Volume Visualization and Compositing on Large-Scale Displays Using Handheld Touchscreen Interaction

Thesis by

Cristhopper Jacobo Armenta Gastelum

In Partial Fulfillment of the Requirements for the Degree of Master of Science

King Abdullah University of Science and Technology
Thuwal, Kingdom of Saudi Arabia
July 27th, 2011
The undersigned approve the thesis of Cristhopper Jacobo Armenta Gastelum

Alyn Rockwood
Committee Member

Niloy Mitra
Committee Member

Markus Hadwiger
Committee Chair

King Abdullah University of Science and Technology
2011
Volume Visualization and Compositing on Large-Scale Displays Using Handheld Touchscreen Interaction

Cristhopper Jacobo Armenta Gastelum

Advances in the physical sciences have progressively delivered ever increasing, already extremely large data sets to be analyzed. High performance volume rendering has become critical to the scientists for a better understanding of the massive amounts of data to be visualized. Cluster based rendering systems have become the base line to achieve the power and flexibility required to perform such task. Furthermore, display arrays have become the most suitable solution to display these data sets at their natural size and resolution which can be critical for human perception and evaluation.

The work in this thesis aims at improving the scalability and usability of volume rendering systems that target visualization on display arrays. The first part deals with improving the performance by introducing the implementations of two parallel compositing algorithms for volume rendering: direct send and binary swap. The High quality Volume Rendering (HVR) framework has been extended to accommodate parallel compositing where previously only serial compositing was possible. The preliminary results show improvements in the compositing times for direct send even
for a small number of processors. Unfortunately, the results of binary swap exhibit a negative behavior. This is due to the naive use of the graphics hardware blending mechanism. The expensive transfers account for the lengthy compositing times.

The second part targets the development of scalable and intuitive interaction mechanisms. It introduces the development of a new client application for multitouch tablet devices, like the Apple iPad. The main goal is to provide the HVR framework, that has been extended to use tiled displays, a more intuitive and portable interaction mechanism that can get advantage of the new environment. The previous client is a PC application for the typical desktop settings that use a mouse and keyboard as sources of interaction. The current implementation of the client lets the user steer and change the opacity transfer function of the visualization via simple multitouch gestures. Nonetheless, the user can freely move around, engage into discussion with other users and easily pass the tablet around for others to use. Before, this was not possible with the same ease of use. Ultimately, the collaborative possibilities are many and extremely interesting to explore.
ACKNOWLEDGEMENTS

I would like to dedicate this thesis to the people for whom I feel greatly thankful. Without their help, this thesis would not have been possible.

First, I would like to sincerely thank my thesis supervisors: Johanna Beyer, whose enthusiasm, patience, guidance and good humor never let me down and encouraged me during the difficult times; and Markus Hadwiger, who believed and supported me when I first decided to start this project.

I am also thankful to the people who helped me achieve this work: Ronell Sicat and Thomas Höllt. They were there always willing to help at anytime. I wish you all the best in your PhDs. Additionally, I thank my fellow students from the GMSV who made the journey more pleasant with their chat, laughs and great environment.

Finally, I would like to dedicate this thesis to my family and friends. They have been key for me to keep coming back for more. Their enthusiasm during the difficult times have lifted me up countless. I would not have made this far without them. They are always motivating me to pursue greater goals.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Examination Committee Approvals Form</td>
<td>2</td>
</tr>
<tr>
<td>Copyright page</td>
<td>3</td>
</tr>
<tr>
<td>Abstract</td>
<td>4</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>6</td>
</tr>
<tr>
<td>List of Abbreviations</td>
<td>10</td>
</tr>
<tr>
<td>List of Figures</td>
<td>11</td>
</tr>
<tr>
<td>List of Tables</td>
<td>14</td>
</tr>
<tr>
<td><strong>I Introduction</strong></td>
<td>15</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>16</td>
</tr>
<tr>
<td>1.2 The compositing bottleneck</td>
<td>17</td>
</tr>
<tr>
<td>1.3 The display wall interaction challenge</td>
<td>18</td>
</tr>
<tr>
<td>1.4 Document Organization</td>
<td>19</td>
</tr>
<tr>
<td><strong>II Fundamentals</strong></td>
<td>21</td>
</tr>
<tr>
<td>2.1 Volume data visualization</td>
<td>21</td>
</tr>
<tr>
<td>2.1.1 Volume Rendering</td>
<td>22</td>
</tr>
<tr>
<td>2.1.2 Light-transport physical model</td>
<td>23</td>
</tr>
<tr>
<td>2.1.3 Transfer functions</td>
<td>25</td>
</tr>
<tr>
<td>2.1.4 Data acquisition for Volume Rendering</td>
<td>26</td>
</tr>
<tr>
<td>2.1.5 Volume Rendering Methods</td>
<td>27</td>
</tr>
<tr>
<td>2.2 Basic Ray casting</td>
<td>28</td>
</tr>
<tr>
<td>2.2.1 Empty space skipping and early ray termination</td>
<td>29</td>
</tr>
<tr>
<td>2.2.2 Texture based ray casting</td>
<td>30</td>
</tr>
<tr>
<td>2.2.3 Ray casting in programmable pipeline</td>
<td>31</td>
</tr>
<tr>
<td>2.2.4 Advanced Applications and Techniques</td>
<td>33</td>
</tr>
<tr>
<td><strong>III State of the art</strong></td>
<td>35</td>
</tr>
<tr>
<td>3.1 Parallel Rendering</td>
<td>35</td>
</tr>
<tr>
<td>3.1.1 A memory architecture classification</td>
<td>36</td>
</tr>
</tbody>
</table>
3.1.2 The sorting classification
3.1.3 Sort-first systems
3.1.4 Sort-last systems
3.1.5 Hybrid algorithms
3.1.6 Parallel rendering on clusters
3.1.7 Load imbalance
3.1.8 Load balancing techniques
3.2 Volume Rendering of large data
3.2.1 Brickling
3.2.2 Multiresolution techniques
3.2.3 Octrees
3.2.4 Multilevel cache and out-of-core techniques
3.3 Display Arrays
3.3.1 Visualization on display arrays
3.3.2 Generic display array frameworks
3.3.3 Custom display array solutions
3.3.4 Interaction and input mechanisms
3.3.5 The interaction challenge
3.3.6 Speech
3.3.7 Tracking and indirect Interaction
3.3.8 Gestures and direct interaction
3.3.9 Hand Gestures
3.3.10 Touch interfaces

IV The HVR Framework
4.1 Multilevel out-of-core memory hierarchy
4.2 Display-aware multi-resolution scheme
4.3 The rendering process
4.4 HVR distributed architecture

V Parallel Compositing on HVR
5.1 The serial compositing bottleneck
5.2 Direct Send
5.2.1 Algorithm discussion
5.2.2 Implementation details
5.3 Binary Swap
5.3.1 Algorithm discussion
5.3.2 Implementation details
5.4 Experiments
5.5 Discussion
5.6 Chapter Summary and Conclusions
5.7 Future work
## VI Interaction mechanisms for HVR

6.1 The HVR user interface ........................................... 87
6.2 The HVR Client ....................................................... 89
  6.2.1 The client development minimum requirements .......... 91
  6.2.2 Application Design .......................................... 93
  6.2.3 Network Layer ............................................... 93
  6.2.4 TCP Socket Connectivity ................................... 94
  6.2.5 The Client-Server Protocol ................................ 95
  6.2.6 Display Layer ............................................... 96
  6.2.7 Low resolution images ..................................... 98
  6.2.8 Transfer function interface ................................ 98
6.3 Interactivity tests ................................................. 99
  6.3.1 System configuration ....................................... 100
  6.3.2 Results ....................................................... 100
6.4 Chapter Summary and Conclusion ................................. 102
6.5 Future Work ......................................................... 103
6.6 Photo gallery ....................................................... 106

## VII Summary and General Conclusions ............................ 109

References .................................................................. 111
# LIST OF ABBREVIATIONS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1D</td>
<td>One dimensional</td>
</tr>
<tr>
<td>2D</td>
<td>Two dimensional</td>
</tr>
<tr>
<td>3D</td>
<td>Three dimensional</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>BSD</td>
<td>Berkeley Software Distribution</td>
</tr>
<tr>
<td>CGLX</td>
<td>Cross-Platform Cluster Graphics</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CT</td>
<td>Computerized Tomography</td>
</tr>
<tr>
<td>CUDA</td>
<td>Compute Unified Device Architecture</td>
</tr>
<tr>
<td>DVR</td>
<td>Direct Volume Rendering</td>
</tr>
<tr>
<td>EM</td>
<td>Electron Microscopy</td>
</tr>
<tr>
<td>FPS</td>
<td>Frames Per Second</td>
</tr>
<tr>
<td>GB</td>
<td>Gigabyte</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>HCI</td>
<td>Human Computer Interaction</td>
</tr>
<tr>
<td>HVR</td>
<td>High quality Volume Rendering</td>
</tr>
<tr>
<td>I/O</td>
<td>Input / Output</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>LCD</td>
<td>Liquid Crystal Display</td>
</tr>
<tr>
<td>MPI</td>
<td>Message Passing Interface</td>
</tr>
<tr>
<td>MRI</td>
<td>Magnetic Resonance Imaging</td>
</tr>
<tr>
<td>OpenGL</td>
<td>Open Graphics Library</td>
</tr>
<tr>
<td>PC</td>
<td>Personal Computer</td>
</tr>
<tr>
<td>PET</td>
<td>Positron Emission Tomography</td>
</tr>
<tr>
<td>RGB(A)</td>
<td>Red Green Blue (Alpha)</td>
</tr>
<tr>
<td>SAGE</td>
<td>Scalable Adaptive Graphics Environment</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction Multiple Data</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface</td>
</tr>
<tr>
<td>VTK</td>
<td>Visualization Toolkit</td>
</tr>
<tr>
<td>WiFi</td>
<td>Wireless Fidelity</td>
</tr>
</tbody>
</table>
LIST OF ILLUSTRATIONS

1.1 The client application can input multitouch gestures to steer the HVR visualization on the display wall .......................... 16

2.1 Images rendered from CT data ........................................ 22
2.2 The model including light emission and absorption is usually used in DVR. Image is courtesy of [20] ......................... 23
2.3 Different transfer functions enable different optical properties in the data. They can be used to highlight the areas of interest. Image is courtesy of [71] .......................... 25
2.4 According to empty space skipping, only the gray boxes in the picture contain the volume. All other pieces of data are skipped. Image is courtesy of [20] .......................... 30
2.5 Plane proxy geometries used for texture based rendering. The ones in the left are aligned according to the viewport, while the ones in the right are axis-aligned. Image is courtesy of [70] .......................... 31
2.6 In GPU ray casting, the front and back faces of a cube are used to encode the 3D coordinates and calculate the ray direction. Courtesy of [40] .......................... 32
2.7 A screenshot from a fly-through visualization. Image is courtesy of [75] .......................... 34

3.1 The sort-first and sort-last schemes according to Molnar et al. [55] .......................... 38
3.2 The binary swap compositing for 4 processors. At every level, the tile is split between two processors. This method ensures all processors are used throughout the algorithm execution [47] .......................... 43
3.3 A volume decomposed into bricks extracted from [20] .......................... 51
3.4 Four levels of detail in a 1D data set. Higher resolution is used for areas closer to the regions of interest. Image is courtesy of [85] .......................... 52
3.5 An octree is used to encode different resolutions of the same data set. Image is courtesy of [20] .......................... 53

4.1 The HVR multi-level cache hierarchy has two levels: one on texture the GPU memory and one in the CPU memory. Image is courtesy of [4] .......................... 68
4.2 The HVR architecture courtesy of [4] .......................... 70
5.1 Direct send compositing using three processors. The first three images depict the state of the rendering buffers with the partial image tiles. The final composited image is shown at the rightmost. Image is courtesy of [19].

5.2 Direct send image decomposition. The row width is determined by dividing the height of the image over the number of processes.

5.3 The image shows the entire binary swap algorithm process at every level of the tree for 4 nodes. The arrows show how the binary communication is done between two processors at any time. As the tree traversal advances further, nodes tend to be farther away but the size of the image decreases the required communication. Level 3 shows the final image gathering.

5.4 The compositing order vector is used to compute the tree on the fly by calculating the corresponding exchanging block ID at any time. Next, the block ID is searched in the global node-block distribution vector to find the node owner of the block and return this for the binary communication to take place.

5.5 Compositing times for the different viewports. Direct send shows the performance gains expected, whereas binary swap's performance is penalized because of the expensive blending operation done on the GPU.

5.6 Overall times for the different viewports. The differences between the domain decomposition implementation are part of this graph. Rectangular decomposition is more expensive and is expected to further penalize performance for larger systems.

6.1 An overview of HVR sort-first decomposition. Image tiles are split into the working processes for independent rendering and compositing.

6.2 The desktop client setup in front of the display wall.

6.3 Diagram of the previous HVR configuration. The client connects to the server via TCP sockets. On the server, there are two types of processes executing in different machines: rendering and display. The latter are directly connected to the display tiles.

6.4 Screenshots from both client system user interfaces. The desktop client has lots of different interface controls to really manipulate the visualization on the server.

6.5 The application design is functionally split into three software layers: main, network and display. Each layer is in charge of providing specific functionality to the system. Touches are perceived on the display layer, transformed in the main layer and sent to the server using the network layer.

6.6 The client communicates with the server head node via WiFi using the network protocol described.
6.7 The client display modes. In 6.7a the client only shows a wireframe box to accelerate the process and still provide some hint on the state of the visualization. In 6.7b the client displays down sampled images received from the server.

6.8 The opacity transfer function can be changed by double tapping and then dragging two fingers over the screen. The screen space maps the x and y axes to the density and opacity values in the data set. The corresponding threshold functions are shown at the bottom of the figure.

