Functionality and Performance Visualization of the Distributed High Quality Volume Renderer (HVR)

Thesis by

Sara Mohammed Shaheen

In Partial Fulfillment of the Requirements for the Degree of

Master of Science

King Abdullah University of Science and Technology
Applied Mathematics and Computer Science (AMCS)
Computer Science (CS)

Thuwal, Makkah Province, Kingdom of Saudi Arabia

July, 2012
The undersigned approve the thesis of Your Full Name

Dr. Alyn Rockwood
Committee Member

Dr. Christopher Knox
Committee Member

Dr. Markus Hadwiger
Committee Chair (Thesis Supervisor)

King Abdullah University of Science and Technology

2012
ABSTRACT

Functionality and Performance Visualization of the Distributed High Quality Volume Renderer (HVR) Title

Sara Mohammed Shaheen

Volume rendering systems are designed to provide means to enable scientists and a variety of experts to interactively explore volume data through 3D views of the volume. However, volume rendering techniques are computationally intensive tasks. Moreover, parallel distributed volume rendering systems and multi-threading architectures were suggested as natural solutions to provide an acceptable volume rendering performance for very large volume data sizes, such as Electron Microscopy data (EM). This in turn adds another level of complexity when developing and manipulating volume rendering systems.

Given that distributed parallel volume rendering systems are among the most complex systems to develop, trace and debug, it is obvious that traditional debugging tools do not provide enough support. As a consequence, there is a great demand to provide tools that are able to facilitate the manipulation of such systems. This can be achieved by utilizing the power of compute graphics in designing visual representations that reflect how the system works and that visualize the current performance state of the system.
The work presented is categorized within the field of software Visualization, where Visualization is used to serve visualizing and understanding various software. In this thesis, a number of visual representations that reflect a number of functionality and performance aspects of the distributed HVR, a high quality volume renderer system that uses various techniques to visualize large volume sizes interactively. This work is provided to visualize different stages of the parallel volume rendering pipeline of HVR. This is along with means of performance analysis through a number of flexible and dynamic visualizations that reflect the current state of the system and enables manipulation of them at runtime. Those visualization are aimed to facilitate debugging, understanding and analyzing the distributed HVR.
Acknowledgments

"And say: My Lord! Increase me in knowledge.” Al Quran Surah 20: Verse 114

First, I praise God, the almighty for granting me this opportunity and providing me the capability to successfully accomplish this thesis. This work would not have been accomplished without the support of several great people around me for whom I dedicate this thesis.

I would like to deeply thank Prof. Markus Hadwiger who gave me this opportunity in the first place. My deepest thanks go to him for believing in me and for having me as a member of his great team during the thesis work. It is with his kindness and support I managed to complete this work within an amazing research environment. I also would like to express my deepest appreciation and gratitude to Dr. Johanna Beyer for being an exceptional teacher and supervisor. I would like to thank her for all the helpful advice, for the great pushing and the encouragement during the difficult times, and for being involved with her help and support in every detail of this work from the beginning till the end.

An honorable mention goes to my friends and colleagues. I start by thanking an exceptional close friend of mine, Sawsan Al-Halawani, for being such a supportive friend. I would like to deeply thank her for the sweetest joyful moments and for bearing hearing my complains and never hesitating to offer her help and valuable advice. I would like to also deliver my gratitude and deep thanks to some of my dear friends: Amal Aboulhassan, Imene Boudellioua and Enas Yunis. I sincerely thank...
them for being there whenever I needed them and for being a valuable source of motivation with their great company. I extend my thanks to Ronell Sicat for offering his technical support when I need it.

My utmost gratitude, deep love and appreciation should go to my parents for their dedication and endless patience. I owe it to them for any achievement I have made in this life including this thesis. It is their inspiration and their ever lasting support that gave me all the strength I needed to move forward. I would like to deeply thank them for being such great examples for me to imitate and follow. Those few lines cannot adequately express my feeling of appreciation towards them. Thank you for always pushing me to pursue my dreams and for dedicating your lives in eliminating any obstacles that can threaten them. Thank you for your prayers and lovely hugs and for having faith in me. I am also grateful to my sisters Wafa’a, Ghadah, Raghdah and Fatma and to my brother Abdulaziz for always being by my side and never delaying attempting revealing my concerns.

Lastly, I would like to thank all the people at KAUST, my friends and my family members for their help and support. Without their presence in my life, nothing would have been accomplished.
# TABLE OF CONTENTS

List of Illustrations .............................. 11

List of Tables .................................... 15

I Introduction ................................... 16
   I.1 Motivation .................................. 17
   I.2 Problem Statement and Objectives ........... 18
   I.3 Document Organization ..................... 19

II Fundamentals ................................. 21
   II.1 Volume Rendering .......................... 21
      II.1.1 Volume Rendering Methods .............. 22
   II.2 Physical Model of Volumetric Light Transport ... 22
   II.3 Transfer Functions ......................... 24
   II.4 Data Acquisition Techniques for Volume Rendering ... 25
   II.5 Basic Ray-Casting .......................... 27

III State of the Art ............................. 29
   III.1 Parallel Rendering in Distributed Volume Rendering Systems ... 29
      III.1.1 The sorting strategy .................. 30
      III.1.2 Memory architecture .................. 33
      III.1.3 Parallel rendering on clusters ....... 34
III.1.4 Parallel compositing .............................................. 35
III.1.5 Load balancing .................................................... 37

III.2 Software Visualization .............................................. 39
III.2.1 Software visualization for supporting debugging ................. 41
III.2.2 Visualization of visualization software .............................. 44
III.2.3 Visualization of the software development process ................. 46

IV HVR Environment ...................................................... 49
IV.1 Distributed HVR Motivation and Goals ................................. 50
IV.2 Distributed Architecture Overview ................................... 51
IV.3 Multilevel out-of-core memory hierarchy .............................. 52
IV.4 Adaptation of a Multi-resolution Scheme Using Octrees .......... 53
IV.5 Distributed Rendering and Composition ................................ 54
IV.5.1 Volume distribution and implicit load balancing ................. 54
IV.5.2 Distribution unit rendering ......................................... 56
IV.5.3 Composition ........................................................... 57
IV.6 Overview of Multi-Threads Communication Using MPI ............ 58

V HVR Functionality and Performance Visualization ..................... 61
V.1 Demand-Driven Data Back-End Visualization .......................... 62
V.1.1 HVR demand-driven data back-end overview ....................... 62
V.1.2 Implementation ....................................................... 65
V.1.3 Results ................................................................. 67
V.2 Distribution Unit Rendering Visualization ............................... 67
V.2.1 Distribution unit rendering overview ................................ 68
V.2.2 Implementation ........................................................ 68
V.2.3 Results ................................................................. 71
V.3 Distribution Unit Composition Visualization ............................. 73
# List of Illustrations

I.1 A collection of different HVR functionality and performance visualizations presented in this thesis .......................... 17

II.1 Emission and absorption of the light along the ray that is commonly used in DVR. Image is courtesy of [1] ..................... 23

II.2 Different optical properties are enabled by applying different transfer functions to the same volume data. Image is a courtesy of [2] .... 25

II.3 Basic ray-casting technique, an imaginary ray is shot for every pixel until reaching the back of the image. Image is courtesy of [1] .... 27

III.1 The sort-first and sort-last distributed rendering schemes. Image is courtesy of [3] ..................................................... 30

III.2 The binary swap composition scheme. The image is split among two processors in each level, shown is a system with a total of four processors. Image is a courtesy of [4] ................................. 36

III.3 Visualization disciplines hierarchy ........................................... 40

III.4 An example of using software visualization in a form a three-dimensional circular bundle to detect system behavior. The outer ring represents the system architecture and the edges within the circle represent system dependencies. Metrics can be obtained by observing color, width and height of ring elements. Image is a courtesy of [5] ............ 43
III.5 An example of using color maps to extract the characteristics of system primitives. To trace the behavior of mesh elements, each quad is color-mapped based on the characteristic of the flow field. Image is courtesy of [6].

III.6 A screenshot of the StarGate visualization system. Circles represent developers. Circles colors represent the domain where the developers work and their sizes represent the amount of contribution each developer accomplished in a specific domain. Connections between developers are also visualized. Image is a courtesy [7].

IV.1 Distributed HVR networking architecture. Image is courtesy of [8].

IV.2 HVR Multilevel Cache Hierarchy Architecture. Image is courtesy of [8].

IV.3 Spatial arrangement of distribution blocks. Image is courtesy of [8].

IV.4 Per-Node Threads Asynchronous Communication in Distributed HVR. The main thread is responsible for managing MPI sending and receiving messages between the nodes along with handling client/server communication issues. The render thread constantly checks if a new frame update has been received, if so it renders and stores results in buffer. The compositing thread: constantly checks if new screen tiles have arrived for compositing, composites, and stores results.

V.1 Demand-driven data back-end within the context of the HVR rendering pipeline.

V.2 Demand-driven data back-end requests visualization. There is a total of two nodes, shown in different colors (a) Requested blocks streaming at different resolution levels (b) Requested blocks visualization along with the actual volume being rendered.
V.3 Illustration of Volume data communication paths during the rendering process in the context of distributed HVR. 69

V.4 Illustration of rendered tiles communication paths to create per-distribution unit visualization for distributed HVR. 70

V.5 Distributed rendering visualization. It shows a separated rendering view of eight distribution blocks which are together forming the entire volume view of the current frame. Distribution block 0 is on the top left corner. Below the separated rendering view, the actual rendered volume is displayed. 72

V.6 Direct-send parallel composition. The first three images shows the state of partial image tiles after rendering and the last fourth image is final compositied image. Image is courtesy of [9] 73

V.7 Direct-send final image partitioning into n horizontal stripes to handle composition in parallel across n nodes. 75

V.8 Distributed composition visualization. It shows the compositied image parts produced by each node separately and the composition was carried out across a total of two nodes. The user can input a valid node id number to view the result of composition by that node. 77

V.9 Visualization of image partitioning in screen-space. 79

V.10 Visualization of the volume distribution in object space following a 3D checkerboard pattern. The visualization provided is for a volume of $512^4$ resolution that is divided into a total of eight distribution blocks each of size $256^3$. 81
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>V.11</td>
<td>Visualization of the volume distribution in object space for a volume of 1008 x 1065 x 101 resolution that is divided into a total of two distribution blocks distributed among two nodes. The image presents a situation of load imbalancing as the work size is not equally distributed.</td>
</tr>
<tr>
<td>V.12</td>
<td>The Visualization result of the volume bounding box and view frustum.</td>
</tr>
<tr>
<td>V.13</td>
<td>Rendering frame-rates visualization graph.</td>
</tr>
<tr>
<td>V.14</td>
<td>Blending frame-rates visualization graph.</td>
</tr>
<tr>
<td>V.15</td>
<td>Copy back frame-rates visualization graph.</td>
</tr>
<tr>
<td>V.16</td>
<td>Networking frame-rates visualization graph.</td>
</tr>
<tr>
<td>V.17</td>
<td>Overall Frame-rates visualization graph.</td>
</tr>
<tr>
<td>V.18</td>
<td>Simultaneous frame-rates graphs visualization.</td>
</tr>
<tr>
<td>V.19</td>
<td>Graph visualization of the number of downloaded data blocks to GPU.</td>
</tr>
<tr>
<td>V.20</td>
<td>A graph visualization of the block cache usage percentage.</td>
</tr>
<tr>
<td>V.21</td>
<td>The number of downloaded blocks to GPU on each node is shown on top and a progress bar of the GPU cache usage percentage for the current frame is shown below.</td>
</tr>
<tr>
<td>V.22</td>
<td>A graph representation of the average rendering frame rates for various data sets.</td>
</tr>
<tr>
<td>V.23</td>
<td>A graph representation of the average times consumed at different stages of the rendering process for the Alignment data set.</td>
</tr>
<tr>
<td>V.24</td>
<td>A graph representation of the obtained average GPU cache usage percentage on each node for various transfer functions. The block cache usage percentage increases as the transparency level of a transfer function increases.</td>
</tr>
</tbody>
</table>
## List of Tables

| V.1  | The obtained average rendering frame rates measured in frames per second for various data sets (fps). | 92 |
| V.2  | The obtained average times at different stages of the rendering process measured in milliseconds (ms) across three data sets. | 94 |
| V.3  | The obtained average overall frame rates measured in frames per second for various data sets (fps). | 95 |
| V.4  | The obtained average GPU cache usage percentage on each node for various data sets (fps). | 96 |
| V.5  | The obtained average GPU cache usage percentage on each node for various transfer functions. While TF1 represents a completely opaque transfer function, TF2 represents a semi-transparent one and TF3 represents a more transparent transfer function. | 97 |
Chapter I

Introduction

"The understanding can intuit nothing, the senses can think nothing. Only through their union can knowledge arise" By Immanuel Kant, 1781.

