel volume rendering using 2-3 swap image compositing. In *Proceedings IEEE/ACM Supercomputing*.

[Zhang et al., 2001] Zhang, X., Bajaj, C., and Blanke, W. (2001). Scalable isosurface visualization of massive datasets on COTS clusters. In *Proceedings IEEE Symposium on Parallel and Large Data Visualization and Graphics*, pages 51–58.

CURRICULUM VITAE

Personal Information

Name	Maxim Makhinya Evgenievich
Date of birth	June 25, 1983
Place of birth	Bratsk, Russia

Education

2007 - 2012 Doctor in Informatics in the Visualization and Multimedia Lab, Department of Informatics, University of Zurich, Switzerland
Subject of dissertation: Performance Challenges in Distributed Rendering Systems
Advisor: Prof. Dr. R. Pajarola
Co-Examiner: Dr. B. Raffin

2005 Master of Science, Graphics and Media Lab, Department of Computer Science, Moscow State University, Russia
Subject of master thesis: Suppressing of de-interlacing artifacts in algorithm that uses bidirectional motion compensation
Advisor: Prof. Dr. Y. M. Bayakovski
Co-Advisor: Dr. D. S. Vatolin

2000 - 2004 Bachelor degree, Department of Computer Science, Moscow State University, Russia

1990 - 2000 Secondary school with a mathematical bias, Bratsk, Russia

Work experience

2004 - 2007 Researcher, Graphics and Media Lab, Moscow State University, Russia
Advisor: Dr. D. S. Vatolin

Publications

Conference Publications

P. Goswami, M. Makhinya, J. Bösch, R. Pajarola, Scalable Parallel Out-of-core Terrain Rendering, Proceedings Eurographics Symposium on Parallel Graphics and Visualization, 2010, pp. 63-71.

M. Makhinya, S. Eilemann, R. Pajarola, Fast Compositing for Cluster-Parallel Rendering, Proceedings Eurographics Symposium on Parallel Graphics and Visualization, 2010, pp. 111-120.

Journal Articles

S. Eilemann, M. Makhinya, R. Pajarola, Equalizer: A Scalable Parallel Rendering Framework, IEEE Transactions on Visualization and Computer Graphics, 2008, vol. 15, pp. 436-452.

P. Schlegel, M. Makhinya, R. Pajarola, Extinction-Based Shading and Illumination in GPU Volume Ray-Casting, IEEE Transactions on Visualization and Computer Graphics, 2011, vol. 17(12), pp. 2135-2143.