6.9 Screenshots of the low resolution images of the various datasets running on the iPad simulator.

6.10 Screenshots of both visualizations: from the iPad client and the display wall. The images on the iPad were zoomed in using the built-in functionality.

6.11 Screenshots of the low resolution images of the various datasets running on the iPad simulator.
LIST OF TABLES

6.1 The obtained frame rates measured in frames per second (fps) . . . . 101
Chapter I

Introduction

The main goal of this thesis consists of improving the performance and usability of volume rendering systems that target visualization on display arrays. It extends the High quality Volume Renderer (HVR) framework, a tool that enables the visualization of large volumetric data. The first part consists of the implementation of parallel compositing algorithms. Our goal is to reduce the compositing time and make the system scalable for larger systems. The second part targets the development of scalable and intuitive interaction mechanisms. It introduces the development of a new client application for tablet devices like the Apple iPad. The goal is to provide a portable and intuitive user interface for the framework executing in tiled display systems. The portability and interactivity possibilities of this new client allow the users to freely move within the wireless internet range and steer the visualization using multi-finger gestures. It is also possible to change visualization parameters like the opacity transfer function. Figure [1.1] shows the system executing on the tiled display with the new client as the user interface.
1.1 Motivation

The goal of volume visualization is to help scientists, engineers and field experts to interactively explore and navigate datasets obtained by different imaging techniques like Computed Tomography (CT), Magnetic Resonance Imaging (MRI) or Electron Microscopy (EM) among many others. In the past, various methods were devised to visualize the intrinsics of volumetric data and produce realistic high quality images. Nonetheless, the methods developed are computationally intensive and resource demanding. Consequently, an important amount of effort has been put into building systems that can help to visualize very large datasets at interactive frame rates. Over the last two decades, high performance volume rendering has been defined by the extensive use of modern parallel computer architectures and by the creation of many different strategies to achieve the best performance.

Alongside, larger dataset resolutions have far surpassed common desktop displays.
A more recent line of investigation comprises efficient and transparent methods for taking the visualization systems output to display arrays. It is highly desirable to visualize the data at their full resolution and natural size for this might result critical to perception and evaluation. For more than a decade, tiled-displays have been built and studied. Challenges include flexibility, efficiency and intuitive user interaction mechanisms. More appropriate and intuitive user interfaces have been the subject of a vast amount of research. Devices ranging from laser pointers to gyroscopes and accelerometers found in modern mobile phones have been explored in the search of new interaction metaphors for the ultra high resolution displays. With the commercial success of smart-phones, multitouch capable devices have become a technology that is both available and affordable to everyone. As a consequence, rich user interfaces, driven by increased usability, has made these devices a very interesting alternative for intuitive interaction in visualization systems.

1.2 The compositing bottleneck

As data set sizes grow larger, parallel systems need to be revised to keep the fast pace of data. Bigger parallel systems have been deployed and explored. The sort-last parallel rendering approach consists on distributing the rendering work by dividing the data among the processors. Each one gets in charge of producing an image tile from the corresponding smaller subvolume. Depending on the current view point, some pieces of the volume might end up at the back occluded by the front pieces. Consequently, the images produced will need to be composited in the right order to achieve the correct final image. This compositing stage has been the subject of a considerable amount of research [47, 79, 63, 32]. When the number of processors increases, compositing the final image becomes a major bottleneck. For systems to scale with the processor count, the compositing work has to be distributed. Various
strategies can be found in the literature to solve this bottleneck.

Distributed HVR follows a sort-last approach. However, before this work, no parallel compositing capabilities were given to it. The goal in the first part of this thesis is to investigate parallel compositing strategies and provide HVR with a better performance compositing scheme. Therefore, two different parallel compositing techniques following different domain decomposition strategies were explored, implemented and tested. Performance comparisons and discussion conclude this part of the present document.

1.3 The display wall interaction challenge

The second part of this thesis entails a different line of development for the HVR framework. This time the visualization output has been taken to tiled displays. However, these systems have different capabilities and requirements than regular desktop systems. Evidence in the literature demonstrates how desktop interaction metaphors tend to break given the different environment settings: increased resolution, larger form factor and inherently distributed architecture. A considerable amount of research has been put into finding the most suitable user interface mechanisms that are both scalable and intuitive. The same interaction challenges found in the literature \cite{25, 34, 2} are applicable to this version of volume renderer. The goal here is to create a more appropriate interaction mechanism for the new environment. Given that HVR user interface is handled by a remote thin client, the objective is translated into the development of a new client application that can utilize the display wall setting resources more effectively. The purpose of the new client is to serve as a remote user interface for configuring and steering the produced visualization.
1.4 Document Organization

The remainder of the document is organized in the following way: Chapter II introduces the underlying fundamental concepts that serve as a basis for the work in this thesis. It starts by establishing the volume rendering problem and then continues introducing transfer functions and common data acquisition methods. It concludes with a brief summary of the historical evolution of ray casting and the acceleration techniques developed in the earlier years. Right after, Chapter III presents the current state of the art by introducing the parallel rendering problem. It describes current parallel systems aimed at rendering massive volumetric data along with the use of various strategies. Later in the chapter, the different approaches to take parallel rendering systems to high resolution display arrays are presented and discussed. It concludes by discussing the various interaction mechanisms explored for the visualization on tiled displays.

Chapter IV presents the HVR framework: a high quality volume renderer system that comprises a variety of methods and techniques to visualize volumetric data at interactive rates. The key points described include the techniques implemented for visualizing large data and its current distributed architecture. This chapter serves as the base line for the following ones, since the two projects build upon some given state of the framework at the starting point of development. Nonetheless, the framework features presented here are applicable to both development lines.

The following Chapters, V and VI, form the main part of this work. Chapter V introduces sort-last HVR. It then proceeds to describe the compositing stage bottleneck as reported in the literature. The work here consists of the investigation and implementation of two well known parallel compositing algorithms: direct send and binary swap. The algorithms and their implementations are further discussed before presenting the results of the various experiments conducted. The chapter concludes
by summarizing and providing an overview of possible future work.

Next, Chapter VI starts by describing the version of HVR working on tiled displays. It then presents a discussion on the challenges these systems pose to the traditional desktop interaction mechanisms. These arguments are used to motivate the development of a client application for the new display wall environment. Later it discusses the minimum functional requirements and presents the application design comprising three different layers: main, network and display. These layers are described in depth before presenting the interactivity tests performed on the system. The chapter concludes with a summary and a discussion about possible future work.

Finally, Chapter VII presents the general conclusions for this thesis by briefly stating the main objectives and summarizing the two projects.
Chapter II

Fundamentals

This chapter introduces the fundamental concepts of volume rendering. It starts by discussing the different requirements that make volumetric data visualization distinct from geometry visualization. It introduces important concepts like the light transport physical model, transfer functions, data acquisition methods and the classes of existing volume rendering techniques. Special attention is paid to ray casting and the various optimization strategies developed to further improve the efficiency of the algorithm.

2.1 Volume data visualization

Volumetric data, in contrast to surface or geometry rendering, have different visualization needs. While surfaces might suffice to picture the shape of a volume, they work under the implicit, possibly inaccurate assumption, that the volume consists of thin surfaces surrounded by air. Nevertheless, very often the interesting phenomena to observe lie in the interior of the data, where multiple components might interface in irregular, subtle and hard to predict patterns. As a result, the local variations of light emission and absorption might get lost. These features, however, are hard to visualize if only surfaces, intrinsically two-dimensional, are derived from the explicit three-dimensional shaped space of volumetric data [17]. Figure 2.1 illustrates two
2.1.1 Volume Rendering

Volume rendering is an area that developed within Computer Graphics. It describes techniques for generating images from discretized functions sampled in 3D. Volume data are acquired by measurement or generated by numerical simulation. Examples include medical data obtained by different techniques like Computed Tomography (CT), Magnetic Resonance Imaging (MRI) or Electronic Microscopy (EM). Other examples include computational fluid dynamics, seismography, and molecular biology amongst many others [20].
2.1.2 Light-transport physical model

Volume rendering aims to provide systematic ways to visually represent the physical properties of the light interaction with the participating medium. Light transport is computed from these properties and then an image is produced. In fact, mathematical expressions have been derived to model and compute the light transfer for image synthesis. The so-called optical models attempt to provide a physically-realistic framework based on geometric optics [20]. Volumetric data visualization techniques require a model that accurately portray how the light is generated, reflected, scattered and occluded by the volume particles [52]. The light transport equation in differential form is as follows:

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

(II.1)

where \( I \) represents the intensity and \( \rho \) the attenuation caused by absorption and out-scattering. Due to the computational intensity of equation (II.1) many different simplified models can be used. Each model provides a different level of photorealism depending on the light transfer equation terms involved. Intuitively, the more terms included result in a more physically accurate evaluation performed. The following are brief descriptions of the simplified models. More insight and in-depth explanation can be found in [52] and [20].

(a) Absorption only. The simplest model assumes that all particles of the partici-
pating medium are cold and perfectly black so that they can only absorb all the
intercepted light. No light is emitted or scattered. In this way, light attenuates
along a ray from its source across the material at some ratio. This reduces the
amount of incoming light that effectively exits the volume after traversal.

(b) Emission only. This model assumes that the volume consists of glowing gas that
is completely transparent, it does not absorb light, but instead it emits light only.
Light can also be emitted by reflection of external illumination, but for simplicity,
this model accounts only for the light coming from the material particles, and
disregards the light coming from other sources. Any other light interaction is
ignored. In the limit as the particle size or number density approaches zero,
while the emission goes to infinity, absorption can be neglected.

(c) Absorption plus emission. The premise to this model is based on assumptions
taken in the previous two. The particles in the material can both absorb and
emit light. This is, naturally, a more realistic approach to the physical light
transfer. Therefore, the terms accounting for both absorption and emission of
light are included in the model. This is the usual mode utilized in Direct Volume
Rendering (DVR), which is later defined in Section 2.1.5. Figure 2.2 shows the
emission and absorption of the light along the ray.

(d) Single scattering and shadowing. The assumption here includes the scattering of
light caused by reflection of external light sources. These sources can occlude the
light and cast shadows onto the volume. However, the single scattering model
only accounts for one reflection from the illumination ray to the observer, in spite
of the possibly multiple light bounces actually happening.

(e) Multiple scattering. This model attempts to evaluate the complete light transfer
terms described before: absorption, emission and scattering. In the need to in-
volve the multiple interacting light sources along the multiple scattering events,
calculations to evaluate this expression represent a very computationally intensive task that can result into prohibitive execution times. Consequently, it can potentially restrict the degree of applicability to the many different interactive volume rendering scenarios.

2.1.3 Transfer functions

The optical properties, emission, absorption and scattering, just explained in section 2.1.2, need to be known at every point within the volume for solving the light-transport equation. However, the data scalar values available do not provide natural ways to get these coefficients. Alternatively, optical properties are assigned using arbitrary mappings called transfer functions. Finding appropriate transfer functions to highlight the interesting volume physical properties is also known as classification [20]. There is an extensive variety of possible transfer functions. Although, they can be possibly extracted from the volume or from the images [37], they become a critical parameter to produce meaningful and intelligible volume rendering given such flexibility. Figure 2.3 illustrates how different transfer functions can show different optical properties in the dataset.

An opacity transfer function is often the simplest case. Its domain and range span the scalar dataset and opacity values correspondingly. Naturally, important features should be given high opacity values and vice-versa. In this way, they are
not hidden behind the not so interesting parts. Opacity transfer functions can be extended to include color. In fact, any optical property can be encoded in the range of a transfer function. Generalizations can be done by increasing the dimensionality of the domain. Consequently, multi-dimensional properties like density gradients can be directly represented. Furthermore, multi-dimensional transfer functions can also be the result of the generalization of the data to account for multi-variate instead of simple scalar values. Thus, multi-dimensional derivatives, like Jacobians, can be measured and directly represented in the transfer function. Nonetheless, all this flexibility and power come in a complexity price. Higher dimensionality, however, results in harder and more complex ways to edit the transfer functions appropriately. More in-depth descriptions can be found in the work by Engel et al. [20].

Given its critical role in meaningful visualization, there is a considerable amount of effort to find less tedious and simpler but powerful strategies to set transfer functions. RezkSalama et al. [71] study the complexity of transfer function visual parameters while attempting to provide the non-expert users with more usable and friendly methods to specify these. Through the addition of an abstraction level, transfer function models can be designed by visualization experts and still be used intuitively by the non-experts.

2.1.4 Data acquisition for Volume Rendering

Data come from different sources depending on the area of application. Scalar data is especially important in many scientific fields and is generally obtained by using imaging techniques like the ones presented next. More details can be found in the work by Engel et al. [20].

- CT (Computerized Tomography) is a technology based on x-rays cast onto the patient’s body. The radiation effectively exiting the body is recorded, as it is attenuated accordingly depending on the different types of tissue encountered on
the way. The process is repeated from different directions to record the attenuation of the rays along the area of interest. Subsequently, a three-dimensional image is reconstructed. This technique can effectively detect bone tissue with good precision.

- MRI (Magnetic Resonance Imaging) reconstructs the three-dimensional image in a different way. The spins of atomic nuclei are aligned with a strong magnetic field. MRI scanners use an excitation pulse to perturb these spins. Radiation is emitted when each spin realigns with the outer magnetic spin. Then, radiation is recorded. To identify different materials, different nuclei are used. It is especially suitable for soft tissue like brain imaging.

- EM (Electron Microscopy) uses an electron beam to create images with super high resolutions of up 3-5 nanometers and slice distances of 25-30 nanometers. The higher resolution is due to the much smaller size of electrons compared to visible photons. The datasets produced reach from hundreds of gigabytes to terabytes. It is considered a destructive method describing the damage caused while preparing the samples to be captured [3].

- PET (Positron Emission Tomography) produces three-dimensional images by detecting gamma rays indirectly emitted by a positron-emitting radionuclide injected into the blood circulation. Images are reconstructed from the concentration within the body of the substance via computer analysis. It is often used along with CT or MRI for multi-modal co-registration [82].