This valuable quote forms the main goal of this thesis. The work presented in this thesis is aimed to utilize the power of computer graphics to provide functionality visualizations and means of performance analysis for the distributed HVR, a high quality volume renderer system that uses various techniques to visualize large volume sizes interactively. The work presented is categorized within the field of software visualization, where visualization is used to serve visualizing and understanding various software. The proposed visualizations provided in this work are built using multiple graphical views which are able to reflect how the HVR system operates and performs. Together, they form the start towards building a friendly debugging and performance tracking environment of such complex system, which in turn is predicted to enhance the quality of the overall development process. Figure I.1 shows a collection of a number of visualizations presented in this thesis.
17

Figure I.1: A collection of different HVR functionality and performance visualizations presented in this thesis

I.1 Motivation

Volume rendering visualization systems are aimed to provide interactive means to enable scientists, engineers, and a variety of experts to explore volume data by providing well presented 3D views of the volume. However, volume rendering techniques are computationally intensive tasks which requires developers and researchers to dedicate a lot of efforts to provide various techniques and algorithms aiming to produce high quality volume rendering with an acceptable performance level. Moreover, volume data sets are getting bigger and bigger, especially for Electron Microscopy (EM) volume data which is the main focus of the distributed HVR and that and that are often extremely large. EM scans result in a very large volume data size, sometimes reaches up to hundreds of Terabytes. As a result, parallel distributed volume rendering systems and multi-threading architectures were proposed as natural solutions to efficiently render large volume sizes along with the use of computationally intensive
algorithms by increasing the number of processing units and allowing for concurrent execution.

All of the above challenges and the suggested solutions for them have contributed in making developing, debugging and analyzing volume rendering systems time consuming and complex procedures. Moreover, traditional debugging tools do not provide enough support and do not fulfill the developer need for smoothly analyzing and debugging such large complex systems. As a consequence, there is a great demand to provide tools that are able to facilitate the manipulation of the current volume renderers. One proposed solution is to use the power of computer graphics to design visual representations that reflect how the system works and to visualize the current performance state of the system.

I.2 Problem Statement and Objectives

Distributed parallel volume rendering systems are among the most complex systems to develop, trace, and debug [6]. This is because of the huge data sizes, complex data structures and non-trivial algorithms involved with the current volume renderers. Moreover, mapping functionality to the code is not a straight forward process especially when multi-threading architecture is employed. Available debugging tools are not enough to overcome this difficulty. Visual debugging tools are among the best suggestions to facilitate the work upon such dynamic visualization systems. However, it is recommended that for better analysis and debugging support, visualization debugging tools should be system specific as volume renderer systems have different implementation approaches and needs and different parameters are needed to be visualized [10]. Before this thesis, there were no visualizations provided in analyzing the distributed high quality volume renderer system (HVR).

The goal of this thesis work is to utilize computer graphics abilities in designing
visual representations that reflect some functionality and performance aspects of the distributed interactive high quality volume renderer (HVR). This work is provided to visualize different stages of the parallel volume rendering pipeline of HVR. This is along with means of performance analysis through a number of flexible and dynamic visualizations that reflect the current state of the system and enables manipulation of them at runtime. Those visualization are aimed to facilitate debugging, understanding and analyzing the distributed HVR.

1.3 Document Organization

This thesis is organized as follows. Chapter II introduces a number of fundamental concepts related to the volume rendering field. Different methods of volume rendering are first discussed. After that, physical models of light transport, transfer functions, different data acquisition techniques, and basic ray-casting are presented. This chapter is included to help discussing the thesis topic in further details in the next chapters. The current state of the art of distributed volume rendering and software visualization is presented in chapter III. The first part of this chapter investigates a number of stages in the parallel volume rendering pipeline. Moreover, it discusses related concepts including memory architecture, examples of parallel rendering systems, and load balancing techniques in parallel volume rendering systems. The second part of this chapter presents the related work in the field of software visualization which is presented as a discipline under the field of visualization. The need of software visualization is first emphasised. Later in the chapter, more focus is given to the use of software visualization in serving visualization applications. Chapter IV provides an insight into the HVR framework, an interactive high quality volume renderer system that includes a variety of techniques to visualize large volume data. As the work on this thesis was implemented for serving the distributed HVR system, this chapter
places the work provided in this thesis in the right context. The HVR main goals and objectives are discussed along with an insight into the HVR architectural choices. Moreover, a number of techniques that are implemented to visualize large volumes are also discussed. Chapter V presents the work contribution provided by this thesis. This chapter is divided into the different visualizations that were implemented in approaching the thesis work. For each visualization, implementation details and results are discussed along with placing each provided visualization in the right context of the HVR environment. Later in the chapter, performance analysis that is based on readings provided by the performance visualization is discussed. The very last chapter of this document is chapter VI. It provides the general conclusions for this thesis and briefly states the overall work that has been accomplished.
Chapter II

Fundamentals

This chapter provides a number of fundamental concepts related to the field of volume rendering which will assist in discussing the thesis topic with more details in the upcoming chapters. After generally defining the field of volume rendering and its two different methods, more attention will be given to introduce physical models of light transport, transfer functions, different data acquisition techniques, and basic ray-casting.

II.1 Volume Rendering

Volume Rendering is one of the fields classified under Computer Graphics. It provides scientific visualization of volumetric data where the biggest challenge is the interaction between light and the participating media. This interaction must be evaluated at every position of the 3D volume which makes volume rendering a computationally intensive task and that in turn imposes an increased demand for efficient volume rendering techniques [1]. As a result, lots of efforts have been made by scientists in applying smart rendering techniques and dedicated volume rendering systems, such as the High Quality Volume Renderer (HVR) [8]. Volume data needed for rendering can be obtained by variety of means, mainly by measurements or by numerical simu-
lations. For example, medical data is obtained by measurement using different types of medical scans, such Electron Microscopy (EM) and Computed Tomography which is discussed in detail in section II.4. On the other hand, fluid dynamics, seismography, and molecular biology are examples of obtaining volume data using numerical simulations.

II.1.1 Volume Rendering Methods

Volume rendering strategies are classified as either direct volume rendering (DVR) or indirect volume rendering (IDVR) techniques. Direct volume rendering can directly generate a 2D image from 3D volume data. Ray-casting, introduced in section II.5 and material percentages [11] are two well known direct volume rendering techniques that do not need geometric representation. In contrast, indirect techniques are based on extracting surfaces from the volume data in a pre-processing step. Intermediate geometric representations are used to approximate the volume. Moreover, indirect rendering techniques visualize only the boundaries of a given volume or material as the geometric representation uses binary classification. This is because it is only concerned whether the surface passes through the voxel or not [12] [11]. Marching cubes algorithm is one example of an indirect volume rendering method [13].

II.2 Physical Model of Volumetric Light Transport

The interaction between light and the participating media is one of the most important aspects of volume rendering. As a result, presenting the light transfer equation and the different optical models resulted is vital. In direct volume rendering, first the data of the volume is sampled in a rectilinear grid or even an irregular grid. After that, interpolation takes place to make it possible to have a scalar function $f(x)$ defined for all sampled points $x$ in the volume. Optical properties can then be included as
functions of the interpolated value \( f(x) \). Light travels in straight lines, known as rays, from a light source behind the volume and goes through the volume where different optical effects take place until reaching the eyes of the viewer. Those different optical effects are related to the optical properties of the participating medium particles \([14]\).

The following are the different types of interaction between light and the participating medium.

1. Absorption. When particles of the participating medium absorb light, this means that radiative energy is being transferred into heat and as a result light energy is reduced. Absorption in its extreme case occurs when particles are perfectly cold and black so they absorb all coming light and emit none.

2. Emission. The particles of the medium can emit light and thus they increase the light energy of light rays passing through them. In this case, heat is being transferred into radiative energy. Emission in its extreme case occurs when particles of the medium emit light while being completely transparent so absorption and scattering are neglected. A combination between absorption and emission interaction is the most commonly used in volume rendering. The particles of the medium can both absorb incident light and emit light. Figure-1 shows the emission and the absorption of the light along the ray.

3. Scattering. Light can also be scattered by particles of the medium, by changing the direction of the light propagation. There are two types of scattering. The
first type is the in-scattering which causes an increased amount of radiative energy of an incident light ray on a particle. Light comes from different directions including the direction of the viewing ray. Out-scattering is the second type of scattering and it causes a decreased amount of radiative energy of an incident light ray. This is because light scatters out in different directions other than the direction of the incident light ray \[1\].

For a participating medium that emits scatters and absorbs light, the complete light transport equation in its differential form is:

\[
\frac{dI(s)}{ds} = -\rho(s)I(s) + g(s),
\]  

where \( I \) represents the intensity and \( \rho \) is the attenuation caused by absorption and out-scattering. Emission and in-scattering are represented by the term \( g(s) \). There are different simplified lighting models that are usually used due to the computational intensity of equation 1. More in-depth explanation and the classification of different lighting models can be found by referring to \[14\] and \[1\].

### II.3 Transfer Functions

Transfer functions are arbitrarily mapped functions that are needed to assign optical properties to the visualized data. These functions can be extracted either from the volume or from the images and they play a major role in producing meaningful and intelligible rendering results. Along with transfer functions, classification is always mentioned. Classification represents the process of choosing among transfer functions to highlight certain physical properties of the volume \[15\]. Figure \[2\] shows different optical properties of the same data by just applying different transfer functions.

Given that transfer functions play a critical role in visualization results, leads to the need to have simpler, but more powerful techniques to set transfer functions.
II.2 Different optical properties are enabled by applying different transfer functions to the same volume data. Image is a courtesy of [2].

As consequence, setting transfer functions and manipulating them should become a feasible process to both experts and non-experts. Efforts were made to provide an abstraction level that provides a friendly environment to specify transfer functions parameters as suggested by RezkSalama et al. [2].

### II.4 Data Acquisition Techniques for Volume Rendering

There is a variety of data acquisition techniques used to obtain data for volume rendering. Below are the main data acquisition techniques used to obtain data from the medical field along with their main uses and basic technology.

- **X-Rays** is the most common form of diagnostic imaging. It is based on measuring the attenuation of electromagnetic radiation travelling through a scanned object. Based on the tissue density, x-rays get absorbed and scattered as they are passing through an object. As a result, dense objects are viewed as bright areas, such as bone tissue, and less dense objects, air for example, are viewed as dark areas.

- **CT (Computerized Tomography)** is accomplished by running multiple X-Rays through the object from different viewpoints and then projecting them back
to form a 3D volume of the scanned object. CT is well-suited for 3D volume representation of dense tissues and most commonly bone tissue.

- EM (Electron Microscopy) produces images with very high resolutions of 3-5 nanometers and slice distances of 20-30 nanometers. This results in large volume sizes of hundreds of gigabytes or even hundreds of terabytes. It uses an electron beam to create the images. The very high resolution is because of the fact that electrons sizes are much smaller than visible photons. EM microscopy exists to enable neuroscientists to examine very tiny structures of human brain tissues at a very high resolution. Due to the resulted very large volume sizes, EM scans are targeted for volume rendering using distributed volume rendering systems in the scientific visualization field.

- MRI (Magnetic Resonance Imaging) is based on how human body tissues react differently in magnetic field. The object intended to be scanned is first exposed to a strong magnetic field so that atomic nuclei aligns themselves to the magnetic field. After that, a radio frequency pulse is induced which causes the atomic nuclei to spin in phase. After stopping the radio-frequency pulse the protons start to relax and measuring the time of this relaxation can be used to calculate the protons density. The final 3D image depicted by MRI is a representation of the proton density differences of the scanned tissue. MRI is perfect for soft tissues like brain tissue.

- PET (Positron Emission Tomography) is based upon tracing the metabolic activity of the tissue. It involves injecting the patient with radioactive substance. When this injected material starts to decay, gamma rays are emitted. With certain measurement of the gamma rays and their movement direction their source can be located. By using these information and with the help of computer analysis, images are constructed. As PET can be used for within limited space only,
II.5 Basic Ray-Casting

Ray-casting is a fundamental concept for applying direct volume rendering. In ray-casting, an imaginary ray per pixel is shot from the viewpoint until reaching the back of the image plane, as shown in figure II.3. The image background is set to be fully opaque to denote rays stopping points. Generated rays are traces as they are passing the volume at equally separated points. The collected values of tracing are composited in a back-to-front order. Composition assigns color to the pixel where the ray passed through. This briefly introduces the original ray-casting algorithm [12]. However, it is prone to artifacts that are generated during acquisition. Moreover, it is considered to be computationally intensive. As a consequence, efforts have been dedicated to come up with enhancements to be added to the original concept of ray-casting. Two major optimization techniques are early ray termination and empty space skipping. In early ray termination, rays stop traversing the entire volume if an opacity threshold reached already. Empty space skipping, on the other hand, an additional data structure is

Figure II.3: Basic ray-casting technique, an imaginary ray is shot for every pixel until reaching the back of the image. Image is courtesy of [1].

it is usually viewed on top of MRI and CT images to show the special context of the PET image. PET is very useful in viewing areas with high metabolic activity such as tumors [8] [1].
used to determine transparent regions to be skipped during ray traversal [1].
Chapter III

State of the Art

After covering a number of fundamental concepts related to the field of volume rendering in chapter II, this chapter provides more specific related work to the topic of this thesis. First, different stages and implementation choices involved in the parallel volume rendering pipeline are investigated. Next, the field of software visualization is discussed in detail.