2.1.5 Volume Rendering Methods

Historically, there have been many proposed strategies to perform volume visualization. These strategies can be classified into indirect and direct methods depending on the way they attempt to produce the image.
(a) Indirect Volume Rendering.

These methods are characterized by an attempt to approximate the volume using intermediate geometric representations. Many of the geometric surface approximations were derived from computer graphics before the volume rendering specific techniques were created. Techniques based on extracting 2D contour curves [53] and 3D contour maps [88] or detecting 3D surfaces [46] from implicit functions can only visualize the boundaries of interfacing materials at once [17]. They assume the rest of the data does not contribute to the image produced. Furthermore, the geometric intermediate representation is inherently limited to a binary classification decision: whether the surface passes or not through the voxel. Moreover, it exhibited considerable artifacts (false positives and negatives) when small or noisy features were present in the data [43].

(b) Direct Volume Rendering (DVR).

On the other hand, the premise in direct methods is to render two-dimensional images from the three-dimensional volume directly. Techniques based on ray casting [22, 43] or material percentages [17] proved to be good starting points that avoided intermediate representations but instead attempted to produce the images from the data either by shading each sample along the ray or by using gradients to estimate the amount of surface present. Ray casting is furtherly described in-depth in the following section as it accounts for the foundation of this work.

2.2 Basic Ray casting

In ray casting, imaginary rays are shot from the viewing point through the image plane, usually one ray per image pixel. Each ray is then traced across the volume at a chosen number of evenly spaced points. A fully opaque background is set as
a stopping point behind the volume. The collected values are then composited in back-to-front order to produce the color for the pixel location where the ray passed through. More importantly, shading is independent from the classification (RGB and alpha) which yields a more robust strategy to represent volumes internals than the limited binary decision approach just explained.

The original ray casting algorithm [43] suffered from its very same main benefit: given that there is no intermediate geometric representation, the shading parameter selection proved critical to the effectiveness of the visualization. As a matter of fact, the technique is very sensitive to artifacts introduced during the acquisition process. Alongside, it is characterized by a computationally intensive nature that used to be far from executing interactively.

2.2.1 Empty space skipping and early ray termination

After the original ray casting algorithm many modifications and enhancements were produced. Levoy himself elaborated on his work by adding two important optimizations that helped alleviate the cost of the algorithm computational intensity. First, he proposed an empty-space skipping strategy based on a hierarchical spatial enumeration (an octree) that allows the rays to further advance more quickly in case of encountering coherent regions of empty voxels. Different levels of volume granularity can be coded into the octree, in a way that during traversal, the coarser levels are the starting point. Then, only in case of the coarse branches having non-empty volume children, finer levels need to be traversed. Otherwise, empty coarse volume pieces can be skipped completely, making the traversal faster. Figure 2.4 illustrates how the empty space skipping technique works.

The second optimization is early ray termination. Basically, it attempts to stop the ray traversal when an opacity threshold, usually close to opaque, is reached. This threshold can be chosen by the user. Worth noting is that in some cases, the
Figure 2.4: According to empty space skipping, only the gray boxes in the picture contain the volume. All other pieces of data are skipped. Image is courtesy of [20].

most meaningful opacity changes take place closer to the ray entry point. In fact, the changes occurring after the maximum possible opacity is reached cannot be accounted by this additive model as they are assumed to be occluded in any case [44].

2.2.2 Texture based ray casting

The processing power required to achieve the necessary frame rate is not possible using a single CPU. Therefore, the use of graphics hardware to accelerate volume rendering is a key point for interactive visualization. The main principle is resampling the data from 2D or 3D textures to a specified surface or proxy geometry. These polygons can be aligned according to the axis or the image plane. Inside the graphics processing unit (GPU) there is a massive number of smaller dedicated processing cores that can execute trilinear interpolation in parallel. This effectively reduces the time of execution [14, 6]. Figure 2.5 depicts proxy geometries aligned utilized for rendering with texture memory.

Later, there was a considerable amount of effort to develop new optimization strategies using the graphics hardware and to adapt previous ones from software based
While these advanced hardware operations were exclusive to high end graphics stations, it did not take too much time to make it available to off-the-shelf components. Interactive volume rendering on consumer graphics hardware was possible by means of 2D multi-texturing capabilities found in such devices [70]. The approach was based on object aligned 2D slices and bilinear interpolation to substitute the need for trilinear interpolation. Intermediate slices are computed on the fly in real time regardless of the less powerful hardware.

2.2.3 Ray casting in programmable pipeline

Direct volume rendering based on 3D textures, or 2D multi-texturing otherwise, have become one of the main tools for interactive display and visual analysis of volumetric
Figure 2.6: In GPU ray casting, the front and back faces of a cube are used to encode the 3D coordinates and calculate the ray direction. Courtesy of [40]

scalar fields. Research in acceleration techniques for CPU based algorithms and the design of dedicated graphics hardware have brought well-known techniques on board, like early ray termination and empty space skipping, to be applicable to texture based volume rendering [40, 58]. These techniques were first possible with the availability of programmable graphics pipelines. It was possible then to use early z-test to discard empty fragments and further advance the ray traversal. Moreover, stream models for ray tracing [67] were designed to feed the data from the rasterizer to the fragment processing units as a general strategy to leverage the intrinsic data parallelism of the graphics hardware.

Rendering to 2D texture maps instead of the framebuffer made multi-pass direct volume and isosurface rendering algorithms possible [40]. This approach allowed the introduction of many optimization techniques using the graphics hardware to accelerate the volume rendering task. The algorithm consists of two passes to render 2D texture maps to calculate rays entry and termination points. Then, a fixed number of passes are used for ray traversal plus some intermediate passes for early ray termination checking. The latter could potentially reduce the number of fragment operations to be performed. Figure 2.6 illustrates the coordinates encoded into colors for cal-
culating the direction vectors of the rays. Single pass ray casting was later possible with the introduction of dynamic flow control to the fragment shaders [28, 58]. Consequently, using the fragment shader one could trace a ray through the volume within single execution and avoid the overhead of executing the shader multiple times.

2.2.4 Advanced Applications and Techniques

Graphics hardware accelerated volume rendering have made real time fly-through applications like virtual endoscopy possible, but it also poses an important step for new functionality to be explored. One big limitation is the fixed amount of available memory when data sets have further grown larger. Different approaches like bricking and deferred shading [75, 28] were introduced to circumvent this limitation. Bricking is described in section 3.2.1. More advanced techniques can be found in the work by Scharsach et al. [75] where OpenGL normal geometry rendering is used along with volume rendering to display mixes of arbitrary 3D meshes and volumes in the same scene, making the applications more flexible and interesting. The screenshot of a fly-through application is shown in Figure 2.7.
Figure 2.7: A screenshot from a fly-through visualization. Image is courtesy of [75]
Chapter III

State of the art

Building up on Chapter II, this one presents a rather specific overview of the more recent related work. It is devoted to both parallel rendering of extremely large data and the use of tiled displays for visualizing the rendered images. It concludes with a summary of the various interaction mechanisms previously researched to leverage the visualization in these ultra high resolution display systems.

3.1 Parallel Rendering

The introduction of the massive parallelism within the graphics hardware architecture has opened multiple windows to exploring ways to exploit the parallel resources in multiple ways to achieve better performance. The design of the architecture was driven by the significant amount of arithmetic processing and computational intensity generally found in interactive 3D graphics. Different designs were devised to accommodate massive multiprocessing as a consequence of the incapability of the processing technologies at the time to provide a decent performance. In his design, Torborg \cite{83} described an arbitrary number of identical processors operating in parallel. These processors could be programmed identically as if they were a single processor system, something similar to the SIMD processing architectures found today in modern GPU
hardware. Other designs were focused on developing parallel interfaces [36] to run on parallel architectures [32] or libraries [13, 81] to leverage parallelism in a different way.

Other approaches consist of using off-the-shelf components like clusters of PCs [29, 73]. Although these systems do not have the power found in high-end ones, high speed networking has made possible to join all the distributed resources to achieve equal processing power at lower costs.

Along the years, different parallel models and architectures have been explored in the quest for optimal performance when handling the ever increasing data set sizes and the more demanding processing tasks. Different parallel approaches have been explored in the parallel rendering framework context. These approaches can be classified according to the architectures they were designed to run on. Moreover, a different classification can be performed depending on the sorting step if the parallel rendering task is seen as a sorting problem [55].

### 3.1.1 A memory architecture classification

Parallel rendering systems can be classified into three main groups, according to their memory architecture: shared, distributed and hybrid. For distributed memory systems, object space partitioning is frequently used [32, 48] while image space partitioning schemes are more often found in shared memory architectures.

1. Distributed memory

In a distributed memory architecture, the total amount of available memory accounts for the aggregation of physically separated but interconnected systems. On a higher level abstraction, it is seen as a logically contiguous big space, whereas at a lower level, the necessary mechanisms to consistently deliver the seamless availability of the data are provided. Data decomposition
maps naturally in this distribution scheme because it can reduce the amount of synchronization needed. Inter-node communication is generally expensive as network transfers tend to be more costly than actual intra-node memory transfers. These systems are highly scalable and flexibly as more nodes can be added to increase the total amount of total memory available.

2. Shared Memory

In shared memory systems [56] the memory space is physically contiguous and accessible from all processors. Image decomposition is more often used for this architecture. Inter-node communication is performed by actual memory transfers which provides faster communication in comparison to distributed memory systems. The main limitation is scalability as it becomes prohibitively expensive to build systems with bigger amounts of shared memory and they cannot easily be extended.

3. Hybrid systems

Hybrid approaches consist of mixing features of the previous categories. In one example, data are distributed across the nodes using an image space partitioning scheme [54]. It is also possible to partition both data and image space to a keep coherent distribution of the work [73]. In distributed shared memory systems both shared and distributed memory schemes are used by having clusters of multicore processors. Challenges for these systems include ensuring to maintain cache coherence throughout the execution of the program because failing to do so progressively deteriorates the overall system performance [8].

Classification of the parallel rendering systems according to their memory architecture is not extensively explained here. The sorting classification described thoroughly in section 3.1.2 is the one used in this document from here on.
3.1.2 The sorting classification

A comprehensive way to classify the parallel geometric rendering algorithms in the literature can be found in [55]. The classification is done depending on where the sorting task takes place in a fully parallel rendering system. According to Molnar, rendering is fundamentally calculating the contribution of each primitive to each pixel which can be seen as sorting the primitives to the screen. There are two different parallel processing stages: geometry processing and rasterization. Parallelization usually takes place by splitting the primitives across the geometry processing units and by dividing portions of the pixel calculations over the rasterizing units. As a result, there are 3 categories: sort-first, sort-middle and sort-last. Nonetheless, in the context of volume rendering only sort-first and sort-last are applicable [51]. Another important consideration is that if rendering is based on consumer graphics hardware, a sort-middle strategy results impossible given that these systems are generally constrained to standard APIs, such as OpenGL. These APIs give no high performance access to intermediate rendering results. Therefore, sort-middle becomes impractical [74].
3.1.3 Sort-first systems

Sort-first algorithms follow an image order distribution approach. The final image is split into disjoint tiles and distributed across the rendering processors. Every processor effectively applies the complete pipeline to their piece of work. Redistribution can happen at the beginning of every frame computation. At first, the volume data is distributed arbitrarily across the processors. Then every processor determines the ownership of the subvolume, it keeps the data it is responsible for and sends all other data to the corresponding processor in charge. Enhancement to this part of the algorithm include parallel I/O so that every processor loads the data is responsible for rendering directly from disk without the need for transfer between processors [89].

1. Per-frame coherence

In geometry rendering, sort first algorithms can take advantage of per-frame coherence to reduce communication costs. When the scene being rendered does not contain sudden changes, retained-mode [55] can be activated to keep the geometry from the last rendered frame and use it in subsequent ones reducing the amount of transferred data. However, in real-time volume rendering, scenes (or volumes) are expected to be user interactive, which allows the user to change the whole scene from one frame to the next. These sudden changes can break any per-frame coherence that could serve as an advantage [36].

2. Back compatibility

Given that tiles of the final image are produced by every processor, a simpler way to look at it is that many rendering processes of smaller images are concurrently being executed in the system. Sort-first algorithms execute the full graphics pipeline in each processor making it instantly compatible with all different kinds of algorithms previously designed for serial rendering. This, of course, sets up
a great advantage in the applicability of previous techniques in a more straightforward manner lowering the effort needed to port them.

3. Scalability

Scalability of sort-first algorithms depend directly on the size of the final image and indirectly on the size of the underlying dataset. With increasing image sizes, sort-first systems keep the data sent to the frame buffer reasonable given that only complete pixels are transmitted. In this way, sort-first can scale well to handle increasing image resolutions. On the other hand, the indirect dependence on the size of the dataset can become a big drawback if redistribution of the data is not managed efficiently. In a shared memory architecture (section 3.1.1), which can be considered as an optimal setting for sort-first systems [47], there are clear limitations to the size of the datasets and the number of processors attained. Therefore, scalability is immediately restricted by the size of the built-in memory. However, given the simpler memory model, straightforward implementations are possible. In contrast, for the distributed memory systems (section 3.1.1) memory is physically separate; then, as the number of processors increases, the image space can grow into too many pieces (even if there is only one per processor) so that data redistribution and boundary data redundancy become serious limitations in scalability [73].

3.1.4 Sort-last systems

Sort-last systems follow an object-order distribution approach. The volume is split into sub-volumes and these are distributed across the rendering processors. Each of them does the different required transformations and renders its subvolume into a partial image regardless of its visibility. Later, all partial images are composited. This can happen in several different ways as detailed below. In the general sense, the
partial images need to be blended in a specific order for the final image to be correct. Volume pieces should be occluded by the ones in front accordingly [55].

1. View angle Independence

As with sort-first systems, parallel I/O can be advantageous so that each processor can only load its piece of volume concurrently with the others. Furthermore, different from image space, where work distribution is view dependent, in object space it can stay fixed and redistribution is not necessary. This obviously reduces network communication for distributed memory systems, making sort-last techniques a natural choice for these platforms [32, 47].

2. Parallel Compositing

At the end of the rendering pipeline, the resulting partial tiles still need to be composited in order to produce the final image. In serial compositing, n processors will produce n full size partial images. These n images then sent to a master node that will blend the images using depth or visibility tests to achieve the correct results. As the image size grows higher, transmission of partial images can become a serious bottleneck. Bandwidth requirements can grow prohibitively, only possible for expensive high-end network solutions [48]. On top of this, a considerable amount of data sent are still unfinished pixels. Furthermore, sort-last systems do not scale well on higher image resolution given these reasons [73]. A way to overcome this situation, compositing methods that work in parallel have been the subject of considerable research. In this way, potentially all cluster nodes can contribute with the compositing of the final image and reduce the time required. These parallel compositing methods are:

(a) Direct send [32, 49] divides the final image into n screen space tiles to avoid exchanging full size images. Each processor is then associated with the compositing of a tile, and sends over all other tiles from the full size
image it rendered. Tile images are then exchanged in one all-to-all communication stage. Once all tiles have arrived, each processors composites its tile independently. Next, composited tiles are sent to the master process for final gathering. Direct send only has one communication step with two synchronization points: when images are exchanged and gathered. Therefore, the algorithm works alright even if the synchronization mechanism is expensive. Moreover, as the processor count increases, the tiles become smaller, so do the messages transmitted over the network. This potentially reduces the amount of communication required. In the worst case scenario, direct send can send up to \( n(n - 1) \) messages. However, the algorithm can be tuned so that only the pixels that need to be composited are sent over, further reducing the number of exchanging nodes to an average of \( n^{1/3} \).

(b) Binary swap \[47\] and tree-like algorithms use a hierarchical communication scheme to reduce the amount of communication by avoiding all-to-all patterns. In fact, communication happens only between two processors at any time. The algorithm utilizes all processors by splitting the image between the two corresponding processors. However, the binary swap algorithm restricts the amount of processing units to powers of two. The 2-3 swap algorithm is an extension to the original binary swap algorithm to circumvent this restriction \[90\]. It allows having an arbitrary number of compositing processors by making it possible to have groups of three exchanging processors instead of two. The binary swap compositing is shown in Figure 3.2.

(c) Hybrid methods like the Radix-k algorithm \[63\] combine both approaches (direct send and binary swap) depending on the problem setting. It can adaptively run binary swap or direct send depending on some criteria. Other hybrid solutions include parallel compositing libraries like IceT used
Figure 3.2: The binary swap compositing for 4 processors. At every level, the tile is split between two processors. This method ensures all processors are used throughout the algorithm execution [47]

in [21] that implements many different compositing modes that can be extended as well as mixed.

3. Scalability

Scalability of sort-last algorithms is heavily bound to the network bandwidth. Given that sort-last systems send pixel data, higher resolutions increase the network bandwidth requirements. Therefore, they do not scale well with increasing screen resolution. However, given every time faster network interfaces available, sort-last systems have been brought from the high-end spectrum to
more approachable alternative of clusters of PCs interconnected with high-speed networks. Contrary to sort-first, data can be left stationary reducing network transfers, making sort-last systems more scalable with the volume data size. In fact, with the very large size of the data, high-speed networks have become compulsory for any kind of parallel system and thus widely available in modern systems. Generally, sort-last approaches are built on top of memory distributed systems. Historically, the scalability of these systems has been proven superior over shared memory systems. They are generally comprised of clusters of inexpensive PCs. Advantages include ease of replacement and technology upgrade, making them very flexible and highly modular.

3.1.5 Hybrid algorithms

There is a third possibility apart from sort first and sort last: a hybrid between the two. In one type of these algorithms, the rendering is performed in object space but the compositing in image space. Some of the references included previously can also be put under the category of hybrid \cite{32, 47, 79}. Both object and screen space can be partitioned into tiles and groups on a dynamic basis (after every frame) to balance the load among the PC nodes \cite{73}.

3.1.6 Parallel rendering on clusters

As graphics hardware became available to consumer markets, considerable effort was put into developing strategies to perform interactive volume rendering on them. High speed networks were the key to leverage clusters of PCs or GPUs as an alternative to supercomputers.

CPU clusters are basically arrays of interconnected PCs. Samanta et al. \cite{73} presented a polygon rendering system that can use off-the-shelf components in a cluster of PCs. Partitioning takes place in both object and screen space into groups and
tiles correspondingly. Moreover, regions on screen covered by groups of polygons are
closely correlated with the pixel tiles assigned. These tiles and groups are distributed
to the different nodes and used to balance the load and minimize the overheads. In the
same manner, the system presented by Magallon et al. [48] delivers interactive frame
rates in a sort-last volume rendering system executing on a cluster of PCs connected
by a high speed network. One of the implementations of binary swap found in [47]
is ported to a cluster of networked workstations. In this version, communication and
synchronization of nodes is managed by means of a daemon. Later, Stompel et al. [79]
introduced a system that runs on a PC cluster with 100BaseT network. The system
is based on the sort-last approach and the compositing is performed by computing
a communication schedule of overlapping pixels. The traverse is done in scanline
order to provide a better load balancing. In the same way, Peterka et al. [64, 65]
present a study on the scalability and costs analysis of parallel volume rendering on
the IBM Blue Gene/P. This is a distributed-memory parallel CPU architecture, as an
alternative for GPU volume rendering for large volumes, large image sizes, and very
high quality results in the peta and exa-scales. More recently, Marchesin and Ma [50]
use a CPU cluster stating that although GPU clusters are a better match for parallel
volume rendering, these are not always available. Moreover, CPU clusters are often
the only way to achieve in-situ visualization useful for extremely large simulations.

GPU clusters are also arrays of interconnected PCs but every node has access to a
hardware accelerator. Muller et al. [58] introduced a volume rendering system based
on a sort-last approach that executes on GPU clusters using infiniband. There is also
a client and server separation to make remote rendering possible. The thin client
communicates via TCP/IP and works as an image viewer that handles user input
events. Among the server nodes, communicate and synchronization is performed using
message passing (MPI). Compression of the final image is possible before sending it to
the client. In fact, partial images can be sent to the client in two ways: gathered and
composited or directly without compositing. In the same way, Stuart et al. [81] use a cluster of GPUs to perform parallel volume rendering using MapReduce. The latter is an API that handles the communication and the computation in parallel. There is an asynchronous streaming interface that allows network communication, CPU/GPU data transfers, disk access and GPU kernel execution to all happen concurrently. Very recently, Fogal et al. [21] presented a system that is meant to run on a cluster GPUs. In fact, multiple GPUs per node are possible. Each GPU should match to a CPU core as part of a parallel streaming model.

3.1.7 Load imbalance

Load imbalance is a condition where the cluster nodes end up performing different amounts of work. The load differences are very important because the system speed is bounded by the slowest of its units. In any distributed environment, it is ideal to have the work be distributed as equal as possible to the different resources. Load imbalance and balancing strategies have been researched for a long time in the parallel computing community. Naturally, parallel rendering systems are not exempt from load imbalance conditions.

Sort-first algorithms are more susceptible to load imbalance problems due to an unequal distribution of the scene on screen. This can lead to processors with empty tiles or with a very small amount of data to render. One way to overcome this situation is to have the final image be decomposed into smaller tiles or differently shaped partitions, and make each processor render many of them reducing the possibility that a processor could be left with empty tiles only [55]. However, sort-first systems do not scale well with the number of processors because the image space has to be split further as the processor count increases yielding smaller regions and making data redistribution heavier and more frequent [73].

One important design decision in load balancing for sort-first algorithms is the
shape and size of the image pieces. The key according to [57] is to make the length of the boundaries minimum in order to reduce the redistribution of the data. Another concern rises when voxels go off the screen since data redistribution is not immediately necessary for the next frame. It can potentially accumulate and overload one single node later. A way around is to share the amount of off-screen data per processor and adjust the number of on-screen accordingly.

Load imbalance in sort-last systems can also be due to uneven distribution of the rendering work. In contrast to sort-first, the scene distribution on screen is not directly a source of imbalance for sort-last systems. Equal blocks of data are distributed among the rendering nodes. This ensures an initial even distribution. It is possible, however, that a given view frustum discards the work produced by some of the nodes. Consequently, there are nodes rendering but not contributing to the final image. This is due to the late visibility test in sort-last systems. Generally, sort-last systems are less prone to load imbalance in comparison to sort-first.

### 3.1.8 Load balancing techniques

Load balancing techniques can be classified into static and dynamic.

1. **Static load balancing**

   In static load balance, the work is distributed among the nodes once during the whole pipeline. This decomposition is invariant for the life of the program execution. There are two basic partitioning options: contiguous and interleaved. Contiguous partitioning splits the image plane into one block and assigns one piece to each processor. Interleaved partitioning, in contrast, splits the image plane to many smaller pieces that are distributed in a round-robin fashion. The former is characterized by a poor load distribution depending on the view dependent complexity of the image. The latter results in a more even distribution
as the tile size decreases; however, processing too many images increases the corresponding overhead. The main advantages over dynamic balancing techniques are the implementation simplicity and the lightness of execution [59].

2. Dynamic load balancing

On the other hand, dynamic approaches offer a more powerful way to handle load balancing. Different information like the previous frame rendering time [58, 51, 21], pixel rendering estimated costs [50] or occlusion information [50] can be used to calculate and possibly redistribute the load for the next frame. These statistics are generally managed by hierarchical data structures like kd-trees [51, 58, 56, 47, 21].

The general principle consists of using the kd-tree to encode the decomposition of the data space. At each level the data is split alternating along orthogonal axes. Leaf nodes contain the effective partitions of work, while inner nodes encode whichever load statistic is chosen. Kd-trees are traversed in specific orders to balance the partitions with different strategies but can also be used for empty-space skipping [51]. The kd-tree can also be used to distribute uniform texture bricks in object space partitioning [58] and to determine the compositing order of the partial rays. The drawback to this is that replication of boundary data is necessary [47].

A different technique consists of introducing a central queue of work where processors can go and grab pending loads as soon as they become available [59]. Processors can use a shared queue of work that contains a front-to-back sorted list of bricks [50]. The different processors take work away from that queue and render the corresponding bricks to a shared frame buffer. Race conditions can happen if rendering to the shared buffer directly. Nevertheless, this is solved by having each processor render each brick to a private buffer.
and then moving to a complete work queue. A given volume rendered brick is only blended from the private buffer onto the shared buffer once all the bricks it depends on are in the completed work queue. However, choosing a central dispatching mechanism serializes the program back again. A single point of control can quickly grow overwhelmed and become the bottleneck of the system as the number of processors increases.

In general, dynamic load balancing introduces more complexity and overhead to the systems either by managing the queues, sharing load statistics or calculating and performing the redistribution of the data for the following frame calculations if necessary.

## 3.2 Volume Rendering of large data

State-of-the-art simulations of physical systems can generate extremely large datasets. Previous attempts to solve the large data visualization problem is using compression to fit the entire dataset into the texture memory [86, 26, 84]. However, the visualization of current data sizes becomes impossible to perform in single systems. Therefore, techniques that address the visualization of such massive systems have been developed on top of the parallel rendering paradigms previously described.

Throughout the years, systems have been designed to address increasing data sizes. Childs et al. [10] built a system that can interactively render up to $3000^3$ unstructured elements using hundreds of cores interactive rates. Super computers like the IBM Blue Gene/P parallel supercomputing architecture are studied in [64, 65] to be used as exploration tools for volume rendering of data that are still not possible to visualize on available clusters. Data sizes range up to $4480^3$ using tens of thousands of CPU cores. Later, Fogal et al. [21] presented a system that can render up to $8192^3$ using a GPU cluster based on sort-last algorithms. And more recently, Howison et
al. [31] can use above two hundred thousand cores by exploiting different forms of parallelism. Worth to note is that some these authors have reported deliberate use of less cores for some tasks like compositing yields better results. Apparently, there is a limit for the simple scalability of only adding more processing cores.

We now briefly summarize recurrent techniques developed to handle large datasets.

### 3.2.1 Bricking

Bricking is technique that consists of dividing the volume into equal size blocks. The blocks are much smaller than the volume itself and provide a higher level of granularity for the rendering. Furthermore, bricks that do not contribute to the final image can be easily culled to reduce the load and provide a basis for empty space skipping [56, 58]. Figure 3.3 depicts how the bricks would look like. Unfortunately, there is no correct brick size. If the bricks are too small, the overhead of processing many of them can potentially deteriorate the performance. This is possible in GPU rendering given that multiple texture memory loads and swaps are required to render the complete volume [5]. On the other hand, bricks that are too large increase the chance that they contain some part of the volume, making bricking pointless and unhelpful.

Bricks can also be used as the unit of work to distribute among the parallel processors. However, if neighboring bricks are processed by different processors, their borders need to be replicated every time to keep interpolation consistent. Otherwise, image artifacts can occur.

A different way to handle bricks is to associate them with an histogram of their data [51]. The histogram can be then convoluted with the current transfer function to determine the visibility of the corresponding brick. Similarly, bricks outside the view frustum can be discarded. Bricks can also be used as cache memory management units as well as part of multiresolution methods where high resolution bricks are used
Figure 3.3: A volume decomposed into bricks extracted from [20] for areas of interest combined with low resolution ones for everything else [50].