III.1 Parallel Rendering in Distributed Volume Rendering Systems

As discussed in section II.1.1, direct volume rendering (DVR) has the advantage of being able to map 3D volume data into a 2D image plane without the need of an intermediate geometric representation. However, DVR is computationally intensive, especially that volume data sizes are getting larger given the advances in the data acquisition techniques which are now able to collect data at a very high resolution level. As a consequence, parallel volume rendering was introduced as a natural solution to provide a satisfiable performance by increasing the number of processing units. Unfortunately, this has induced a new set of challenges in the field of volume
rendering. Parallel volume rendering implementation should achieve a reasonable optimal performance through all the stages of the volume rendering pipeline. Work should be distributed as much equal as possible among the processors of the system. In this section, some aspects and methodologies related to parallel volume rendering of which within the scope of this thesis are discussed.

III.1.1 The sorting strategy

Sorting strategies are mechanisms used to determine the visible surface order in the rendering pipeline so that correct composition results are obtained. Molnar provided a comprehensive classification and discussion of the different parallel rendering algorithms. Rendering can simply be introduced as the set of calculations needed to map different primitives to turn graphical primitives into an image. According to Molnar, this can be viewed as sorting the primitives to the screen. In parallel processing, raw primitives of the graphical database are first passed through geometry processing units and then the results are passed through rasterization units. Based on that, three categories of sorting algorithms have been developed. These algo-
rithms are sort-first, sort-last and sort-middle. Each of these algorithms has its own usage, advantages, and drawbacks. However in the context of volume rendering, the sort-middle approach is impractical [16]. This is because systems that are based on consumer graphics hardware are constrained to the available standard APIs, such as OpenGL. And those APIs give very limited access to intermediate rendering results [17]. Coming parts in discussing sorting algorithms will introduce both sort-first and sort-last systems and then hybrid systems which combines both of them.

1. Sort-first systems

Sort-first rendering systems employ an image-order distribution technique. First, volume data is split randomly across the rendering nodes. In other words, the final image is split into tiles that are distributed across processors. After that, each rendering node determines what parts of the volume it is responsible for and keeps these parts while the remaining subvolume parts are communicated to other processors that are responsible to render them. The full graphics pipeline is executed independently on each rendering node. This adds a great advantage to the sort-first algorithm as backward compatibility with previously designed serial algorithms has become possible. Sort-first algorithm can become more efficient if communication between processors is reduced by enabling parallel I/O so that each processor can load the data it is responsible for rendering directly from the desk [18]. Per-frame coherence is another advantage that can be utilized when applying sort-first algorithm [19]. It simply means to keep the rendered geometry of one frame and use it in the next frame specially if the scene being rendered does not encounter sudden changes. However frame-to-frame coherence might cause load balancing issues between the nodes as primitives might be grouped into clusters. Moreover, sort-first algorithm is infeasible for large data, because if the view-port changes then a huge amount of new data has to be loaded on each node every frame.
2. Sort-last systems

In sort-last systems, the volume is split into sub-volumes and distributed among rendering processors following object-order distribution technique. On each node, sub-volumes are rendered into partial images and required transformations are applied. After that, compositing of all partial images takes place using one of the common parallel compositing algorithms as discussed in section III.1.4. One important thing to consider is that partial images must be rendered in a specific order to get correct results, i.e. volume pieces should be occluded by the ones in-front of them [3]. Parallel I/O can be an enhancement to sort-last algorithm as rendering processors can concurrently load volume pieces. Moreover, unlike sort-first systems, the distribution in sort-last algorithms is view independent in object space as it can be fixed and redistribution is not required. An advantage of this independency is reduced network communication in distributed memory systems [20]. In addition, performance in sort-last systems is highly dependent on the network bandwidth and as bandwidth demands increase for high screen resolutions, sort-last systems might not scale efficiently enough in high resolutions screen system. However, high speed network communication in serving the increased demand to provide distributed memory systems for the ever increasing data sizes has become available. Given that sort-last systems are usually built on top of fast distributed memory architectures and data can be stationary with minimum transfers, sort-last algorithms are scalable to high resolution when fast network communication is provided [4]. Figure III.1 shows a comparison between sort-last and sort-first algorithms schemes in distributed rendering systems.

3. Hybrid systems

Hybrid approaches that combine both sort-first and sort-last algorithms also
exist. One hybrid approach is based on performing the rendering in object space but the compositing in image space. Both spaces can be split into tiles on a dynamic basis. Hybrid approaches discussion and implementation can be found in [20] [4] [21].

III.1.2 Memory architecture

The architecture design choices of a parallel rendering system, such as the choice of a memory organization, has a great impact on the overall design and performance of the targeted system. The memory architecture can be either based on a distributed memory architecture, a shared memory architecture or a hybrid architecture that combines them.

Rendering systems based on distributed memory usually employ object space partitioning. A distributed system consists of a set of interconnected processing units each of which has its own local memory and processing power. The total distributed memory of the system is the summation of all individual memories that are available locally in each unit. Although it looks as a big connected space, there are mechanisms which assures the seamless availability of the data. One main advantage of distributed memory architecture is its scalability as new nodes can be easily added to increase the amount of memory available as needed. However, inter-node communication requires the availability of high network speed or otherwise data transfer across processors can cause a serious bottleneck. Examples of distributed memory implementation can be found in Garcia [22], and Muraki [23].

Shared memory architectures, on the other hand, provides access to a single, continuous and global shared memory space. As data communication is based on actual memory transfer, faster communication is guaranteed compared to the distributed memory architectures. However, it suffers from isolation and lack of extensibility because of the overhead and complexity of adding a bigger shared memory space [24].
Hybrid memory architectures are another category of memory architectures that combine features from shared and distributed memory architectures. This is possible by providing clusters of multicore processors. An example of adapting a hybrid memory architecture that uses image space partitioning scheme, can be found in [25]. However, one challenge involved with adapting hybrid memory architecture is the ability of the system to maintain cache coherence while the program is executed to avoid severe degradation in performance [26].

III.1.3 Parallel rendering on clusters

In meeting the computational power demands and the need to process large data sizes, in a variety of applications in the engineering, medical imaging and scientific visualization fields, a number of parallel volume rendering systems have been designed. Clusters of PCs or GPUs have recently been proposed for designing powerful parallel rendering systems as an alternative to supercomputers. This shifting was encouraged by the availability of very high network communications speed with considerably low prices along with the availability of graphics hardware to consumer markets.

CPU clusters are CPU based parallel systems. CPU clusters were examined in an interesting study and analysis provided by Petrek el al. [27] [28] on the scalability and cost overhead of the IBM Blue Gene/P parallel volume renderer. In their analysis they suggested the use of interconnected CPUs following a distributed memory architecture instead of using GPU parallel architecture for parallel volume rendering for very large volume data sizes. Some developers have adapted the use of parallel CPU architecture instead of GPU clusters. Their choice is not because they were expecting a better performance of CPUs over GPUs parallel architecture, but because CPU clusters are affordable and more available than GPU clusters, as justified by Marchesin and Ma [29]. One example of a volume rendering system using a cluster of PCs was proposed by Magallon et al. [30]. Their parallel volume renderer implements a sort-
last algorithm and it is distinguished with its ability to provide interactive frame rates in their system. Another parallel volume rendering system which runs on a cluster of PCs with 100 BaseT network was proposed by Stompel et al. [21]. Their volume renderer is based on a sort-last algorithm as well.

GPU clusters also comprise a number of PCs, but each GPU has an access to a hardware accelerator. A GPU based parallel volume rendering system which is closely related to the interest of this thesis was proposed by Muller [31]. Their system followed a client/server distributed architecture where computations are mainly carried out on the server side and a thin client is used to display rendering results and to interact with users. On the server side communication is managed using MPI while TCP/IP connections are used between the thin client and the server. Moreover, compression of an image before sending it is given as a choice in their system. Fogal et al. [32] have also proposed a GPU based parallel system which is able to assign multiple GPUs per processing unit or node. To make this possible, the distributed architecture maps every GPU to a CPU core in their parallel architecture. Moreover, Stuart et al. [33] presented a similar work that uses cluster of GPUs. They used a technique called MapReduce which is able to perform parallel computations and also able to handle communication issues.

III.1.4 Parallel compositing

When sort-last algorithms are used for parallel volume rendering in distributed systems, compositing is needed to put together the resulting partial image tiles. The need to come up with parallel compositing algorithms was due to the lack of efficiency induced by using serial compositing schemes in distributed rendering systems. When serial compositing algorithm is applied, every processor will produce a full size partial image, such that n partial images are produced by n processors. All tiles are sent to the master processor where final image blending takes place using visibility or
Figure III.2: The binary swap composition scheme. The image is split among two processors in each level, shown is a system with a total of four processors. Image is a courtesy of [4].

depth tests. If the image size is large, data transfer can become a serious bottleneck. Obtaining the required bandwidth can be impossible if no high speed network is available. Moreover, the transferred partial image tiles are sometimes unfinished pixels [20]. This can be an obstacle to scalability when sort-last algorithms are applied. All of these limitations involved with serial compositing has lead to the invention of a number of parallel compositing algorithms. Direct send, Binary swap and hybrid algorithms are discussed in the coming parts.

1. Direct send

First, the final image is split in screen space into a number of tiles that equals the number of available processors. After that, each processor sends the other screen tiles of its rendered image to the node responsible for compositing that tile. [20] [9] [21]. When tiles have become available locally on each node, compositing takes place. Next, composited tiles are sent to the master node for final image blending. The direct send algorithm performs better as the number
of processors increase. This is because of the reduced communication size as the size of exchanged tiles is smaller. The direct send algorithm is discussed in details in section V.3.1.

2. Binary swap

In Binary swap, communication is carried out only between two processors at any time. This has the advantage to reduce the overall communication overhead by adapting a hierarchical communication scheme which eliminates the all-to-all communication pattern [4]. Figure III.2 illustrates the binary swap composition scheme where the image is split between two corresponding processors. However, this limits the number of processors to powers of two. To overcome this limitation, 2-3 swap algorithm was introduced which allows for any number of processors count by allowing three processors to communicate instead of only two [34].

3. Hybrid methods

Hybrid algorithms exist which combines both direct and binary swap algorithms in an adaptive way, by depending on the current problem settings. Radix-k algorithm is one example of hybrid algorithms [35]. Another example of using hybrid methods to apply parallel compositing is provided as libraries such as IceT which is implemented and discussed in details in [32].

III.1.5 Load balancing

Load balancing is a very important aspect in assessing the effectiveness of parallel architectures. It refers to how evenly the work is distributed among different processors of a distributed parallel system, including parallel volume renderers. This is due to the fact that the speed of the overall system is bounded to the speed of the slowest unit. In other words, severe state of load imbalance threatens to destroy the
overall value behind using parallel volume renderers. Given all of that, techniques
to evenly distribute the work load and the data load among processors in parallel
volume renderers have been investigated and provided to achieve the best possible
even work distribution. There are two categories of load balancing techniques which
will be discussed in detail.

1. Static load balancing

Static load balancing is commonly used by the developers of parallel volume
rendering systems. This is because it suffers from less overhead as work is
distributed between processors only once during the whole rendering pipeline.
In addition to the lightness of execution, static load balancing is fairly simple
to be implemented. However, the partitioning usually takes place in a pre-
processing step according to the parameters of visualization, grid type, and data
for the unstructured data-set. Work is divided following either contiguous
or interleaved partitioning. In contiguous splitting, the image plane is divided
into one block and each processor is assigned one piece. The drawback of
contiguous partitioning is that its quality of distribution is labeled to be poor
due to the view dependent complexity of the image. Interleaved partitioning,
on the other hand, usually results in more even work distribution. It splits the
image plane into a number of pieces and distribute them among the processors
following a round-robin fashion as suggested by Ma and Crockett.

2. Dynamic load balancing

In dynamic load balancing, usually more evenly work distribution is achieved
compared to static load balancing. It uses information, such as previous frame
time, per-pixel rendering costs, and occlusion information, to equally redis-
tribute the work among rendering processors in the next frame. A kd-tree data
structure is generally used to build the hierarchy that manages those infor-
It encodes the decomposition of data space following a hierarchical architecture such that in each level of the kd-tree the data is split alternating along the orthogonal axes. In kd-tree, inner nodes contain the data as chosen by load statics obtained from various resources related to the previous frame while the leaf nodes encode the efficient partitioning of data. The traversal of a kd-tree takes place in a specific order to achieve balanced partitioning. Besides kd-tree algorithm, a different technique that is based on adding a central queue of work and that is also accessible by all rendering processors is provided. A shared queue is implemented that contains a front-to-back sorted list of work. Whenever a renderer node is ready to render, it accesses that queue, renders the corresponding bricks, and stores the rendering result to a shared frame buffer. The use of dynamic load balancing strategy in a parallel volume rendering algorithms was suggested by Lacroute. In their work, they proved that data redistribution overhead does not exceed acceptable time on shared memory architectures. Moreover, Amin suggested a parallel algorithm based on the shear-wrap algorithm that supports dynamic load balancing in distributed memory architectures. Generally, dynamic load balancing induces an extra overhead because of the per-frame queues management, or load statics calculations, and the need for frequent data redistribution.