### 3.2.2 Multiresolution techniques

Multiresolution techniques consist of adaptively rendering depending on the screen resolution. Provided that the data resolution is higher than the one from the display, there is no real benefit from rendering the larger resolution data over the one of the display. In fact, real performance benefits can be gained from rendering display-aware resolutions, potentially lower, whenever possible. This effectively reduces the amount of samples in the data potentially decreasing the amount of computation needed for rendering. It also decreases the texture memory requirements making it possible to render larger datasets [42, 85] and achieve interactive frame rates [80]. Moreover, it helps preventing aliasing and loss of detail. Figure 3.4 illustrates how four levels of detail can be applied to one-dimensional data sets. Deeper level values are used for the areas closer to the regions of interest.

Different resolutions can be mixed to save both memory and computation cycles without necessarily compromising the final image quality. For instance, far away
data can be replaced with textured maps containing depth information as done by Aliaga et al. [1]. Furthermore, adaptive schemes allow the system to render regions of interest at high resolution and the rest at progressively lower ones leaving more room for possibly larger datasets [12]. Li et al. [45] developed algorithms that perform automatic level-of-detail selection to match different parameters like the desired frame rate. The algorithm is also known as time-critical level-of-detail selection. It attempts to predict the rendering time by collecting performance statistics and dynamically selecting the level-of-detail for the next frame. It is possible for the user to change some parameters so regions of interest are rendered with higher quality.

When using mixed resolutions, potential artifacts, like discontinuities, can occur. A way to avoid them is by using different proxy geometries other than planes [85, 42]. There is a considerable computational price to pay for this, but it ensures the results are artifact free. A different method is to limit the artifacts by modifying the transfer functions in the lower-resolution nodes [12].
3.2.3 Octrees

In the general case, multiresolution techniques are applied by encoding the data at different resolutions into a spatial hierarchy structure, e.g., an octree [77], that can be selectively queried for adaptive rendering. The structure is generally constructed in a single preprocessing step. Each resolution level is associated with a level in the octree and is half the resolution of the next level. The leaves define the original data while the internal nodes encode progressively lower-resolutions by means of subsampling the node’s eight children. The root node is the coarsest resolution. Figure 3.5 illustrates the octree hierarchical structure.

At every frame, nodes from the structure are chosen to match some parameter like display resolution. After tree traversal, chosen nodes are sorted and rendered in back-to-front order. Rendering is done by selecting high resolution nodes closer to a center of attention, or areas of interest, and low resolution ones for far away regions. Rendering from octrees can be tunable. The quality of the image might be defined as a combination of importance or other user specified bias parameters [5].
For data sets larger than the texture memory available, it is not possible to have the entire octree in memory. However, an out-of-core octree can save the data contents of each node on disk. The tree is constructed by breaking the data into texture memory fitting chunks and incrementally building the tree on disk by updating only the spatial regions touched. Nodes are loaded on demand depending on the field of view. If the files of the tree are written one for each node, a hierarchy structure file can be created. Such file can contain the connectivity information as well as some other information needed for visibility culling like the node's bounding box. This file must fit in memory, but it is in principle trivially small [11]. In addition, bounding boxes can also be computed on the fly in case there is a need for reducing texture memory access even further [23].

3.2.4 Multilevel cache and out-of-core techniques

Out-of-core techniques consist of using data structures that no longer fit into the texture memory of the graphics hardware. Ideally, all necessary data should fit into main memory as it provides the fastest access and reduces execution times. For the graphics hardware, the main memory is the texture memory. Nevertheless, for the cases when data is larger than the texture memory available, out-of-core techniques can physically split the data into pieces that are smaller enough to fit while providing a virtual unified view to the rest of the system.

Internally, the underlying system provides mechanisms to dynamically load the required pieces of data and unload the no longer needed ones when no more room is available. Asynchronous data loading can potentially hide the time it takes to get the missing pieces [8, 23]. It is also possible to use speculative prefetching combined with visibility algorithms to guess which blocks the user may see next and prefetch the request to memory [11].

A similar behavior is found in memory paging mechanisms for CPU cache systems.
The goal is to keep the necessary data on memory all the time it is needed. As well as bricking, the page size is a parameter to tune. It is possible to have multilevel cache hierarchies [56, 51]. Intuitively, every level and its work unit are generally much larger than those of its predecessor.

Cache levels are managed according to different policies like least recently used (LRU) replacement strategy [12, 77]. Thus, whenever space needs to be freed to accommodate new data pages, the system will check for usage statistics and remove enough pages prioritizing the ones that were used the last. The main goal is to keep the most frequently used data available close to the processor, given that the difference between accessing local or remote mass storage is critical. Cache coherence becomes another point to take into account for the best performance of the system.

Hierarchies with more levels have been investigated before. In the work by Castanie et al. [8] a four-level cache hierarchy is implemented. It spans across the graphics memory, local main memory, cross-node memory and disk. A global software-based distributed shared memory scheme is implemented over a high speed network that can check page residency on other cluster nodes before accessing the local disks.

### 3.3 Display Arrays

Display arrays have become increasingly important for user interactive and collaborative visualization applications. The ultra high resolution enables very detailed visualization of the large datasets described in section 3.2. The large physical form factor can show rendered objects at their natural sizes. This is a property that might prove critical to human perception and evaluation. It also favors collaborative work by people viewing and discussing the visual data. In addition, it leverages user immersion by providing wider fields of view and possibly richer visual interfaces.

The possibilities are many, but so are the challenges to realize them. One of the
basic requirements is a high performance rendering system that can power interactivity. Also, a flexible and powerful framework to support a wide range of old and new applications. Interactive rates for extremely large three-dimensional models are desired in fields like computer-aided design, molecular biology, scientific and medical visualization.

Display arrays are generally composed of an array of liquid-crystal displays (LCD) or projectors driven by clusters of graphics accelerated PCs. This has opened a window to a myriad of applications in attempts to get the best performance and most efficient ways to utilize the resources. Advantages of a PC cluster include inexpensive prices and a flexible architecture that can be extended without compromising the whole system. In contrast, design is generally constrained to standard APIs making low level tuning usually impossible.

3.3.1 Visualization on display arrays

The display array major challenges include compatibility to a wide range of applications, the capability to scale with the least performance penalties and intuitive interaction mechanisms that can use the resources more effectively. Therefore, a large amount of research has been dedicated to develop efficient, scalable and flexible strategies to use the ultra high resolution of the display arrays. Traditional desktop displays constrain the spatial analysis of data. They trade-off resolution for spatial extent, as it is only possible to have finer resolution of smaller volume pieces on screen, or a global volume overview with little level of detail.

The various solutions can be classified into generic and special purpose \[18\]. Generic systems provide toolkits and frameworks to build applications on top of them. They usually impose little or no change to the existing applications and internally handle the distribution of the work and the management of the resources. Limitations include the degree of optimization tuning and flexibility. On the other hand, custom
solutions are specific to every application, potentially providing the best performance and the greatest scalability at the price of incompatibility with other systems and the development costs [76, 13].

### 3.3.2 Generic display array frameworks

Generic frameworks like WireGL, Chromium, Equalizer, SAGE and CGLX attempt to solve the display visualization challenges by building up efficient middleware that minimizes the application changes required to use display arrays. At the same time, they try to provide efficient and transparent mechanisms to utilize the available resources. These systems are briefly discussed next.

WireGL [33] is a sort-first system executing immediate mode rendering. It stands as an OpenGL API driver that intercepts the local OpenGL calls from the application. It then streams the graphics commands selectively using a compact network protocol while keeping track of the machine state for correct execution order with little performance penalties. The main advantage is that applications can be efficiently rendered to the tiled display without modifications. The major drawback is that the system is limited to a single application. Moreover, since its design is only focused on efficiently scaling to the higher resolution, no performance gains are achieved by taking the application from a single workstation to a cluster of workstations.

Chromium [35] is based on WireGL and provides mechanisms to manipulate streams of graphics API commands on PC clusters. It consists of stream processing units that can be reconfigured to support sort-last and sort-first architectures depending on the application. Moreover, the framework can be extended to integrate existing user interfaces and many different algorithms with little or no change. However, in highly dynamic applications, Chromium can potentially exchange prohibitively large quantities of data and affect the application performance. Since it works by intercepting OpenGL command streams, like WireGL, it is limited to ap-
lications using the same graphics API.

The Equalizer framework [18] provides an OpenGL based API to develop scalable graphics applications for distributed visualization clusters. It takes care of the distributed execution, synchronization and final image compositing for both sort-last and sort-first based applications. However, effectively using these features requires in-depth knowledge of the framework, significant code changes and application specific adaptations.

More recently, the Scalable Adaptive Graphics Environment (SAGE) [69] is a flexible framework that applications can use to render to a display array with minor changes. It separates rendering from display and it makes no assumption other than the pixel stream it expects as an input. The user interface metaphor builds on the idea of an extended screen extension giving the idea of a continuous computer screen. SAGE allows for multiple clients and all changes are broadcasted via message passing. Multiple instances of the same application are possible and application windows can be freely moved or resized on the tiled displays. Remote visualization is possible over Wide Area Network (WAN). OpenGL applications can be easily adapted by adding a pixel read back call at the end of the rendering. On the downside, the bandwidth requirements can be extremely high.

Finally, the Cross platform cluster Graphics Library (CGLX) [16] is a flexible and transparent OpenGL based graphics framework for distributed, high-performance visualization systems. It allows OpenGL based applications to display on multi-tile environments. It provides a unified OpenGL rendering context by intercepting and manipulating certain OpenGL directives. It can manage the visualization grid without additional implementation requirements to the user. In contrast to SAGE, CGLX does not impose the pixel read back operation on the system that can potentially become prohibitively expensive at higher resolutions.
3.3.3 Custom display array solutions

Custom solutions provide the advantage of greater support and scalability at the price of inherent inflexibility. They are specifically built for each application and are based on the parallel rendering algorithms discussed previously in Section 3.1. Solutions based on sort-first systems working with tiled displays, have commonly been investigated in the past. They perform a similar domain decomposition in screen space to the one naturally provided by each PC driving a single screen tile of the display array. Some examples of these systems are described next.

Correa et al. [11] developed a sort-first parallel approach for out-of-core rendering of large models to tiled displays. Approximate and conservative modes available on the system allow to trade off responsiveness over accurate rendering. A client machine can be attached to the server through socket communication for input processing and view parameters sending at every frame. Schwarz et al. [77] present another sort-first system for tiled displays. It uses a multiresolution octree that is physically distributed across the cluster memory, as well as a multi-level cache hierarchy that spans both the GPU and CPU memories. Given that sort-first algorithms produce final image tiles, these can be sent for display in a straight forward manner.

Display wall domain decomposition can also be performed by using virtual tiles. The main goal is to avoid overlapping replicated data across different nodes and little interprocess communication. Tiles can be allowed to be of any shape and size as long as data map to only one. This sometimes means that pieces of big virtual tiles might lie in different physical memory from the processor in charge of display. These tiles will need to be transferred accordingly. Solutions having one graphics processor for each screen and sorting data among the processes according to their overlaps with projected regions have poor performance if graphics are not uniformly distributed as explained in section 3.1.7. Instead, this system minimizes the communication by
replicating the data on every PC and only transferring remotely rendered pixels [74].

3.3.4 Interaction and input mechanisms

Display arrays provide the high resolution required by many applications in different fields. Overall, high resolution tiled displays are preferred over standard desktop for various reasons like their capabilities to approach larger audiences, for naturally engaging into collaborative work, to further immerse the user in virtual reality applications and to facilitate visualization of large and complex data sets by maintaining both overview and detail view simultaneously.

Nonetheless, these systems also present important challenges that have been the aim of much research effort in the past. One big challenge is system interaction mechanisms that can intuitively and efficiently use the system resources. Simpler natural interactions like speech, hand gestures and the more general direct interaction are preferred over traditional devices like mice and keyboard [2]. In addition, natural and well known methods are also preferred over having to learn new specific forms for every system.

3.3.5 The interaction challenge

Input methods become a major challenge as traditional interaction mechanisms are less applicable under the different environment settings. Normal window-based interaction metaphors break down [34]. One example is moving a window via the drag-and-drop mechanism. It requires the user to hold the object all the way down to the desired location. While trivially short in desktop settings, distances in display arrays can grow to thousands of pixels long, making it an impractical way to perform the same task.

Current stationary installation of traditional devices like keyboards and mice require the user to sit, disregarding the different capabilities and possibilities of the
environment settings. The larger spatial dimensions of display arrays invite the user to stand, walk and move. Consequently, classical input mechanisms are considered unintuitive and missing the naturalness required [62]. In fact, high resolution displays have been shown to improve users target finding and comparison skills as well as reducing frustration and increase response confidence. They also outperform smaller displays for navigation tasks [2].

Alongside, the wall interaction metaphor brings the advantages of the user interface with those of whiteboards. The computer based content can be managed with the ease of informal writing, sketching and space management [25]. One approach is to emulate office environments that use the space to display a collection of visual material that support group activities [33]. The important feature of any wall-sized display input mechanism is that it minimizes the amount of attention on the mechanics and focuses on the interaction itself.

We now discuss the various input methods found in the literature:

3.3.6 Speech

Speech recognition and synthesis for computer command consist of having the computer to recognize human voice by using a set of established commands. A bigger tendency supports interfaces combining speech and hand gestures, considered more natural and efficient than speech or gestures on their own [62].

These techniques are based on computer vision and speech recognition algorithms to construct a more natural human-computer interface. Virtual reality environments [61] are a good example where combinations of freehand gestures and spoken words have been investigated. The combination of multiple ways to interact makes the interfaces more robust and powerful.
3.3.7 Tracking and indirect Interaction

These methods include a rich variety of tracking technologies ranging from the more traditional mice, camera-tracked laser pointers and digital pads to the more sophisticated devices integrating gyroscopes and accelerometers found on modern smart phones and video game consoles. Strong effort is placed into investigating mechanisms that use cameras for tracking movement and positions with computer vision image processing algorithms.