III.2 Software Visualization

"...thought is impossible without an image" By Aristotle, 350 BC.

This is such a valuable philosophy which reflects the great value of using images and visual representations of ideas in enhancing people’s reasoning abilities and supporting scientific discoveries. Realizing this natural fact, computer scientists utilized computing powers in serving it. Computer science has highly contributed in serving
Figure III.3: Visualization disciplines hierarchy

all disciplines of science such as physics, chemistry and genomics. This is mainly through providing a variety of applications and tools that can be used to graphically represent theories and complex phenomena.

Using Computing facilities and power to construct visual representations of ideas and concepts across any field of science defines a new discipline categorized under computer science called Visualization. With the presence of the visualization field, computer science is now able to support people’s understanding and perception abilities by allowing them to observe information and to derive more information which could not be obtained by just looking at the actual data. Surprisingly, computer scientists made only little efforts in utilizing visualization to serve computer science. Visualization can play a great role in designing, implementing, and maintaining software and as a consequence, a new branch of visualization called software visualization has been defined. To our knowledge, the only book that is available to discuss in details the field of software visualization is by Diehl [41].

Visualization in computer science is classified as either scientific visualization or information visualization. Software visualization is categorized under the discipline of information visualization which processes abstract data, unlike scientific visualization which processes physical data, as shown in figure III.3. It is intended to visualize all aspects related to software development, such as algorithms, bug reports, development process, its evolution over time and even work distribution among developers. Finally, software visualization is aimed to provide visual means to easily comprehend
software systems and to improve the way developers talk about their systems which will increase the overall productivity [41].

After discussing the meaning of the software visualization discipline, the discussion in the next parts will provide a brief overview of the main contributions categorized under software visualization. There are mainly three categories of contributions. First, software visualization contributions for debugging applications in general. Second, software visualization in visualizing visualization systems. The final category is the contribution of software visualization in supporting software development process and human interaction during software system implementation.

III.2.1 Software visualization for supporting debugging

One well known visual debugging approach among all developers is the use of print statements to show results and debugging messages on the screen. However, this simple technique and many others do not provide enough support and do not fulfill developer’s need for smoothly analyzing and debugging large systems. As a consequence, lots of work has been done in developing a variety of software visualizations to different kinds of software and systems. Many of them were provided to assist in making debugging programs a less painful process on developers. Providing visual representations that reflect how the system works can highly contribute in quickly locating and fixing of bugs. Moreover, software visualizations are also used for systems performance analysis and evaluation which will help in detecting system bottlenecks and areas of improvement. In an attempt to reduce the difficulties that developers face in debugging C++ programs, Laffra and Ashock presented a visual debugger for Object-Oriented C++ programs [42]. This was among the first attempts to take advantage of visualization in building a friendly debugging environment. Their visual debugger is called HotWire and it was developed by IBM research center. The main idea of such visual debugger is to provide as many views as needed to find a bug.
There are two categories of visualizations provided. They are either static, provided by the HotWire or custom-built, defined by the developer. In static views, all classes and their instances are visualized. Moreover, all generated method calls and all messages that are passing in one frame are also shown. This is along with visualizing the interaction between individual instances and finally providing an analysis of all of these views. Custom-built views, on the other hand, are defined using special visual scripting language. When a specified model gets manipulated the corresponding view of the visual debugger will get updated accordingly.

Multi-threading systems are among the most complex systems when it comes to code tracing, maintaining and debugging. This is due to the fact that many concurrent events take place which cannot be derived by just tracing the source code, as correlation between software structure and system behavior at run time is not clear. To overcome this, Trumper et al. suggested a visualization tool that helps to reduce the lack of correlation between a system’s source code and the resulting behavior [43]. First, methods calls at runtime are recorded and then visualizations of multi-threading trace data are provided to help developers understand complex multi-threading systems.

Attention was also given to implement visualizations to assist in debugging distributed systems. Back and Dollner proposed an automated analysis visualization technique for better comprehension and debugging of distributed software systems [44]. First, communication data from system components is automatically gathered and then a visual heuristic layout is provided. Following those visualizations help developers to explore structural and behavioral information related to the system. Another work dedicated to visualize the behavior of mobile agents in a distributed system is suggested by Wong et al. [45]. A generic tool that can visually assist in debugging and evaluating mobile-object-based distributed programs was proposed. The suggested framework aims to improve the reliability of such programs by addressing
issues such as: how much time the mobile agent takes to prorogue from one node to another; where the mobile agent actually exists; detecting the mobile agent behavior over time and what possible faults can be detected in the network where the program operates.

One real challenge developers face is when they attempt to perform maintenance and re-engineering of existing systems to add new features as requested by the end users. Re-factoring software systems can sometimes be impossible, especially for legacy software where the code contains millions of lines with no proper documentation and original developers are not around anymore. To overcome these obstacles, Bohnet and Dollner proposed a prototype tool [46] that helps a programmer to quickly answer the questions: where a certain feature is located in the code and how is it being manipulated? This is accomplished through the use of an interactive multi-view visualization that enables users to explore the function call graph during the execution of a specific feature. After that, generated graphs are interpreted within the architecture of the system. This is along with providing dynamic metrics such as function execution time. Their visualizations help in having the developer completely
oriented during the analysis and exploration of the code. Another work aiming to utilize software visualization to support software re-engineering and maintenance was proposed by Trumper et al. [5]. They suggested decision-support visualization techniques that help in assessing program elements that can cause the need for future maintainability. Being able to detect the modules of the system that are probably going to generate maintenance overhead in the future, helps in reducing overall maintenance expenses. The visualization is presented using an easy to read circular bundle as shown in figure III.4.

Another side of software visualization focuses on building a friendly programming environment using flexible and portable tools which was proposed in the Pretty-Printing project [47]. The system provides color coding of the source code while programming.

### III.2.2 Visualization of visualization software

Visualization programs usually deal with huge data sizes, complex data structures and non-trivial algorithms. For example, a bug in a visualization system may only emerge when large data sets are investigated or after many iterations of computation. As a result, traditional debugging tools do not offer enough support which encouraged researchers to provide general guidelines and software visualizations to facilitate the debugging process.

Laramee offered valuable guidelines which are expected to help researchers and programmers during the development of visualization software [6]. The guidelines focus on the importance of exploiting the power of graphics and visualization to effectively debug and evaluate visualization systems. For example, one guideline emphasises the great value of using color-maps to classify resulted primitives so that unaccepted results can be easily detected. Figure III.5 shows an example of using color-maps in coloring primitives for the rendering output of a visualization system.
Another tip suggested the use of step functions and to incorporate that on the user interface so that the user can pause the execution at anytime to observe the results at a specific time or after a specific number of iterations. Those guidelines can be a source of inspiration to software visualization developers to get ideas on what needs to be visualized and reflected to visualize visualization systems.

An interesting visual debugger for visualization systems was presented by Crossno who designed a visualization tool of their particle visualization system [10]. The particle system is an application used to extract iso-surfaces from volumetric data. They presented system specific debugging visualization tools which can also be extended and applied across other visualization systems. Their work is based on coloring the particles based on specified attributes and viewing particles color changes in an animated way over successive iterations while the program is executed.

Moreover, a number of applications focused on providing visualizations of general algorithms to enhance the understanding and perception of how different algorithms work which in its turn will make debugging a more feasible process. One of these systems is the Balsa-II algorithm animation system [48]. The system provides multiple window views that can help in understanding the dynamic behavior of a given algorithm. It can be viewed as an extensible intercepter that has a graphical interface and
that is completely domain independent, i.e. it has no dependency on the visualized algorithm. An interesting feature provided by Balsa-II is the availability of scripting facility where user’s actions are recorded and the viewer of the script can interactively change parameters of the recorded video. More work on visualizing algorithms can be found in [49] [50]. In addition, geometric algorithms were given special attention by Tal and Dokin who presented a system called GASP, which is dedicated to visualize geometric algorithms from computational geometry [51]. To reduce the complexity involved with implementing geometric algorithms, GASP was proposed to allow the use of multiple visualization tools for such algorithms. Those visualization tools are provided to help in debugging and demonstrating geometric algorithms. Another framework called CoMeT was proposed by Wolfenbarger, et al. [52]. The framework was designed to provide a graphical debugging environment of their meshing algorithms. By visualizing particle positions and links between particles, the evolution of the mesh can be observed.

III.2.3 Visualization of the software development process

Software visualization is also concerned to design visualization tools to organize the software development process among developers. This is mainly intended to apply information visualization techniques to assist software engineering, scientists and project managers with analyzing the data and successfully developing the final system efficiently. Moreover, it is about using visualization techniques that provide an intuitive, time-series, interactive summary view of the social groups that form, evolve and vanish during the entire lifetime of the project. For example, Ogawa and Ma [53] presented a technique that visualizes the developers interaction during the software project evolution. Inspired by metro maps, software evolution storylines are provided. It has been proven that such representations provide more information about software development process than animated software histories. However, storylines lack the
Figure III.6: A screenshot of the StarGate visualization system. Circles represent developers. Circles colors represent the domain where the developers work and their sizes represent the amount of contribution each developer accomplished in a specific domain. Connections between developers are also visualized. Image is a courtesy [7].

scalability to analyze very large projects since the presented view will become very complex to analyze and comprehend. Another visualization techniques can visualize the history of commits in a software project. One technique was presented by Ogawa and Ma [54]. When a developer saves or commits changes made to the code, those changes are transferred to the central project repository. Their suggested visualization technique is called code-swarm. It binds between the developers and the files they changed when committing takes place. This is through a well presented animation, for example a committed file lights up and flies to the developer representation icon.

Some visualization techniques were also dedicated to serve open source software. Open source software projects have highly contributed in enriching the computer science field. Developers, project managers, scientists and software engineers massively communicate through electronic communication means. As a result, they present an opportunity for software visualization of which was proposed by Ogawa et al. [55]. They suggested a visualization technique that provides an interactive summary view of the social group of people who are working upon an open source software through
its entire lifetime. Their focus is on visualizing the evolving networks that exist in project mailing lists and provide an easy to analyze diagrams of the electronic interaction between different categories of developers. Another visualization technique worth mentioning was presented by Bohnet and Dollner [56] to monitor code quality and development activity. Their software maps can provide up-to-date information related to software development, system dynamics, software quality based on the user specifications. Those maps are easy to read and follow with little practice and can support the decision-making process. Another interesting visualization technique called StarGate was presented by Ma [7]. It focuses on visualizing developers’ activities as they are working on software systems. As developers formulate the center of their designs, they are grouped into cluster based on what areas of the file repository they are working upon. Connections between developers by monitoring e-mail communications are also visualized. Figure III.6 shows one screen of the StarGate system. More examples of using software visualization techniques in serving software development process can be found in [57] and [58].
Chapter IV

HVR Environment

The work on this thesis focuses on covering some software visualization aspects which are aimed to help in evaluating and understanding the distributed High Quality Volume Renderer (HVR). Thus, this chapter is dedicated to introduce the main functionality aspects related to HVR. Lots of those aspects are covered in detail in the earlier chapters, but here the focus will be on the specific technical choices of HVR and the goals that guided those choices. The version of HVR used during this thesis is the distributed GPU based HVR. In this chapter, distributed HVR motivation and goals are first discussed. Then, an overview of the distributed architecture is provided. After that, memory access and organization along with multi-resolution scheme using virtual octress are provided. Next, a discussion of volume distribution, rendering process and composition is available. The final part of this chapter introduces some information about the implementation environment using MPI and the asynchronous assignment of tasks among different threads. Information provided in this chapter was obtained through discussion sessions with members of the team who highly contributed upon developing HVR and also many of the technical information related to HVR can be found in [8][59].
IV.1 Distributed HVR Motivation and Goals

As discussed earlier [14], Electron Microscopy is one of the recent high resolution data acquisition techniques which produce extremely large volume data size. EM microscopy assists neuroscientists to examine very tiny structures of human brain tissues at a very high resolution. As consequence, building 3D volume visualizations for the EM data is important as this will help neuroscientists to interactively extract more information and induce more results easily from a well presented 3D image. This, in its turn, was one of the motivations behind designing the High Quality Volume Renderer. Moreover, scan results of EM have 3-5nm pixel resolutions and slice thickness of 30-50nm. And this results for volume size of hundreds of gigabytes or terabytes and can even grow up to reach hundreds of terabytes which is known as the petascale. Considering above information, it has become vital to provide an efficient distributed version of the High Quality Volume Renderer (HVR) that is interactively and quickly able to display volume portions as they are being requested. Another fact related to examining EM scan and which highly influences the HVR implementation choices is that scientists usually examine the tissue at a zoomed level and they only view the entire volume at the beginning. Thus, the goal of the distributed renderer of large volumes is not to display the entire volume as quickly as possible, but to interactively render subset of the volume as quickly as possible and according to the user request. Another influence on HVR implementation choices is the need to avoid pre-processing that requires the entire data volume, such as regular octrees. Pre-processing of data will cause a gap between visualization and data acquisition making it impossible to have an interactive environment. Thus, one goal of distributed HVR is to eliminate any pre-processing that involves the entire volume data and to avoid storage and processing of invisible data as much as possible. To summarize, the branch of HVR used to accomplish this thesis is a demand-driven volume renderer of high resolution EM data with implicit data load balancing and occlusion culling and
Figure IV.1: Distributed HVR networking architecture. Image is courtesy of [8]

distributed using cluster of GPUs.

**IV.2 Distributed Architecture Overview**

The Distributed HVR has a Client-Server networking architecture as shown in Figure IV.1. It uses a thin-client which delivers user requests received through a user interface to the server and displays the rendered volume images sent back from the server. The server, on the other side, constitutes of a number of GPU nodes, together called a cluster. They are responsible for volume rendering. One distinguished GPU node is known as the server node which is connected over TCP/IP sockets with the client and manages the communication between nodes of a cluster using MPI. Client-server connection is acceptable to be fairly slow, which makes remote connection very feasible to neuroscientists. However, the communication between cluster nodes on the server side is assumed to be fast enough to handle MPI communication. The server will manage the distributed parallel rendering of the volume among nodes of a cluster using MPI. To reduce the overhead of communication between nodes, each node maintains an independent memory hierarchy, as will be discussed later. This eliminates the
need to use MPI to distribute any actual volume data between nodes. Rendering is accomplished by single-pass ray-casting using CUDA. After compositing is performed on each node, the final image is sent back to the client for display.

IV.3 Multilevel out-of-core memory hierarchy

With the goal to minimize the communication overhead between different GPU nodes, a multilevel memory hierarchy is employed independently on each node. The actual data needed during ray-casting, data representing the current resolution level and culled against the view frustum, is located in the GPU cache. This is the first level of cache with the size of the actual working set size. A second larger cache level is the CPU cache which contains a superset of the working set. The aim of the second level is to minimize the need to read data from disk as it is more expensive. Figure IV.2 is an illustration of the HVR Multilevel cache hierarchy.
IV.4 Adaptation of a Multi-resolution Scheme Using Octrees

The resolution of a displayed rendered image is bounded by the resolution of the view or display screen. As a result, it is indeed efficient to limit the required amount of data and working set size to the screen or view resolution rather than the original very high volume resolution. Moreover, when the displayed volume has higher resolution than the screen, aliasing resulting from under-sampling will dramatically reduce the quality of the displayed volume. Given the two above reasons, using a multi-resolution scheme becomes a wise approach to efficiently display high resolution volumes on regular screens. Adapting the data resolution for rendering based on the zoom level and the display resolution best describes the multi-resolution scheme. This is mainly achieved by not accessing higher resolution data compared to the number of rays casted. Octrees are used to implement the Multi-resolution scheme.

Each level of the Octree represents a certain resolution level. The root represents the least resolution and it progressively increases so that leaves of the octree contain the original data. The level of octree is used based on the display resolution, trying to reach 1:1 voxel:pixel ratio. However, if no enough physical memory can handle the current display resolution a coarser octree level is used. For efficiency and to avoid pre-processing the entire octree levels and store them, HVR follows a virtual octree representation. It is called virtual because its nodes are mapped to a virtual memory space and only the physical address is calculated when the ray-caster needs to access that node. This means that the actual physical memory is only computed and filled with actual data on demand.
IV.5 Distributed Rendering and Composition

This section introduces the rendering pipeline of HVR. First, it discusses how the volume is distributed among nodes of a cluster using a novel implicit load balancing technique. Next, it explores the rendering process that is based on applying single pass ray-casting and applying it only for the data that represent the current resolution level. The final part briefly discusses the compositing stage that is based on direct send algorithm.

IV.5.1 Volume distribution and implicit load balancing

Load balancing is one of the most important aspects to be considered when designing distributed volume renderers. As discussed in section III.1.5, there are two main types of load balancing techniques. Load balancing can be either applied following static or dynamic approaches. Static load balancing has the advantage in that the work is distributed only once along the entire rendering process which eliminates the need to carry volume data across the network during rendering. However, static load
balancing algorithms lack the assurance of even work distribution across nodes of cluster compared to the dynamic load balancing techniques. Dynamic load balancing tries to overcome the deficiencies of static load balancing. However, it leads to a higher complexity of the rendering system and a higher communication overhead.

HVR follows a hybrid load balancing technique which aims to provide the advantages of both dynamic and static load balancing algorithms. The goals are to provide best utilization and even work size distribution among nodes with the least possible overhead. The entire volume is divided into large bricks following a 3D checkerboard distribution among GPU nodes of a cluster. Those are known as distribution units or distribution blocks and they can be for example of size $512^3$. Sort-Last algorithms, discussed in [III.1.1], are used to distribute the blocks among the nodes of the cluster. Unlike sort-first algorithms which divide the screen space between different nodes, sort-last algorithms divide the volume data among the nodes. The latter are more efficient for rendering large volumes because they do not require updating the entire working set in GPU cache when the user rotates the volume on the screen. However, sort-last algorithms add compositing step to the rendering pipeline where image parts are blended using compositing algorithms such as binary swap. Each of these distribution blocks is further divided into smaller blocks, for example of size $32^3$, across each node.

Large distribution blocks reduce the compositing overhead. Moreover, local data access for each node is provided. To further improve efficiency we introduce a second level of subdivision, which enables per-frame fine culling and update of the rendering cache textures memory on each node when needed so that each node can dynamically determine which sub-bricks can be eliminated from the rendering process. Following above load balancing technique, almost equal distribution of data, equal cache texture size, within the view frustum to the GPU nodes per-frame update is achieved. Figure [IV.3] is an illustration of assigning volume space distribution blocks to nodes of a
cluster following a 3D checkerboard pattern.

**IV.5.2 Distribution unit rendering**

Each GPU node has a list of distribution blocks assigned to it and are supposed to be rendered by this node. By accessing those distribution blocks in a front-to-back order, a single pass ray-casting algorithm is used to render each distribution block until all visible blocks within the view frustum are rendered. GPU based ray-casting is the core of the HVR rendering framework. The ray-caster accesses the data at the current resolution level which is managed by a 3D mipmap structure. In other words, no full ray-casting is performed but in each frame the current target 3D mipmap level is rendered instead. Before rendering starts, the distribution block in every node and with knowledge of their size and geometric position are culled against the view frustum.

In every ray-casting pass, information including cache misses and culling information are collected and used for optimizing the rendering process. Rendering a distribution unit consists of a number of steps, aiming to provide efficient rendering with minimum working set size. First, when a ray-caster requests for data that is marked as visible and not available in cache, a cache miss is reported and data is paged in accordingly by the cache manager. Thus, the ray-caster deals also with a list of cache misses during raycasting.

Second, the ray-caster will access a local culling information buffer which contains information about which sub-bricks are visible and which are not. Obviously, Invisible sub-bricks data are never read into the cache texture during this rendering pass. This is the first level of culling. A second level of culling is based on information received from other nodes during the exchange of rendered distribution blocks in the previous rendering ray-casting pass on other nodes. It results of skipping the rendering of entire distribution blocks as they are being fully occluded by other blocks. If the
transfer function is opaque which is likely most of the times then the number hidden distributed blocks increases and the working set size gets highly reduced. During the rendering loop the ray-caster constantly checks for occlusion culling information passed by from other nodes and adds it to its local knowledge. Occlusion culling is a key of performance enhancement of HVR. It helps in reducing the composition complexity and the amount of data exchanged between nodes. Moreover, as data is loaded into GPU cache and CPU only on demand, that is the ray-caster has marked data as visible, then occlusion culling contributes in loading less amount of data and thus reduce the working size. However, the version of HVR upon which the work of this thesis was implemented does not support occlusion culling, but it is mentioned here as it is an important aspect related to this visualization system.

IV.5.3 Composition

After having visible blocks within the view frustum rendered by the rendering nodes, composition takes place. Each node is responsible for compositing an image part of the final displayed image. Final image partitioning, i.e. distributing the composition tasks among the compositor nodes, takes place in screen-space. HVR screen-space is divided into equal size quads known as screen tiles. Every compositing node is responsible to perform the composition upon a number of screen tiles following a horizontal contiguous stripes pattern across those tiles. An advanced direct-send Algorithm is used for this purpose. HVR parallel compositing using direct-send algorithm is discussed in further details in V.3.1. Since communication is handled asynchronously, sending data to be used for composition interleaves with the rendering process. This avoids the need to have multi-layer composition steps as only one more exchange of image parts is needed after rendering and before final blending takes place. When a node renders a distribution block it checks the screen tiles of the screen space that were touched during the ray-casting and sends the rendered distribution block to the
compositor node of those tiles. However, if the bounding rectangle of those screen tiles is outside the screen space area then the rendered distribution block is never sent out to the compositor node. This not only saves processing time but also significantly reduces network bandwidth consumption.

IV.6 Overview of Multi-Threads Communication Using MPI

Volume rendering in HVR framework is performed on the server side, by distributing the volume among GPU nodes of cluster. Each node is responsible to handle part of the volume and they are designed to work as independent as possible from other nodes to reduce communication overhead between them. Within a node, work is handled by three different threads and they are communicating asynchronously. The following are the three different threads and the tasks assigned to each one of them.

- **Main Thread.** Handles sending and receiving of messages and data between nodes of a cluster using MPI. It also handles the client server communication.

- **Render Thread.** Handles per-frame rendering of the distribution blocks assigned to the MPI node and stores the results in a screen tiles buffers.

- **Composite Thread.** Handles compositing of a frame of data that were placed in the screen tile buffer by the rendering thread. After that, it performs blending and places the result in a screen tiles buffer. Information in this buffer are accumulated in the main server node and prepared to be sent back to the client where the image of the current frame is displayed.

Asynchronous communication allows interleaving the execution of tasks between threads within a node which serve the overall performance positively. However, it
Figure IV.4: Per-Node Threads Asynchronous Communication in Distributed HVR. The main thread is responsible for managing MPI sending and receiving messages between the nodes along with handling client/server communication issues. The render thread constantly checks if a new frame update has been received, if so it renders and stores results in buffer. The compositing thread: constantly checks if new screen tiles have arrived for compositing, composites, and stores results.
requires care when implemented to make sure that no data gets lost and that data is available when needed. Intermediate buffers between different threads are used to ensure reliable asynchronous communication between threads. Moreover, to avoid having threads running so much ahead, user request are placed in a FIFO buffer and it has a limited length. If the FIFO reached its maximum size, user input is discarded until space is made available. Figure IV.4 provides a detailed illustration of the job assigned to each thread and how they communicate.
Chapter V

HVR Functionality and Performance Visualization

Distributed HVR is a rich environment involving a number of valuable techniques and algorithms to efficiently render large volumes of high resolution, as discussed in IV. It provides a great chance for researchers to experiment with options related to volume rendering. However, this in turn induces complexity which induces the need to provide proper means to make debugging and performance tracking a feasible process. For these reasons and as a start, we work on this thesis to provide a number of visualizations that are aimed to reflect some aspects related to how distributed HVR operates. We also provide means of performance analysis by collecting and computing a number of parameters on the fly while the system is in operation. Results are presented in a user friendly way and displayed by expanding the same end user interface that is used to display the final rendered images. Moreover, they are flexible as they allow the user to choose which parameters to inspect or visualize during runtime using the same GUI. Working upon software visualization for the HVR system is aimed to provide dynamic and immediate performance results which help in detecting bottlenecks in the rendering pipeline. Moreover, with the means of
software visualization a number of functionality aspects has become visually present and are displayed as a reflection of the current state of the system. As a result, the implemented visualizations are aimed to assist in making HVR concepts easier to follow as providing graphical representation of its functionality are expected to enhance the learning ability of the users and eventually help them to debug and expand the current visualization system.

After discussions with members of the team who are currently working on HVR, to best serve their needs, the decision was made to visualize a number of HVR functionality and performance related concepts. Those are the different sections of this chapter. Implementation details, their added value to HVR and results are presented along with discussions related to the system functionality.

**V.1 Demand-Driven Data Back-End Visualization**

For visualizing the functionality and current status of the demand-driven data back-end, we aim on providing a graphical representation of the process of sending requests to the data back-end during the rendering process on each GPU node. Before discussing the implementation details and the results of this representation, HVR demand-driven data back-end is first discussed to clearly view its role within the context of the rendering process.