Digital whiteboards like the one from Rekimoto [68] use direct pen-like interaction to allow the user to manipulate the different elements on screen. In his system, a handheld computer is given to each participant that works as a personal extension of the whiteboard. The same pen can be used on both devices. The goal is to augment each participant interaction by providing tools like color palettes, an on-screen keyboard and a web browser. The user can pick objects, images and text and drop them into the whiteboard for collaboration. However, although a tiled display can be seen as a bigger whiteboard, its physical form factor becomes impractical for interfaces that require touching directly on the screen.

An interesting way to circumvent this difficulty is by remote pointing using lasers [9]. Chen et al. [15], present a system that can track multiple pen-like devices with a set of video cameras. It is possible then, to free-hand sketch and interact in spite of the size of the screen. It also provides access for an increased number of users as the pointers subsequent positions can be registered according to their gradients. Using computer vision techniques, the system determines the different strokes. Each camera connects to a CPU that processes the incoming video and determines the locations of interaction at each time step. This approach can be scaled up with the screen resolution by increasing the number of cameras. Pen strokes can even be combined wirelessly between bezels spanning multiple displays [30].
Other camera-tracked devices include passive objects like wands [7]. The passive wand is tracked in 3D using computer vision techniques. Combination of wand move state tracking and visual widgets are used for more complex and richer interaction.

Priorly investigated alternatives also include tiled table-top devices [78]. High resolution visualization tables merge interaction with tiled display technology. LCD panels can be used. Advantages include day light visibility and compactness as no projection distance is required. There are also easier color calibration and wider fields of view. In addition, they tend to be cheaper to maintain then high resolution projectors, as changing projector bulbs can grow prohibitively expensive for high-resolution configurations. In the work by Krumbholz et al. [41] cameras were mounted overhead to track objects identified by a unique infrared pattern. A computer mouse was set up to interact with the system. Tables can also be used along vertical display walls for annotation or closer inspection.

More recently, increasingly sophisticated tracking solutions include gyroscopes and accelerometers found in many modern mobile phones. Grubert et al. [24] introduced an interaction technique applicable to display arrays that consisted on using the Wii remote to allow people to sit and interact from a distance with entire display overview. Although direct pointing is also included in the investigation, the results show that some degree of accuracy can be sacrificed in remote pointing in order to gain remote interaction capabilities and similar responsiveness speed.

### 3.3.8 Gestures and direct interaction

In contrast to device tracking, direct interaction aims at enabling the users to use their bodies for system manipulation. The main reason is that human beings are more accustomed to hand things around than they are to use virtual extensions of their body. Generally, these techniques also rely heavily on computer vision or other tracking mechanism but do not impose the utilization of any device to the user [61].
3.3.9 Hand Gestures

Pavlovic et al. [61, 62] explored visual hand gesture analysis enhanced with speech recognition as a bimodal gesture-speech interface. The main goal is to control 3D visualization on a display wall. The authors argue that users find using their hands a more natural and intuitive task. Their work is under the premise to attach fewer or no devices to the user.

In their system, hand gestures are detected through cameras and interpreted using computer vision techniques. It is necessary to be able to efficiently extract the background and distinguish meaningful gestures from unintentional ones. Voice commands are monitored through a microphone and recognized using automatic speech recognition techniques. Fewer restrictions are imposed at the cost of more intensive computing of complex visual interpretation mechanisms.

Alongside, a different alternative is to use touchpad tracking surfaces to explore multi-finger and whole-hand gestures [49]. Hand gestures can be extended to using both hands for more complex interactions.

3.3.10 Touch interfaces

Touch screens and touch interfaces as input devices have been an active area of research for the last couple of decades. Different technologies can be used like camera tracking or capacitive sensing, among others. However, drawbacks like imprecision, bad quality, high production costs and limited interaction were important impediments for them to reach consumer products.

There are three broad categories of multitouch input technologies: optical, capacitive and resistive. The brief descriptions of these categories presented here are extracted from [72].

1. Optical sensing
This technology uses digital video cameras to capture images that can later be processed to obtain touch and proximity information. Other possibilities include the use of infrared lights that get interrupted via hand touch. The scattered light is captured by digital video cameras inside the devices. It is possible for pressure to be sensed by incorporating soft light-scattering layer over the touchable interface. These devices, however, tend to be bulky and sensitive to lighting conditions.

2. Capacitive sensing

Devices using this technology can be compact and flat. Although it is not possible to measure pressure, the capacitive sensors are able to measure surface area of contact. An important limitation is their reliance on the dielectric properties of the human body. Although they are able to track a fingertip, it is not possible to track pen-like devices.

3. Resistive sensing

This technology can be implemented using multitouch sensors at inexpensive costs. It uses little power and can effectively measure pressure. It has been traditionally used to create sensor arrays, but its inability to track signals between the spacing of neighboring elements have limited its use for low-precision devices. Rosenberg and Perlin [72] presented a resistive sensing technology that can acquire high-quality anti-aliased pressure images at interactive rates on thin, flexible and scalable pads. The pressure variation can be detected within three orders of magnitude that help identify multiple fingertips from pen-like devices.

For tiled displays, direct interaction by touch screens turns out impractical and probably of no use. However, there are other strategies like interacting from touch-screen handheld devices. With the advent of multitouch interfaces, richer and more natural interaction can be performed by means of multi-finger gestures. Ponto et
al. [66] presented a system based on display arrays that support interaction through multitouch table, tablet and phone devices. New devices that can be added and removed from the system at any time. The system can recognize many sources of input enabling multi-user support. The system can send down-sampled scenes for the more powerful devices or remotely stream the rendered images for display on the less powerful ones. The system can scale on the number of devices to favor collaborative data analysis.
Chapter IV

The HVR Framework

The work presented in this thesis builds upon the HVR framework. Therefore, this chapter briefly describes its most relevant features. The goal is to provide a overview of the functionality available in the framework, prior to this thesis.

HVR provides interactive high quality volume rendering using consumer graphics hardware. Isosurface and DVR modes are available, but custom rendering modes are possible [39]. It enables real time volume manipulation and configuration. HVR makes it possible to interactively change color, opacity, isovalue and transfer function as well as perform fly-through volume explorations. The core of the framework consists of a GPU-based ray caster, described more in-depth in Sections 2.2.2 and 2.2.3. HVR uses CUDA to perform single-pass ray casting on the GPU.

The framework was originally developed to run on a single PC [27]. However, with the advent of ever increasing data sizes, it was later modified to run on a cluster environment [4] to enable larger data visualization. The distributed version of the HVR framework is the main target of this section and it is then explained more in detail.
4.1 Multilevel out-of-core memory hierarchy

HVR features a multilevel out-of-core memory hierarchy to enable visualization of data sets larger than the total amount of memory. Data is streamed in and out of the different levels independently during ray casting. The blocks needed to produce an image under the current view are selected via view frustum culling and placed into a single 3D texture in GPU main memory. This is the first cache level. Additionally, a second cache level is maintained in CPU main memory for a larger data subset. The objective, as discussed in Section 3.2.4, is to minimize the need to read from disk, an operation that tends to be more expensive, if current data blocks can be reused for later frames. Figure 4.1 illustrates the HVR cache hierarchy.

4.2 Display-aware multi-resolution scheme

HVR uses an octree representation as a multiresolution strategy like the ones discussed in Section 3.2.2. Basically, different resolutions of the data are encoded into an octree
structure in a preprocessing step. Every level of the tree from root to leaves, contain
the data with progressively increasing resolutions. The leaves of the tree have the
original data. This structure serves to selectively provide the most adequate volume
resolution according to the target display resolution. Consequently, it can effectively
bound the working set size on the ray casting and potentially reduce the work load
when targeting display resolutions lower than the data resolution. Additionally, if
physical memory is insufficient to provide the target display resolution, a coarser
octree level is used. It is possible for HVR to mixed resolutions to reduce the memory
footprint and improve performance.

4.3 The rendering process

In a preprocessing step, HVR computes mipmap pyramids for pieces of incoming
volume slices. Then, the octree nodes are reconstructed on-the-fly from the mipmap
pyramids of the 2D slices. Instead of computing a full 3D octree, the mipmap pyra-
mids can be independently computed as the volume is streamed into the system.

During ray casting the required level-of-detail parameter is computed for each
sample. Then virtual-to-physical translation is performed to find the physical location
of every sample individually, resulting in an implicit octree traversal. This is different
from using traversing approaches like depth-first or breadth-first in a node-to-node
sequence. Cache misses are reported in case that the requested data is currently not
in memory at that point of time.

Worth noting is that selecting nodes according to the screen resolution, as ex-
plained in Section 4.2 helps preventing and reduces aliasing artifacts considerably.
4.4 HVR distributed architecture

The distributed version of HVR can execute on clusters of GPUs. Remote rendering is enabled through a thin client interface that is used to display the rendered images and handle user input. All the rendering tasks are done on the server side, where the process is executed in parallel using MPI. The master process executes on the head node and maintains a remote connection with the client via TCP sockets. A fast network interface is assumed within the cluster nodes. Figure 4.2 illustrates the HVR distributed architecture in a more schematic way.

On the server, the load is distributed by assigning work to the slave nodes using MPI. A multilevel cache hierarchy is available in every process to perform the different memory management tasks while volume rendering is performed using CUDA. Once a final image is produced, it can be compressed and sent to the client.
HVR can execute sort-last parallel rendering. As a brief summary from section 2 in sort-last systems the volume is split into pieces in object space regardless of their visibility. Then, each processor performs all the different view transformations and produces a partial image. These partial images are then composited into the final image.

5.1 The serial compositing bottleneck

Partial images need to be blended in a specific order depending on the original domain decomposition. Nonetheless, before this work, sort-last HVR could only do serial compositing. As explained in Section 3.1.4 serial compositing has been recently reported as a major bottleneck that generally hinders the scalability of distributed systems. It has been subject to a considerable research effort that can be found in the parallel rendering literature [19, 47, 58, 63, 79, 90].

Therefore, the goal of the first part of this thesis, is to implement parallel compositing for sort-last volume rendering described more in detail in Section 2. We were particularly interested in implementing direct send and binary swap methods. The remainder of this chapter describes the implementation details for both methods as
Figure 5.1: Direct send compositing using three processors. The first three images depict the state of the rendering buffers with the partial image tiles. The final com- posited image is shown at the rightmost. Image is courtesy of [19]

well as our test results.

5.2 Direct Send

The direct send algorithm is based on the simplest, most natural approach to parallelize the composition stage: Let every rendering node be a compositing node too. Therefore, every node becomes responsible for compositing one or more parts of the final image and for sending the parts of the locally rendered image to be composited by other nodes. Direct send can also be seen as part of a hybrid parallel rendering system since the partial images are essentially a screen space decomposition combined with the data decomposition found in sort-last systems. Hybrid systems are described in section 3.1.5.

5.2.1 Algorithm discussion

Direct send starts right after partial images have been rendered locally by every processor. Next, there are various ways to decompose the task, mainly in the shape of the tiles the final image is split into. There are at least three basic shapes to be analyzed: rectangular, horizontal and vertical stripes. Stripes can be contiguous or interleaved. Each of these alternatives has different implications. Rectangular tiles
minimize the length of the boundaries between the pieces, while stripes interface in longer extents. Contiguous stripes tend to behave more nicely with the underlying memory systems as they generally contain contiguous pieces of memory. Interleaved stripes, in the contrary, tend to distribute the work more equally in case of sparsity as mentioned in the load balancing techniques in section 3.1.7. Many systems define several times more tiles than there are processors so that the rendering load can be balanced. However, this leads to inefficiencies due to high primitive-tile overlap factors and the loss of spatial coherence across tile boundaries. Molnar proposed an equation for modeling the overlap factor for 2D bounding boxes on 2D rectangular tiles [74].

We opted for horizontal stripes since they are easier to implement and fit more conveniently with simpler array structures. Moreover, this pattern helps to reduce the memory footprint by avoiding intermediate buffering as well as complex decomposition routines. In many cases, time complexity of algorithms can be traded off with space complexity. A good example is data replication techniques found on most parallel systems [73, 51, 74]. Nonetheless, space complexity is a very strict concern when rendering large scale data, and avoiding intermediate memory storage can certainly become advantageous.

5.2.2 Implementation details

In our implementation, the height of the final output image is divided over the number of compositing nodes and every node is in charge of a number of pixel stripes equal to height/n. The assigned stripe is dictated by the node id. This is: node number 0 will be in charge of compositing the first stripe while node 1 will composite the second stripe and so on until the last stripe is assigned to node n – 1 as illustrated in figure 5.2. This decision makes memory transfer and copy operations easier to handle. Moreover, tiling the pieces up back into place can be done directly on the MPI calls,
Figure 5.2: Direct send image decomposition. The row width is determined by dividing the height of the image over the number of processes.

as it is done now. We use GPU alpha blending for the actual compositing. Therefore, the images are transferred to the GPU memory as textures and then blended together in the right order. After this, the images are read back from the GPU memory into the CPU main memory. These are generally costly operations but can be amortized if the pipeline is well designed to overlap computation and minimize memory transfers.

After the tiles have been split, every process becomes responsible for compositing a specific stripe of the image and then sends all other local image stripes to their corresponding compositor. Consequently, it also receives all the stripes it needs for the local composition of its piece. All the communication among processors happens in only one stage which reduces the need for synchronization points but disregards the underlying network interface given the all-to-all communication pattern required. A more thorough time complexity analysis can be found in [19]. The worst time complexity is bounded by \( n(n - 1) \) given that each processor can possibly send messages to all other processors. However, it is possible to minimize the required communication so that only overlapping pixels that need compositing are sent over. This strategy effectively bounds the complexity of the algorithm as seen in [79].
Finally, partial image tiles are sent back to the master node for final compositing, which is basically putting the tiles back into place. Again, the stripes pattern makes this step very straightforward as rows only need a pointer where the stripe starts that can be easily derived from the process id.