**V.1.1 HVR demand-driven data back-end overview**

HVR aims to provide an efficient and an interactive volume rendering environment of high resolution volumes. The goal is not to display the entire rendered volume but to quickly enable the user to traverse volume parts as they are requested. This is based on the fact that neuroscientists tend to observe zoomed-in part of very high resolution volumes and not the volume in its entirety. Computing and processing the
Figure V.1: Demand-driven data back-end within the context of the HVR rendering pipeline

entire volume in a pre-processing step, does not meet HVR goals since it involves lots of unnecessary computations and this can form an obstacle in achieving an interactive environment. Instead, data is computed on demand. A virtual octree that is mapped to a virtual 3D memory space is populated. When the virtual memory of a node in the virtual octree is accessed by the ray-caster because it is a visible node, the actual 3D data needed for visualization is computed by the demand-driven data back-end. This means that data that is never viewed is never computed. The input to the data back-end is a collection of 2D image tiles generated by the electron microscopes. Each of these 2D tiles has its own affine transformation matrix which makes it possible for the data back-end to combine tiles of different scales, rotations and resolutions. The data back-end uses this attached information to build a 2D mipmap pyramid for each input tile. The latter is done in a pre-processing step, but it is much faster and feasible than computing a 3d mipmap.

The actual EM slices or their corresponding mipmap form the input to the data-back-end and they are of size of $512^2$. The output is a volume block of size $512 \times 512$.
x 32 voxels for each 2D input which is cached internally in the data-back-end and the HVR reads only a $32^3$ part of it used by the ray-casting. Figure V.1 places the data back-end in the context of HVR rendering pipeline. As shown in the diagram, data is first requested by the ray-caster from the virtual octree structure, memory management details related to managing the virtual octree is not discussed here. Then, the request propagates to the data back-end where tiles are either fetched already in the local cache or need to be read back from the hard-disk before construction. When 3D-image data are reconstructed, they are made available through either CPU or GPU cache.
Figure V.2: Demand-driven data back-end requests visualization. There is a total of two nodes, shown in different colors (a) Requested blocks streaming at different resolution levels (b) Requested blocks visualization along with the actual volume being rendered

V.1.2 Implementation

Implementing the demand-driven data back-end visualization is based on collecting data during the rendering process and more specifically when the requests for data blocks are sent to the data back-end. Those requests are collected, processed, transmitted and finally visualized on a view of HVR client interface. Requests are sent to the data back-end when a ray-caster ray marks a sample of data as visible and
attempts to access the data of that sample using the virtual address of a node at certain resolution level of the virtual octree. After that, the request propagates to the data back-end to obtain the actual data and make it available locally in the GPU texture cache.

A volume bounding box, discussed in section V.6, is first implemented to make it possible to judge where the block of the data requested is located corresponding to the entire volume. The entire virtual volume is split into $32^3$ voxels tiles. Identifying the requested tiles is the first step in visualizing them. Thus, every tile is assigned a unique tile ID number. With the knowledge of the requested tile ID and the current octree level of resolution, tile origin 3D-coordinates are obtained. The left and right borders dimensions of the requested block are enough to render a bounding box representing that tile which is the goal of this visualization. They are calculated based on the current tile dimensions and origin of the requested tile. After that, the border coordinates are normalized according to the volume dimension at the current octree resolution level.

Using MPI, a point-to-point communication is established and dedicated to send the normalized requested tiles borders of every node to the master node of the server. The master node, in its turn, waits until all nodes have sent their information of the requested tiles and then sends it over TCP/IP communication lines to the client.

On the client side, blocks are displayed as they are being received and each displayed block is color coded based on the node that initiated it. The size of the requested block varies based on the current resolution level. If the current resolution level changes, old requests to the data back-end are discarded to display the blocks at the new resolution. Figure V.2(a) is a collection of snapshots of the output result of visualizing the stream of tiles requests to the data back-end at different resolution or zoomed levels. The higher the resolution the smaller is the requested tile dimensions. Figure V.2(b) shows a snapshot of the visualization result aligned with the actual
volume being rendered.

V.1.3 Results

The very first outcome of the data back-end requests visualization is that it has become possible to graphically represent the stream of tile requests to the data back-end as they are being requested. This can be used in explaining the concept of the data-back in a clearer way. It can also support in easier detection of the bugs during this stage of the rendering pipeline. However, during the implementation we managed to detect a hidden bug in the system, as it did not affect the end result of volume rendering. We found that it is possible for two different tile IDs to be mapped to the same block origin dimensions so the same request of data is being made to two different tiles. Moreover, we found that multiple requests are being sent to the data back-end of the same block of data but by different nodes of a cluster. This could have a negative effect on performance as there are duplicated requests. Referring to Figure V.2(b), there are four different requests shown by two GPU nodes. The truth is that there are more hidden and duplicated other four requests. After finding this problem, we recommended that it should be eliminated as it harms the performance and increases the communication bandwidth.

V.2 Distribution Unit Rendering Visualization

Distributed volume rendering has become one of the recent hot research topics. This is because of the increased demand to visualize large volumes on-the-fly without any encountered delay to the end user. Volume rendering involves performing homogenous computation on large data sizes which together form an ideal environment for parallel distributed implementation using GPUs. On major stage during the rendering that is interesting to be visualized, is the results of local ray-casting by every GPU node.
First, unit distribution on HVR is discussed. After, the visualization of the local rendering implementation and results are presented.

V.2.1 Distribution unit rendering overview

Distributed rendering on HVR provides a novel technique for applying implicit load balancing for equal work size distribution among GPU nodes and for the assurance of affine culling. The volume is first statically distributed among nodes following a 3D checkerboard pattern using last-sort algorithm. Each node becomes locally responsible to perform volume rendering for a number of volume blocks known as distribution blocks, for example of size $512^3$ voxels. The distribution unit size with respect to the original volume changes depending on the current resolution level. However, it always stays the same with respect to voxels that are rendered. For efficient culling, those distribution blocks are further divided into smaller blocks ($32^3$) so the ray-caster can possibly discard those smaller blocks from rendering if they are invisible. For each frame, a single pass ray-casting is applied on each of the distribution blocks in a front-to-back order until all assigned blocks within the view frustum are rendered locally by that node. Cache misses and culling updates are also considered during ray-casting. Rendering is done locally and independently on each node. This briefly presents the rendering stage of the whole rendering pipeline and which we are interested to visualize as part of HVR software visualization results presented in this thesis. Information about implicit load balancing, volume distribution and rendering can be found in details in section IV.5.

V.2.2 Implementation

First, it is important to discuss how local rendering is accomplished on each node and how the rendered data blocks are communicated across nodes of the cluster. Local rendering on each node is done independently and communication is managed locally
by using multiple threads that are exchanging information asynchronously between them. There are three different threads. First, the main thread that is responsible for MPI communication and receiving user requests. Next, the rendering thread that is responsible for rendering and finally the compositing thread is responsible for image composition. The main thread sends the request to the rendering thread to start rendering. When a node finishes the rendering of part of the volume assigned to it, the rendering thread places the rendered tiles result in a local buffer and starts rendering other parts of the volume. Meanwhile, the main thread on that node constantly checks for new data in the rendering buffer and whenever data is available, it sends the rendered tiles using MPI to the corresponding node that is responsible for compositing. The compositing thread on that node will handle the composition of that part of the image. Compositing results are also placed in another local buffer and the main thread is responsible for delivering those ready composited tiles to the server node for final image copying. The main server, in its turn, will deliver the final image to the client for display. Asynchronous communication between these threads allows interleaving the execution of tasks between threads within a node which serve
the overall performance positively. However, when implementing the visualization of the result of local rendering on each node, we faced a challenge in collecting the locally rendered tiles due to the irregularity encountered with asynchronous communication. Threads communication is discussed in details in section IV.6 and illustration IV.4.

After discussing how communication is carried out locally, a broader view must be provided to build a clearer vision of how rendered data blocks are carried out in the context of distributed HVR. Figure V.3 shows an illustration of how volume data are communicated across nodes using MPI until delivered to the server where they are sent back to the client for display. An MPI communication is established to send rendered tiles for another node for compositing. Next, another MPI communication is established to send composited tiles from the compositor node to the server for final image blending and then the final image is delivered to the client over TCP/IP communication lines.

In implementing our visualization of the local rendering results, we have added another new MPI communicator that works in alignment with the other already established communications. This communicator takes the rendered tiles produced
by the rendering nodes and instead of sending them to the compositor nodes as discussed earlier, it sends those tiles directly to the server, as shown in the illustration provided in figure V.4. As rendering is carried out asynchronously, data is never sent back to the client unless the server has all the rendered tiles of the current frame as all rendered tiles are accumulated in the master node before sending them to the client. Along with the actual rendered tiles, their information including their frame id, rendering node id, distribution block id and screen tile id numbers are also sent back to the client. Those IDs are needed to classify the rendered tiles in the client side, so that it is possible to display the rendered tiles of each distribution blocks separately. This is along with the origin coordinates of the 2D displayed image tile required for rendering. On the client side, rendered tiles of a given frame are grouped based on their distribution block id and displayed following a multi-texturing scheme where each texture represents a group of tiles belonging to the same distribution block. Figure V.5 is a snapshot of this visualization result and its details are discussed in the coming part.

V.2.3 Results

The output view of the rendering process visualization is presented in the snapshot shown in figure V.5. What is provided is the rendering result of each distribution block viewed separately before compositing and blending the final image. And the complete final image is shown below the distributed view for comparison. The output shows that the volume is divided into a total of eight distribution blocks, each of size $512^3$ and the rendering is distributed among two GPU nodes. This view is updated for every new frame arrival. With the availability of such visualization, debugging the rendering done by each node, mainly the output of ray-casting has become feasible. As an example of using this view for debugging is when the final displayed image is appearing to be incorrect, missing parts of the volume for example. The HVR developer can
Figure V.5: Distributed rendering visualization. It shows a separated rendering view of eight distribution blocks which are together forming the entire volume view of the current frame. Distribution block 0 is on the top left corner. Below the separated rendering view, the actual rendered volume is displayed.
examine the rendering on each node and for each distribution block separately using this view. If the view was producing correct results then the problem in the final image is not because of the local rendering on each node. It could be because of the data communication or compositing and final image blending. However, if the output of this debugging view was incorrect this means that there is a problem with the local rendering and the user can easily specify in which distribution block it resides. Moreover, if occlusion information is added then we will be able to see which areas have been occluded already and which not. This narrows down the area of the code which needs to be explored and debugged and thus this will enhance and facilitates the maintenance of the HVR.

V.3 Distribution Unit Composition Visualization

A compositing step is added to the rendering pipeline when a distributed volume renderer uses sort-last parallel rendering algorithm for distributing the volume blocks among nodes of a cluster. The compositor is responsible of putting together, in parallel, partial images produced by the sort-last algorithm to formulate the final image to be displayed. First, an overview of parallel composition on HVR is presented and then the visualization implementation details and results are introduced.
V.3.1 Parallel composition overview

After rendering image parts in parallel by nodes of a distributed system, composition is needed to blend and build the final image that is to be sent back to the client and displayed. HVR adapts a parallel direct-send algorithm to perform the composition in parallel following a very simple approach [20] [9]. Composition is carried out in parallel to avoid the bottleneck induced from using serial composition. In serial composition, each processor will generate a part of the whole image which is to be sent to the master where blending the images take place using depth or visibility tests. However, with larger image sizes, bandwidth requirements to transfer those image parts can form a serious bottleneck.

Parallel direct-send is based on having the nodes that are responsible of rendering part of the image to be responsible to composite other parts of the image as well. This simply means that once a node finishes the rendering of part of an image, it sends the result for another node for compositing while itself handling composition tasks, as shown in figure V.3. HVR follows a hybrid parallel rendering approach, discussed in section III.1.2. This is because of the difference between rendering and composition in distributing parts of the image among the nodes. The sort-last algorithm divides the volume data in object space, during rendering, while partial images used in composition are resulted from screen space decomposition. However, both object space and screen space can be partitioned into tiles and load balancing is an important criteria to be considered when distributing image parts in both spaces.

Decomposition of tasks among nodes in screen space can take various shapes. Image can be partitioned into horizontal stripes, vertical stripes, contiguous or interleaved stripes, or rectangular. The choice between these different shape patterns has its implications regarding the influence on achieving an equal load balancing, ease of implementation, and minimum space and time complexities.

HVR has made the choice to split the final image for parallel composition using
Figure V.7: Direct-send final image partitioning into n horizontal stripes to handle composition in parallel across n nodes.

horizontal contiguous stripes. This is because horizontal stripes tend to mimic the underlying memory structure which makes implementation a straightforward task, using simple array structures. Moreover, horizontal stripes reduce memory consumption and space complexity by eliminating the need to have any intermediate storage or complex decomposition routines. Although other partitioning shapes can possibly achieve less time complexity, but when rendering very large scale volumes, space complexity is more important to be considered and minimized than time complexity.

Horizontal contiguous stripes can also result in a satisfiable load balancing among the nodes. The actual local composition is carried out using GPU alpha blending along with visibility and depths tests. To make this possible, images are processed as textures in the GPU memory and blended together in the right order, V.6.

Horizontal stripes are calculated by dividing the height of the final image by the number of available compositing nodes. It is organized such that the first stripe on top is assigned to node number 0 and goes on until the final bottom stripe is assigned to node number n-1 as illustrated in figure V.7. This will have the advantage of making copy operations and memory transfers easier to accomplish. After splitting, every node becomes responsible to compositing a specific stripe. When a node finishes the rendering of part of an image using single pass ray-casting, it sends local tiles to the corresponding compositor node which is dedicated to composite that stripe of an
image. And, a node receives all the rendered tiles it needs for the local composition of its assigned stripe pieces. All-to-all MPI communication is used to send the result of rendering to other nodes for compositing. In the worst case, a node can send its local rendered tiles to all other nodes as they are belonging to all final image stripes. Finally, one time copying step is needed by the master node after receiving all composited partial image tiles from other nodes by means of MPI communication and then sent to the client using TCP/IP sockets for communication.

V.3.2 Implementation

After composition is accomplished in parallel by the compositing nodes for a given frame, composited screen tiles resulted, each of size $64^2$, along with their information are sent to the master node using MPI communication. Final blending to produce the final image takes place in the master node. In our implementation of the per-processor composition visualization, we created an MPI communicator which is responsible to send the composited screen tiles generated by all the available compositing nodes to the master node every frame. The reason we chose to dedicate a separate MPI communicator and not to use the one available already, is that we want to distinguish in our implementation between implementing visualization of the functionality of HVR and the actual volume rendering pipeline implementation used to render the volume. In the master node, composited tiles are accumulated and are sent to the client with their information once all composited tiles of the current frame have been available at the master node. At the client side, composited screen tiles are classified based on the compositor node id and then with the use of the screen tile id and the tile dimensions the tiles are rendered to the screen.
Figure V.8: Distributed composition visualization. It shows the composited image parts produced by each node separately and the composition was carried out across a total of two nodes. The user can input a valid node id number to view the result of composition by that node.
V.3.3 Results

The output view of implementing the per-distribution unit compositing visualization is presented in the snapshot shown in figure V.8. The view shows the composition results by each compositor node separately. A user can specify a valid node id number and then the composition results produced by that node is displayed. This view is provided along with the visualization of the final image and the per-distribution unit rendering visualization output an it is updated whenever a new frame is arrived to the client.

With the availability of this visualization, debugging the composition step, mainly the result after applying parallel direct-send algorithm, in the rendering pipeline has become feasible. An HVR developer can refer to this view when doubting that the reason of not being able to display the final volume correctly on the screen is because that the composition of the final image is not being done correctly. Just like the per-distribution unit rendering visualization, this view provide a graphical representation of a very critical and important step of the rendering pipeline. It is now possible before being involved in debugging the code lines to judge if the problem can possibly resides during the composition step or not. If the per-node compositing results are not correct then the problem has been narrowed down to be located in parts of the code responsible for compositing. This, in its turn, will enhance the developer availability to locate and successfully resolve problems in rendering. Finally, this visualization can be used as an appealing and a well formed illustration and reflection of how distributed HVR performs per-node composition.

V.4 Screen-Space Image Partitioning Visualization

One interesting visualization suggested by members of the HVR team, is to view how the screen-space is represented and manipulated in the context of HVR. It divides
the screen-space into 2D tiles of equal size, currently of size $64^2$ pixels. Those tiles exist only in screen area where the volume exist. As a result, the screen-space is not rectangular as it only covers the area surrounding the volume. This has an optimization advantage since it means that there are fewer screen tiles to manipulate and communicate. This will be clear in our screen-space visualization implementation.

For composition to be done in parallel using direct-send algorithm, the final image is divided and distributed among nodes in screen-space, discussed in details in section V.3.1. The final image is divided horizontally by the number of available processors and each horizontal stripe is assigned to a different node for composition and this is something to visualized in this view as well.

V.4.1 Implementation

When image composition is accomplished on each node, composited screen tiles, those of size $64^2$, and its associated information are sent to the master node using MPI. The master processor is responsible to perform the final stage of image blending before sending the blended final image to the client for display for every frame. In our implementation, composited screen tiles information are sent to the client and used
to render the screen-space visualization. Tiles information needed are tiles frame id, compositing node id, and screen tile id numbers. This is along with the origin coordinates of the 2D tile and their dimensions which are needed to render those tiles.

The visualization output is represented by the snapshot shown in figure V.9. Screen-space is only considered for screen area where the volume is displayed. The division of screen-space into smaller equal size screen-tiles is also shown. Moreover, The figure shows the horizontal contiguous stripes the final image is divided into to distribute the composition task between the nodes, in the snapshot there are a total of two composition nodes. This is done by coloring each screen tile according to which node was responsible to perform composition on it.

V.4.2 Results

The screen-space image partitioning visualization provides such simple looking view that can highly contribute in reflecting important functionality aspects related to HVR. It helps in making screen-space division and parallel composition self-explanatory, refer to figure V.9, which supports new HVR developers to build a clear vision regarding how the image is partitioned for composition and how the screen-space is culled around the final image. In addition, it dynamically reflects the current state of the system as the view gets updated for every frame. Moreover, the screen-space visualization provides a valuable debugging support in judging the correctness and load balancing involved in image partitioning and finally compositing.

V.5 Volume Distribution Visualization

Another visualization presented is designed to show the volume distribution pattern in object space among the rendering processors of the distributed HVR. A sort-last algorithm is used for volume distribution. Before rendering starts locally, separately,
Figure V.10: Visualization of the volume distribution in object space following a 3D checkerboard pattern. The visualization provided is for a volume of $512^3$ resolution that is divided into a total of eight distribution blocks each of size $256^3$.

and independently on each node, volume blocks are distributed among available rendering nodes. The entire volume is divided into large distribution units following a 3D checkerboard pattern and distributed between the processors of the system. At this stage, static volume distribution takes place without the need to examine if the volume parts are occluded or whether they are within the view frustum area or not. It is locally on each node where each distribution unit is further divided into smaller sub-bricks so that fine culling is feasible. Moreover, based on the available local visibility information, each node can dynamically determine whether to render a given sub-brick and makes it available in its GPU texture cache or not. A rendering node can sometimes skip the rendering of an entire distribution unit if it is available outside the view frustum area or if it is fully occluded by other distribution units. Following this multi-level division approach is aimed to provide the best equal work size distribution between the rendering processors without slowing down the overall system performance. Full discussion and evaluation of the volume distribution and the adapted load balancing techniques of the HVR is provided in section IV.5.1.
Figure V.11: Visualization of the volume distribution in object space for a volume of 1008 x 1065 x 101 resolution that is divided into a total of two distribution blocks distributed among two nodes. The image presents a situation of load imbalancing as the work size is not equally distributed between the two processing units.

V.5.1 Implementation

In implementing the visualization of the volume distribution, information including the number of the distribution units available, the size of the distribution unit, its original coordinates, and its node id are collected and sent to the master node using a special MPI communicator that is created for this purpose. Those parameters along with the original volume dimensions are sent from the master node to the client. On the client side, a bounding box for each distribution unit is visualized after calculating its normalized coordinates and after being culled against the entire volume dimensions. Each displayed distribution unit is color coded based on its node id in order to clearly visualize the 3D checkerboard distribution pattern. Figure V.10 shows the visualization of the volume division in object space. Moreover, the visualization is designed to be updated every time the volume gets re-distributed between the nodes.

V.5.2 Results

The volume distribution visualization provides an easy to follow graphical representation of the volume partitioning in object space. This assists in judging whether the division provides an equal distribution of the volume blocks between the nodes or not.
Moreover, users can benefit from such a well presented visualization in illustrating how volume distribution takes place. Using this visualization, we tested the volume distribution of different volume data. Figure V.10 represents an ideal representation of the 3D checkerboard distribution of a $512^3$ volume data size that is divided into eight distribution blocks, each of size $256^3$. By just looking at the view, it is possible to judge that there is an equal distribution of volume data among the two available rendering nodes.

However, when examining a volume of size $1008 \times 1065 \times 101$, the resulting object space portioning was not equal between the nodes which leads to an unacceptable state of load imbalance. As shown in figure V.11, the volume is divided into just a total of two distribution blocks and the work size on node 0 is much higher than the one assigned to node 1. In analyzing this situation, we found that originally the distribution units were given a size of $1024^3$ which is larger than the actual volume dimensions. As a result, distribution units were culled against the volume dimensions which resulted in such unbalanced distribution of the volume data. We concluded that it is important to choose the correct dimensions and numbers of the distribution units based on the original volume dimensions to obtain a satisfiable static volume distribution results.

**V.6 Volume Bounding Visualization**

In continuation with our attempt to visualize different functionality aspects related to the distributed HVR, a visualization of the volume bounding box and the volume view frustum are provided. Volume bounding box is important for visibility culling determination. It is commonly used among many rendering algorithms, especially in determining rays intersections in a ray-casting algorithm. This is because it is faster and easier to test for a ray intersection against the bounding box than testing against
Figure V.12: The Visualization result of the volume bounding box and view frustum the rendered object itself. If a given ray in ray-casting does not intersect the volume bounding box then it cannot intersect the volume itself. View frustum, on the other hand, is also valuable for visibility culling as volume parts outside the view frustum area are never rendered or even fetched to the GPU cache in the HVR. This has the advantage of enhancing the overall performance of the distributed HVR. Moreover, in a very high resolution volume data, such as EM, scientists tend to explore only parts of the volume and not the entire volume. As a consequence, the view frustum is important in determining which parts of the volume to render and which parts to discard.

In implementing the bounding box view, we first obtained the normalized world coordinates ranges and the volume three-dimensional slice distances and use them to determine the normalized coordinates of the bounding box. Similarly, the view frustum normalized coordinates are obtained, they are the coordinates that are used for obtaining the volume clipping planes in view space. In our visualization, shown in figure V.12, we rendered the volume bounding box and the view frustum in a separate view from the volume display screen. This visualization can be viewed in alignment with the displayed volume and gets updated whenever a user applies zooming in or out to the displayed volume.
V.7 Performance Visualization

As we are aiming to provide a number of visualizations that are able to tackle various aspects related to the distributed HVR, the system performance is one of those vital concepts to be considered. The HVR is designed to provide an interactive environment such that users can easily manipulate a displayed volume. As a result, providing means of performance analysis is highly valuable to assist in evaluating the overall system speed, memory and resources utilization, and locating where bottlenecks are possibly induced. Our performance visualizations are built on top of the same graphical interface where the volume rendering results are displayed. This has the advantage to enable simultaneous view of the performance and the rendering results and thus enabling the user to observe how the performance of the system is influenced by manipulating the displayed volume. In addition to their ability to reflect the immediate current state of the system, our performance visualizations are designed to be flexible and dynamic so that users can choose among different evaluation parameters at runtime. In the coming parts, we will discuss the different performance evaluation parameters which we choose to provide in this thesis and the implementation details.

V.7.1 Performance evaluation parameters

In distributed parallel rendering systems, utilizing the available processing power and resources across all stages of the rendering pipeline can guarantee providing the highest possible performance. As a result, in our performance visualizations we attempt to tackle different measuring parameters to evaluate different stages of the rendering pipeline of the distributed HVR. The following are the performance parameters and their definitions in the context of the HVR.

- Frame rates at different stages along the rendering pipeline form one category of the selected performance parameters. Time is very important to consider
especially that the HVR is expected to provide an overall acceptable rendering speed to allow for efficient interactivity with the end users on the client side. The following are the different time measurements that we are interested to visualize and examine in this thesis.

1. Rendering time is the time consumed in per-frame local rendering on each rendering node of the distributed system. It includes the time each node spends to apply the single pass ray-casting to render the volume distribution blocks assigned to it. More details about per-processor local rendering can be found in section IV.5.2.

2. Blending time is the time each node takes to composit the rendered tiles assigned to it for a given frame. Each node is responsible to composite specific parts of the final displayed image based on screen-space division. In other words, it is the time needed to download the rendered tiles to a texture and blend it. Composition is discussed in details in section IV.5.3.

3. Copy back time measures the time needed for final image blending at the master node for a given frame. Image tiles that are composited locally on each node are sent to the master node for a final copying stage to formulate the final image before sending it over to the client.

4. Networking time measures the time consumed in asynchronous communication between different processing nodes in the server using MPI for every frame.

5. Overall time measures how fast rendered images arrive to the client per-frame. More specifically, it is the time from sending the client requests to the server until the final image is displayed back on the client side.

- GPU cache usage is another category of our performance visualization parameters. Data blocks are fetched from disk, or a shared file system of a distributed
system, to CPU and then moved from CPU to GPU cache upon request in preparation for rendering. GPU cache usage percentage determines how much of the total GPU cache is being consumed at every frame during the rendering process. It is important to compare the percentage of cache usage across the processing units of the system as balancing the usage of GPU is one of the goals of distributed rendering systems.