5.3 Binary Swap

Binary swap is designed to follow a binary tree structure pattern. This means that communication takes place only between two nodes at a time, and the result builds up along $\log(n)$ iterations, where $n$ is the number of nodes involved in the computation. One important limitation of the original algorithm, that also applies to our implementation, is that it requires a power-of-two number of nodes to work. For non-power-of-two systems, the rest of the nodes are not used at all. However, different strategies can be used to circumvent this as done in [90] and [63].

5.3.1 Algorithm discussion

The general overview of the algorithm is illustrated in Figure 5.3. It shows the three levels of the compositing tree with 4 nodes. One important difference from a normal tree-like algorithm lies in the way all nodes are used throughout the execution by splitting the image between the two nodes at every tree level. Level 1 shows the state of the buffers after the first blending has already happened. Every node splits the partial image across the y-axis, sends one of the halves to the communication partner and receives its corresponding half for local compositing. At level 2, the splitting axis is now the x-axis. The smaller pieces are sent to the corresponding nodes accord to the implicit tree. Therefore, immediate neighbors share the bigger messages and as the communication partner gets farther away, the messages get smaller. This feature potentially reduces the network load requirement. Naturally, it is preferable to consider
Figure 5.3: The image shows the entire binary swap algorithm process at every level of the tree for 4 nodes. The arrows show how the binary communication is done between two processors at any time. As the tree traversal advances further, nodes tend to be farther away but the size of the image decreases the required communication. Level 3 shows the final image gathering.
the underlying network interface for a more realistic control of the implementation execution.

5.3.2 Implementation details

Our implementation of the algorithm uses rectangular tiles following the original domain decomposition presented by Ma et al. [47]. However, tiles could have been stripes, following the memory-aware design we chose for direct send. Consequently, the more sophisticated decomposition requires more complex buffering routines. We expect this extra complexity to affect the performance as the number of processors increases.

At the beginning of the execution, the algorithm first computes the node-block distribution and the global compositing order vectors for the correct tracing of the tree structure. The tree is implicitly formed and traversed by calculating the current exchanging partner of each processor at anytime during compositing. Blocks being compositied need to be neighboring nodes in the compositing tree, otherwise, blocks in between will be blended in the wrong order producing incorrect results.

At every step of the compositing loop, each node calculates its exchanging partner and the block in charge of it in the following way:

1. The node in turn calculates the current exchanging partner by tracing an implicit tree on the global compositing vector.

2. For this, the routine $getPartner()$ takes the rendered block index and the stride to calculate the current level of the tree and then return the tree index of the exchanging node. It then searches for the block in the compositing order vector and uses the resulting block id in the global distribution vector to return the current exchanging partner. This procedure is illustrated in figure 5.4.

Then, both parts of the image, the half that is kept for local compositing and the
Figure 5.4: The compositing order vector is used to compute the tree on the fly by calculating the corresponding exchanging block ID at any time. Next, the block ID is searched in the global node-block distribution vector to find the node owner of the block and return this for the binary communication to take place.

half that is sent out to the current partner, are loaded into corresponding buffers. Every process uses a sending and a receiving buffer to overlap two-way communication. Then, the local and the just received images are transferred to the GPU and \( \alpha \)-blended accordingly. We had to modify the compositing routine to keep the alpha values throughout the compositing tree levels. The background should be composited until the very last iteration.

Finally, image tiles are gathered and put together on the master node. This routine reconstructs the domain decomposition structure using the node index to determine the 2D coordinates of the tiles received.

5.4 Experiments

We measured the times of sort-last HVR with the new parallel compositing routines to compare the performance gains against serial compositing. The data sets used and
the corresponding size are described in the table below.

<table>
<thead>
<tr>
<th>Data set</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aligned</td>
<td>$512^3$</td>
</tr>
<tr>
<td>Hydro</td>
<td>$64^3$</td>
</tr>
</tbody>
</table>

For that, we had access to a cluster with 4 nodes. Each node has access to 2 Nvidia Tesla S1060 cards. The CPU is an Intel Xeon X5570 at 2.93GHz with 24GB of main memory. Special interest was put into the compositing times as well as the overall times. We used two different screen resolutions to change the compositing work load.

5.5 Discussion

According to [19], binary swap and direct send only differ in the main compositing time. Image transmission and tile gathering times are considered roughly equivalent. Figure 5.5 shows the main compositing time for the three different schemes. The first thing to notice is that the same curve is described by the two graphs with different view port size. The difference is only noticeable in the absolute time values. The second thing to notice is that the only decreasing tendency is described by the direct send curve. Serial and binary swap describe increasing trends.

Serial compositing receives $n - 1$ full frame images. Exchanging the bigger images naturally takes more time to composite. On the contrary, direct send and binary swap do not exchange full frame images. The first one bounds the image size to $1/n$ of the full frame size across the application execution. Therefore, images composited are reduced in size when the number of compositors increases. In contrast, binary swap divides image sizes into two at every level of the tree making compositing times progressively shorter. This can be advantageous when point-to-point communication
Figure 5.5: Compositing times for the different viewports. Direct send shows the performance gains expected, whereas binary swap’s performance is penalized because of the expensive blending operation done on the GPU.
can be further optimized given that messages are never broadcasted. Furthermore, the tree pattern can be arranged in a way that nodes farther away only exchange the smaller images at further levels of the tree.

The constant number of synchronization points and the decreasing image sizes when the number of compositing processors increases are the keys behind the decreasing curve of direct send. In fact, direct send algorithms tend to behave better when the exchanged messages are smaller. In the worst case, like the one implemented here, direct send can send up to \( n(n - 1) \) messages. However, the images produced need not to be sent to all other nodes. On average the resulting pixels are distributed to \( n^{1/3} \) number of nodes [79], further improving the performance of the algorithm by reducing the amount of communication required. The next version of the direct send algorithm will selectively exchange only the required pixels.

A possible reason for binary swap to have longer compositing times is generally due to the \( n \log(n) \) synchronization points it is characterized for [19]. The algorithm is very sensitive to the lightness or heaviness of the synchronization mechanisms imposed by the underlying system. In our implementations, we use GPU blending for the compositing of the partial images. Binary swap needs to download two images at every level of the tree. Unfortunately, downloading images into the texture memory is an expensive operation. In general, memory transfers between main memory and the GPU are better suited for massively streamed data that can occupy the entire pipeline. Although the image size decreases over time, the depth of the tree does the very opposite. This affects performance in two ways: one is the increasing number of synchronization points as the system grows in number of nodes and second is an increasing under utilization of the memory transfer hardware by uploading smaller images at further levels of the tree. In contrast, the direct send method is only bound to two fixed synchronization points independently of the number of compositing processors. This fixes the cost due to synchronization and does not interfere with the
scalability of the algorithm when \( n \) gets bigger.

Figure 5.6 presents an overview of the overall time including communication and computation costs. The same patterns from the previous graphs are confirmed here by the binary swap rising curve. We were expecting binary swap performance to deteriorate with the increased resolution in Figures 5.5a and 5.6b given the more expensive buffering used for the rectangular domain decomposition.

### 5.6 Chapter Summary and Conclusions

As a summary, we have presented the implementation for sort-last HVR of two well known parallel compositing algorithms: direct send and binary swap. Previously, the framework could only do serial compositing. When the processor count increases, serial compositing becomes the bottleneck in the pipeline as previously discussed in the beginning of this chapter. Compositing times can be reduced by using parallel compositing strategies. We chose direct send and binary swap for two reasons: they come from two different classes of the parallel compositing methods discussed in Section 3.1.4 and more importantly, they are the base for many of the existing optimization strategies. Therefore, we can further extend our implementations to accommodate the more sophisticated methods described more recently in the literature.

We measured the compositing times of our implementations to compare against serial compositing. From our results, direct send was the only one to improve the performance. In contrast, binary swap delivered no performance gains. We concluded that blending on the GPU becomes too expensive as the algorithm needs to transfer partial images multiple times per frame. This amount increases with the height of the compositing tree as more nodes are added to the system. Direct send is not affected by this because there is only one fixed compositing stage that stays constant when the number of nodes increases.
Figure 5.6: Overall times for the different viewports. The differences between the domain decomposition implementation are part of this graph. Rectangular decomposition is more expensive and is expected to further penalize performance for larger systems.
5.7 Future work

The current implementation of the direct send algorithm is actually sending $n(n - 1)$ messages per frame. However, as mentioned before, this can be further improved by exchanging only the pixels that need to be composited. For implementing this, we will need a method to detect when pixels need to composited. Stompel et al. [79] implement direct send following this premise. In their work, a compositing schedule is computed at every frame to calculate the pixels to be sent and then improve the performance of the algorithm by decreasing the amount of communication required.

In binary swap, the performance losses were due to the compositing pattern that increasingly uses multiple calls to the blending routine. This execution pattern becomes unsuitable for the GPU memory transfer model. Therefore, a good strategy to explore further improvements to the algorithm is to implement the blending on the CPU as it is found in the literature. The CPU blending routine is not constrained by costly memory transfers.

Additionally, a second path to explore is choosing different domain decomposition strategies. Although it was not obvious in the presented graphs, the performance of binary swap was also affected by the amount of computation spent in copying memory back and forth to the various buffers. A simpler decomposition strategy is based on contiguous stripes, like it was done in direct send, which has the advantage of avoiding intermediate buffering since reads and writes can be done directly in the same buffer. This strategy has been previously used for tree-like algorithms in [90].

Finally, the system we used for testing is rather small compared to the ones used in the literature. Our results for scalability can be considered as preliminary given that further testing in bigger systems is necessary to establish more definite behaviors. We consider our implementations as initial, as further developing is needed to accommodate more interesting and efficient optimization techniques.
Chapter VI

Interaction mechanisms for HVR

As previously discussed in section 3.3, ultra high resolution displays are desirable by applications dealing with extremely large models. Reasons are based on the benefits of having the increased resolution enabling immersive and more detailed visualization of the model at its natural size. They can be powered by high-end graphics machines with multiple tightly coupled graphics pipelines or clusters of PCs of aggregated performance and resolution. Other reasons include multi-user collaborative interaction capabilities and the possibility to explore new mechanisms of interaction. Some of these mechanisms are summarized and discussed in Section 3.3.4.

HVR targets large data volume rendering by using techniques like adaptive multiresolution rendering and out-of-core multilevel cache hierarchy as described in chapter IV. The ultimate goal behind these techniques is to progressively render larger volumetric data. However, the HVR desktop setting does not take any advantage of the increased resolution. This is why the framework has been ported to work on display array systems. The current implementation follows a sort-first approach to distribute the rendering and display work. Sort-first systems for tiled displays have been recurrently developed with the same purpose in [11, 33, 34, 77].

Sort-first systems are discussed in section 3.1.3. As a brief summary, the premise
Figure 6.1: An overview of HVR sort-first decomposition. Image tiles are split into the working processes for independent rendering and compositing.

on these systems relies on an image-order approach. The domain decomposition takes place in image space. Every process gets a piece of the final image and full rendering pipeline is independently executed on each of them. Consequently, there is no need for compositing. Figure 6.1 illustrates the sort-first decomposition scheme for 4 processors. In sort-first HVR, just like the sort-last version, there is no actual distribution of the volume data. Each node is assigned a piece of the final image plane and can stream in the corresponding data from disk to its own two-level cache hierarchy. Sort-first HVR is based on the same description from Chapter IV.

The current implementation of the system uses MPI to launch two processes per screen. Half of them execute the HVR code, called rendering processes, and the rest, known as display processes, run a simple custom display program based on OpenGL. Custom display array solutions are discussed in Section 3.3.3. At the beginning of both programs, every display process reads in the boundaries of the tile it will display.
to and sends that information to its rendering pair. Then, every rendering process performs the full HVR rendering pipeline and produces a final tile image. This image is later sent to the display pair for display on the screen. This is a preliminary version of HVR on display wall systems. A hybrid version of the framework is currently in development to further improve the performance by mixing sort-first screen decomposition with sort-last data decomposition similar to the work by Samanta et al. [73].

6.1 The HVR user interface

The goal of HVR is to allow researchers to interactively explore and analyze the large-scale 3D datasets obtained by high quality imaging techniques. Moreover, HVR previously targeted typical desktop settings where interaction is performed using mice and keyboards. The client program is a desktop application that handles user input and displays the composited image in the sort-last version. However, since images do not need to be composited for the display array version, the current client shows a 3D wireframe box instead of the real volume. The idea was to provide the user with some graphical representation of the current state of the model for usability reasons. In spite of the highly limited metaphor, it is network efficient since no images have to be transferred back to the client.

Nonetheless, studies discussed in Section 3.3.5 suggest how desktop user interface metaphors break down in display array settings because they require the user to sit and grab the mouse or start pressing key commands. These interacting methods assume the same physical setting of a desktop system. In display arrays, the screen can cover an entire wall making especially suitable for bigger audiences, bigger images and more input sources for collaboration. In this sense, HVR desktop client falls down onto the same interaction challenges. Figures 6.2 and 6.3 show a screenshot and a
Therefore, the goal of the second part of this thesis is to explore a different interaction mechanism to provide more suitable ways of user input on display wall systems. Previous research [2] has shown that direct interaction can help users skills for faster and more accurate manipulation. With the affordability and current power of modern multitouch tablets like the Apple iPad, it is now possible to explore direct interaction for tiled display systems using these devices. This idea is similar to the work of Ponto et al. [66][68]. We believe tablet-like devices has potential for natural, unobtrusive interactivity. Therefore, we have developed a new version of the client for the iPad that allows the user to steer the visualization on the display wall by means of multitouch gestures. We believe this interface fits more naturally with the system setting and can possibly enable the user to get advantage of the space and physical form factor for collaboration possibilities. We now describe the HVR client.
6.2 The HVR Client

The HVR client is an application designed to provide the user with a local interface to the HVR server that runs remotely on a cluster of hardware accelerated PCs. It connects to the server via TCP sockets. It allows the user to change the state of the renderer through various UI controls. When the sort-last version of HVR is running the client also works as a remote viewer of the rendered images produced on the server. It was originally developed in C++ using different libraries and frameworks like VTK and Qt.