- Number of blocks download is another performance parameter which we are assessing in our performance visualizations. During ray-casting, the renderer can detect which visible data blocks are missing, i.e. which blocks are not available in GPU yet. Those are called cache misses in the context of HVR. As a background process, those blocks are scheduled to be fetched from disk to CPU and then to GPU so that a number of blocks can be downloaded to GPU at each frame. The number of those downloaded blocks is what this parameter measures. The higher the number of blocks download, the slower the renderer is. As a result, tracking this parameter is important in analyzing the performance of the system.

V.7.2 Implementation

In implementing the graphs of the various calculated frame rates, we first start by placing a number of system timers to measure the time in seconds that each stage lasts during the rendering process. Given that the distributed HVR follows a multi-
threading scheme, it is important to consider that other threads do not overwrite the
system timers placed in a specific thread to avoid collecting inaccurate time measure-
ments. Using the resulted time values of different parameters along the rendering
pipeline, the number of frames per second can be calculated on each node. The num-
ber of blocks download is calculated by placing a counter that increases whenever a
data block is requested to be downloaded from CPU to GPU. The last parameter
we examine is the percentage of cache usage which is automatically reported by the
ray-caster during rendering. After placing the appropriate means to read those per-
formance parameters values in every frame, each processing unit sends the obtained
values to the master node using MPI. The master node then sends those performance
parameters to the client over TCP/IP communication lines.

On the client side, multiple graphs are created which represent the various frame
rates values, GPU cache usage percentage and the number of blocks download as new
frames arrive to the client. Those graphs have the advantage of making it possible
to see the history of values in previous frames and compare them with each other,
rather than the regular print statements which are able to show the result of the
Figure V.16: Networking frame-rates visualization graph.

Figure V.17: Overall Frame-rates visualization graph.

Figure V.18: Simultaneous frame-rates graphs visualization.
Figure V.19: Graph visualization of the number of downloaded data blocks to GPU. Those graphs are created using Qt. Qt provides a Qwt Widget library which is able to facilitate the plotting of two-dimensional graphs with unlimited number of plot items that can be displayed on its canvas. Our visualizations are made flexible by enabling the users to choose from a drop down menu which parameter to investigate at runtime. Moreover, we provided the possibility of investigating multiple graphs at the same time. Figures V.13, V.14, V.15, V.16, and V.17 show the graphs that represent the rendering frame rates, the blending frame rates, the copy back frame rates, and the networking frame rates correspondingly. Figure V.18 shows a simultaneous view of a total of four different frame-rates parameters where the user can choose the parameters from a drop down menu and can increase or decrease the number of simultaneous plots by clicking on the “+” and “-” bottoms. Figure V.19 shows the visualization of the number of blocks downloaded to GPU. Moreover, a graph visualization of the block cache usage percentage is shown in figure V.20. In order to provide a variety of visualizations, figure V.21 shows the number of the current downloaded blocks to GPU and provides a progress bar that represent the block cache usage percentage of the current frame.

As shown on the presented graphs, there is a separate curve for representing the values on each processing unit. This adds an extra dimension of valuable comparisons and meaningful analysis especially that distributed rendering systems aim to provide
V.8 Performance Analysis

In order to assess the value and the validity of the performance visualizations provided in this thesis, this section provides an overall analysis and evaluation of some per-
<table>
<thead>
<tr>
<th>Data set</th>
<th>Resolution</th>
<th>Rendering Frame Rate</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DS1</td>
<td>1,008 x 1,065 x 101</td>
<td>437.72 fps</td>
<td>635.48 fps</td>
</tr>
<tr>
<td>DS2</td>
<td>512 x 512 x 512</td>
<td>41.42 fps</td>
<td>55.39 fps</td>
</tr>
<tr>
<td>DS3</td>
<td>14,176 x 10,592 x 308</td>
<td>15.09 fps</td>
<td>41.34 fps</td>
</tr>
</tbody>
</table>

Table V.1: The obtained average rendering frame rates measured in frames per second for various data sets (fps).

formance aspects related to the distributed HVR, as suggested by the implemented performance views. Throughout the analysis, the volume display area has a fixed size of $512^2$ pixels to assure consistent results. Moreover, in our analysis, we examined the performance parameters values across three different EM volume data sets of different resolutions to experiment the influence of the volume resolution on the HVR performance and to provide a rich analysis environment. In addition, during the experiments, the distributed HVR was running on a total of two GPU processing nodes. The last thing related to the experiments environment is that the readings obtained are average values from a total of ten frames and the collection of these values starts only when the HVR reaches a stable state to obtain meaningful results.

Table V.1 shows the obtained average rendering frame rates on every rendering node across different data sets. From that table and the corresponding graph shown in figure V.22, it is clear that the rendering speed is almost dependent on the resolution of the volume being rendered. Moreover, the rendering speed on the master node is slower than the rendering speed on the other slave node. In investigating the reason for this, we found that along with assigning rendering tasks to the master node, it is also responsible to handle the client/server communication issues. As a result, this induces an extra overhead on the master node. Based on that, we recommend eliminating the master node from the rendering process as it can result in slowing down the entire rendering process. However, the rendering frame rates values indicate an acceptable performance of the distributed HVR during the rendering stage.
Figure V.22: A graph representation of the average rendering frame rates for various data sets.

Moreover, table V.2 provides rather a wider range of analysis that involves a number of major stages during the rendering process. The tables show the average time in milliseconds that is consumed during rendering, blending, and copy back processes. This is along with the time needed to perform asynchronous MPI communication between nodes of a cluster. From the table and its graph representation V.2, we observed that individual stages of the rendering pipeline are of a very good satisfiable speed along different volume data sets with different resolutions. However, the performance of the HVR system executed upon the same number of processors is influenced by the volume size. The higher the resolution of a rendered data set, the lower the frame rates across different steps of the volume rendering pipeline. It is also obvious that through the different data sets, networking communication consumes the maximum time and thus HVR developer should consider dedicating efforts to enhance the speed of MPI communications between the nodes. Finally, another fact that can be driven from the table provided is that the master processor, due to the encountered client/server communication management overhead, consumes more time in comparison with the other slave node across all the measured stages. This in its turn, re-enforces our previous recommendation which stated that the master node should not be handling any rendering tasks to avoid inducing bottlenecks during the rendering pipeline.
Table V.2: The obtained average times at different stages of the rendering process measured in milliseconds (ms) across three data sets.

![Table V.2: The obtained average times at different stages of the rendering process measured in milliseconds (ms) across three data sets.](image)

Figure V.23: A graph representation of the average times consumed at different stages of the rendering process for the Alignment data set.

![Figure V.23: A graph representation of the average times consumed at different stages of the rendering process for the Alignment data set.](image)
<table>
<thead>
<tr>
<th>Data set</th>
<th>Resolution</th>
<th>Overall frame rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS1</td>
<td>1,008 x 1,065 x 101</td>
<td>10.21 fps</td>
</tr>
<tr>
<td>DS2</td>
<td>512 x 512 x 512</td>
<td>4.21 fps</td>
</tr>
<tr>
<td>DS3</td>
<td>14,176 x 10,592 x 308</td>
<td>2.92 fps</td>
</tr>
</tbody>
</table>

Table V.3: The obtained average overall frame rates measured in frames per second for various data sets (fps).

After evaluating the individual stages of the rendering pipeline, it is vital to examine the overall system speed by measuring the average overall frame rates across different data sets as shown in table V.3. The overall time measures the time needed to display an image on the client side for a given frame. Given that the individual stages of the rendering pipeline are fast enough as discussed earlier, we predicted that the overall time will be just as good. However, by comparing the frame rates values in table V.3 and in table V.1, the speed of image display is much slower than the speed of rendering. This raised the question that what can possibly cause such huge gap between the speed of the overall system and the speeds of the individual stages during rendering? Based on our investigation, we concluded that the MPI communication overhead is the reason behind the overall system delay. The networking time cannot return the actual time that is spent in MPI communications as it is only limited to measure how long calls to MPI routines last and with an asynchronous communication scheme, calls return very quickly so the actual communication time is not detected. As a result, improving the communication speed is a key to enhance the overall distributed HVR rendering speed.

In continuation with the analysis of the distributed HVR performance, table shows the average GPU cache usage percentage across different data sets resolutions. In assessing the GPU caches efficiency and utilization, one important aspect to examine is whether there is an equal distributed loads between the different GPUs in a distributed system or not. According to the measurements in the table, there is almost an equal usage percentage of the GPU cache between the available rendering nodes.
Moreover, the table shows that there is almost no dependency between the cache usage percentage and the resolution of the volume. However, according to table V.5 and its corresponding graph in figure V.24, there is a clear dependency between the cache usage percentage and the transparency level of a volume. We measured the average GPU cache usage percentage across three different transparent functions ranging from being completely opaque to being very transparent. We found that as the transparency level of a displayed volume increases, the GPU cache usage percentage increases. The final parameter provided in our performance visualization is the number of the downloaded blocks from CPU to GPU as requested during the rendering process. The speed of rendering is negatively influenced as the number of the data blocks downloaded to GPU increases. As a result, it is important to keep the number of downloaded blocks as small as possible so that the rendering speed is maintained. According to Figure V.19, the number of downloaded blocks never exceeds 32 blocks per-frame which is a relatively small number that cannot overwhelm the rendering process.
Table V.5: The obtained average GPU cache usage percentage on each node for various transfer functions. While TF1 represents a completely opaque transfer function, TF2 represents a semi-transparent one and TF3 represents a more transparent transfer function.

<table>
<thead>
<tr>
<th>Transfer function</th>
<th>Block cache usage %</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Node 0</td>
</tr>
<tr>
<td>TF1</td>
<td>0.61 %</td>
</tr>
<tr>
<td>TF2</td>
<td>1.60 %</td>
</tr>
<tr>
<td>TF3</td>
<td>4.90 %</td>
</tr>
</tbody>
</table>

Figure V.24: A graph representation of the obtained average GPU cache usage percentage on each node for various transfer functions. The block cache usage percentage increases as the transparency level of a transfer function increases.
Chapter VI

Summary and General Conclusions

The goal of this thesis is to provide a number of graphical representations that reflect various functionality and performance aspects related to the distributed HVR system. The work presented is aimed to formulate the start towards designing a friendly debugging environment that utilizes the power of computer graphics to support developers during the development and the maintenance processes of the system. The provided visualizations expand the current HVR interface and are designed to be flexible and dynamic so that they are able to reflect the current state of the system. They are selected to visualize the different stages of the parallel rendering pipeline.

Being able to examine the processing result of a specific rendering stage, helps to quickly judge whether the processing is carried out correctly or not and supports quickly locating bugs and areas of improvement. Before this work, obtaining incorrect final rendering results, would leave the developer with a long list of causes which will require investigating and debugging a huge amount of code by following the traditional debugging methods. Our visualizations include visualizing the stream of tile requests to the data back-end, the rendering result of each distribution before compositing and blending, and composition results by each compositor node. This is along with screen space image partitioning visualization, volume data object space distribution
visualization, and volume bounding visualization.

In this work, we also provide means of performance analysis through a number of performance visualizations that reflect various performance measurements of the current state of the system. In order to evaluate the validity of the performance visualizations provided, we conducted a performance analysis of the system. As a result of the analysis, we managed to evaluate the system performance throughout the entire rendering process. There are two major issues that contribute to a slower rendering speed. The first issue is that the master node cannot perform rendering with the same speed as other nodes which can slow down the rendering stage. This is because the master node is also responsible to handle the client/server communication management. The recommended solution for this issue is not to assign rendering tasks to the master node. The second issue is the encountered communication overhead. MPI communication consumes the majority of the time during the rendering process. As a result, enhancing the communication speed is predicted to significantly improve the overall distributed rendering speed.
Bibliography/References


http://doi.acm.org/10.1145/147130.147141

[37] K.-L. Ma, “Parallel volume ray-casting for unstructured-grid data on
distributed-memory architectures,” in Proceedings of the IEEE symposium on

[38] L. Kwan-Ma and T. W. Crockett, “A scalable parallel cell-projection volume

using the shear-warp factorization,” in Proceedings of the IEEE symposium on

[40] M. B. Amin, A. Grama, and V. Singh, “Fast volume rendering using
an efficient, scalable parallel formulation of the shear-warp algorithm,”
in Proceedings of the IEEE symposium on Parallel rendering, ser. PRS
http://doi.acm.org/10.1145/218327.218330

[41] S. Diehl, Software Visualization: Visualizing the Structure, Behaviour, and Evo-


[43] J. Trümper, J. Bohnet, and J. Döllner, “Understanding complex multi-
threaded software systems by using trace visualization,” in Proceedings of
the ACM Symposium on Software Visualization, 2010, pp. 133–142. [On-


[59] “Interactive volume exploration of petascale microscopy data streams using a visualization-driven virtual memory approach.”