The client program is meant to be thin for it does not perform any local rendering. It acts like a remote controller. However, the current desktop implementation is embedded in the framework using shared functionality from the server. The client provides with user interfaces to change the various server parameters. There is a user interface to specify transfer functions. The client can use decompression routines for processing the images from the server in case they have been compressed.

On the main UI interface, the mouse can be used to manipulate the local images as a method to steer the remote visualization. Zooming in and out, panning and rotating...
(a) Desktop client user interface

(b) Multi-touch client user interface

Figure 6.4: Screenshots from both client system user interfaces. The desktop client has lots of different interface controls to really manipulate the visualization on the server.
the model are some of the basic steering facilities. The client is then in charge of calculating the different parameters like the rotation matrix, the new coordinates and the new scaling factor, keep track of the current values and send them to the server. Figure 6.4 compares the user interfaces of both systems. The current implementation of the multitouch client offers comparable steering facilities.

There is a custom network protocol designed for the communication between the client and the server. A custom module of VTK was used for the network programming. The socket is a opaque object that abstracts the end point of a connection established over a network. It is specified by an IP address and a port number. This combination helps to identify the application that established or accepted the connection. The protocol designed for the client server communication consists of a stream of combinations of message identifiers and data. It is a very compact network protocol that works directly with byte streams. Although possible, no special ciphering or encryption is applied to the data. The \texttt{HVRMessage} class is in charge of buffering the messages before sending and of extracting the data. The class can automatically add any kind of data type to a vector-like container any kind of primitive data type or primitive data type arrays. The message is converted to a byte stream and sent through the socket.

\section{6.2.1 The client development minimum requirements}

For the new client to be considerable functional, it has to provide the minimum user interface requirements: rotation, panning and zooming. There should be a way to specify simple opacity transfer functions. Additionally, there should be a way to show a wireframe box to provide some guide for the local state of the visualization just like the desktop client. This basic manipulation of the visualization should be possible by simple gestures. Moreover, the application needs to connect through the network interface using TCP sockets and conform to the network protocol used by the server.
Figure 6.5: The application design is functionally split into three software layers: main, network and display. Each layer is in charge of providing specific functionality to the system. Touches are perceived on the display layer, transformed in the main layer and sent to the server using the network layer.
6.2.2 Application Design

The application design is split into three different layers: main, display and network. These layers are modeled as objects inherited from the NSObject class that provides the classes with a base functionality. These objects are interconnected via protocols, delegates and instances. Protocols and delegates are indicated by the blue arrows on figure 6.5. They are similar to Java interfaces and C++ virtual classes. A protocol declares methods that any class can implement. A delegate is the class that has chosen to implement the method.

In our client application, the main layer contains the HVRViewController is called by the main routine. It takes control of the screen by building up the user interface. When the HVRMainController object gets created, it contains one instance of the HVRNetworkClient and of the GLViewController. Figure 6.5 shows an schematic diagram of the overall design of the application. The general work flow starts by receiving touches on the display layer, transforming the touches into parameters in the main layer and finally sending the parameters via the network layer.

6.2.3 Network Layer

The network layer of the application is in charge of managing the connection with the server. This layer also encapsulates all the network functionality needed to pack and send as well as unpack and receive messages to and from the server.

The HVRNetworkClient is in charge of handling all high-level network connections. The class has the appropriate routines to send and receive the different parameters that can be changed on the server. Once a connection has been established, the class forks out a background thread that asynchronously listens for network activity. When the background thread calls back, it waits until the main thread has finished receiving the data. This avoids the possible race conditions but sacrifices more substantial
asynchronous execution. The rationale behind it is pure simplicity. Sending operations are performed on demand every time changes on the state of the application are registered.

The *cmClientSocketUtilities* class is a lower level interface. It contains the actual connecting, sending and receiving routines. It creates and manages the current instance of the socket object. It also keeps a local version of the data structures. The reason for replicated copies is that in this way, the system is allowed to update the local data structures and send the changes immediately after, while simultaneously receiving changes from the server and writing the changes to the replicated copies that will later be synchronized.

### 6.2.4 TCP Socket Connectivity

Connectivity between the server and the client is performed via TCP sockets over WiFi. The older client used a custom version of VTK network library. However, we decided to change the choice for cross-platform alternative that is available on mobile devices. The goal is to consider porting the client program to other mobile platforms later. We would like to extend the possibilities for interaction with the HVR remote server. Fortunately all libraries are mostly wrappers around BSD sockets that just provide extra layers for ease of use. Consequently, there is some level of compatibility across libraries which makes the decision a bit easier to take. Mixing is not desirable in any case.

Therefore, we created our own wrapper called *HVRSocket*. The wrapping scheme allows selective building of the HVR server using the different network APIs. It is now possible to use any version of the client by recompiling the server with the appropriate configuration. We chose to use the POCO C++ libraries as it is possible to build static libraries for iOS out of the box. We could not find such a convenience in the BOOST Asio library. Inside the *HVRSocket* class different precompiler flags can be
The client communicates with the server head node via WiFi using the network protocol described.

changed to retrieve different versions of every function for connecting, sending and receiving on the current socket accordingly. This class wraps every socket related function call used by the server so much of the code can be left intact or with minor modifications.

The usage of C++ libraries involves mixing Objective C with the C++. Fortunately, this is possible via the objc++ compiler included with the Xcode development platform. Moreover, the API core functionality is only available for Objective C. Thus, although it was desirable to keep the whole development in C++, we still had to use Objective C for the user interface. The new network setting is illustrated in Figure 6.6.

6.2.5 The Client-Server Protocol

The client-server protocol used is based on combinations of message identifiers and data pushed into a queue-like container. A message dictionary header called cmMessages contains all the meaningful message identifiers. The rule is that every transferred message has a message identifier preceding the data array. Then, depending on the message identifier, there are different combinations of identifiers and data arrays that are known for both the client and the server.
We changed the protocol transport class from using VTK streams to *HVRMessage* objects. This class was already in the system. It was previously used for the MPI messages sent between server processes only. We now use it for the messages transferred between the server and the client. The *HVRMessage* class allows to add various types of data, scalars and arrays, to the message before sending. It also has routines for extracting from the messages received. The added data is internally stored as a stream of bytes using standard vector data types. However, the design was chosen to encapsulate the actual implementation of the data stored in *HVR Message*, so this could be changed later on without propagating those changes to the other parts of the system.

### 6.2.6 Display Layer

The display layer is comprised of the *GLViewController*, the *EAGLView* and the fragment and vertex shaders. It is in charge of managing the display. This includes showing the local visualization according to the current display mode: wireframe or low-resolution. Its second major responsibility is to catch user events, interpret and update the *HVRmodel*. It is also responsible for notifying the *HVRNetworkClient* via the *HVRMainController* so the new values can be sent to the server.

The *GLViewController* class is used for all display management tasks. It uses OpenGL ES 2.0 for rendering purposes. Depending on the display mode selected, it is possible to render a wireframe box or low resolution images received from the server. Figure 6.7 shows screenshots from the client configured to use each mode.

When the object is created and initialized, it triggers a series of configuration routines that set up the OpenGL rendering context via the *EAGLView* component. It also starts the display loop and compiles the fragment and vertex shaders. More responsibilities of this class include setting up the vertices, texture coordinates and colors into arrays as well as sending them as uniforms to the shaders. In addition,
Figure 6.7: The client display modes. In 6.7a the client only shows a wireframe box to accelerate the process and still provide some hint on the state of the visualization. In 6.7b the client displays down sampled images received from the server it is responsible for tracking the HVRModel state display parameters and update in case of detected changes. It also contains several routines for creating textures from compressed images and byte arrays.

**EAGLView** is a very special subclass of **UIView**. It creates the OpenGL frame buffer and configures the current rendering context accordingly. It implements the routines for managing the user touch events. Consequently, the class has data structures and routines to track the user input and compute application meaningful quantities. The current instance of the class computes the rotation matrix using the distance tracked with one finger as the angle of rotation. The axis of rotation is calculated by the vector traced with a single finger. The class also computes the scaling factor changes using the distance between the fingers when pinching in and out and panning with the distance traced by a single finger double tapping and dragging.
6.2.7 Low resolution images

One of the two advanced features available on the multitouch client is the possibility to show lower resolution images from the scene rendered on the server. This mode replaces the wireframe box with a fixed quad mapped with the 2D textures created from the received images. We believe that this method provides a more suitable metaphor for remote steering on the client. Figure 6.7b shows a screenshot from the client executing the low resolution mode.

For our current implementation, the low resolution images are computed by averaging the pixels. This routine is executed only on the nodes that had something to render. The downsampling calculates the smaller size of the tiles and the pixel mapping according to the iPad screen size which is then kept fixed throughout the entire execution. The pixel mapping indicates how many pixels in the big images become a single pixel in the smaller ones. Then, every node averages its tile image according to this pixel mapping in parallel. Next, the downsampled images are sent to the master process for final compositing via MPI. Finally, the image is sent to the client.

6.2.8 Transfer function interface

The last feature included in the current implementation of the multitouch client is the capability to specify changes to the current opacity function used on the server. The x and y axes of the screen are encoded as the density and the opacity values correspondingly. Therefore, it is possible to explore this 2D space by first double tapping and then dragging two fingers over the screen.

The new values correspond to the normalized x and y coordinates of the two fingers. We opted for using two fingers because we believe it is a natural way to specify simple threshold functions represented by two points. For now, the user
Figure 6.8: The opacity transfer function can be changed by double tapping and then dragging two fingers over the screen. The screen space maps the x and y axes to the density and opacity values in the data set. The corresponding threshold functions are shown at the bottom of the figure.

The interface is very simple and shows no tracking cues for the current values. For the next version, the interface should include a way to easily track the current state of the values on the iPad screen so it is easier to make changes accordingly. Figure 6.8 illustrates the opacity transfer function interface.

### 6.3 Interactivity tests

The application frame rate is the most meaningful statistic for an interactive visualization software. We have tested five different datasets on the current implementation of HVR running on a display wall system. We then measured the most important timings and calculated the corresponding frame rates in an attempt to model the interactivity of our system. The test session includes loading each data set and basically performing all possible manipulation as well as changing the transfer function.
6.3.1 System configuration

The hardware configuration is illustrated in figure 6.6. The display wall is composed of 40 individual screens with an individual resolution of 1360 x 768. They are arranged in a 10 x 4 rectangle fashion. The displays are driven by a cluster of ten PCs. Each PC node has two GeForce 285 GTX and a dual quadcore i5 CPU with 12GB of main memory. Locally, every node drives four screens. We also use twenty more identical nodes to execute the HVR rendering process. All nodes are connected to the same 10GB Ethernet network. A dedicated WiFi network is available for wirelessly connecting to the system. The client program is run on a first generation iPad and we use the dedicated WiFi to the iPad client to the server.

6.3.2 Results

We ran the system many times with the different client display modes using different data sets. Screenshots from the different data sets can be found in Section 6.6. The numbers are specified in frames per second. We have included both the minimum and maximum frame rates, as well as the mean of all the runs. Table 6.1 summarizes all the results of the various runs.

The top part of the table shows the frame rates measured on the server. The included time lapse starts from the moment the socket registers the incoming message until a the sever sends a response to the client. Under the low-resolution image mode, this includes the time spent for downsampling, compositing and sending the image to the client.

The client side rates are shown in the bottom part of the table. Here, the time measured starts from the moment the client has registered a new touch event until the screen has been updated with a new image. For the wireframe mode, the timer stops when client has received a dummy response from the server. We have included
Datasets | 101 | Aligned | Hand | Skull | Hydro
---------|-----|---------|------|-------|-------
Modality | EM  | EM      | CT   | CT    | Synthetic
Server   | Wireframe | Min 6.143 | 0.2183 | 12.0305 | 13.3158 | 5.8363
         |         | Max 22.6367 | 22.8236 | 22.7555 | 22.544 | 22.7998
Low-res  | Min 0.5086 | 0.3967 | 0.1895 | 0.1832 | 0.1285
         | Max 7.0467 | 7.0374 | 9.6612 | 6.1329 | 5.7007
         | Mean 4.6484 | 3.0091 | 4.75 | 2.473 | 2.8579
Client   | Wireframe | Min 0.5084 | 0.1653 | 0.4783 | 0.4639 | 0.5236
         | Max 79.7384 | 67.5036 | 66.432 | 64.973 | 73.153
         | Mean 14.4857 | 10.5182 | 16.104 | 8.0717 | 15.899
Low-res  | Min 0.261 | 0.3253 | 0.4155 | 0.3476 | 0.3455
         | Max 38.2921 | 23.4417 | 24.7788 | 39.8803 | 41.8691
         | Mean 4.7028 | 2.0782 | 6.6895 | 6.6875 | 6.5872

Table 6.1: The obtained frame rates measured in frames per second (fps)

the minimum and maximum measured rates to illustrate the variation range. We believe it gives a better idea on how slow the system can run at times.

Furthermore, as expected, the extra computation due to the image downsampling and compositing routines accounts for the lower frame rates achieved on the low-resolution display mode. In this mode, rates measured from the server vary from less than 1 to 9 fps, but the average stays in the range of 2 to 4 fps. This means that interactivity is compromised when the low-resolution mode is activated the achieved frame rates are not enough to deliver smooth interaction. On the other hand, the absence of these routines effectively shortens the execution times in the wireframe mode. The corresponding rates vary from less than 1 to 22 fps with a mean of 14 to 16 fps. On the client measurements, wireframe mode frame rates vary between less than 1 to 79 fps with an mean rate of 8 to 16 fps. In contrast, the rates in the low-resolution image mode tend to be lower in the range of 0 to 41 fps with an average rate of 2 to 6 fps.

Overall, the frame rates of the whole system are highly variant and irregular. We believe this is due to wireless network having to resend dropped data packets.

The results of the runs show that even though the mean frame rate can be con-
sidered interactive, the performance delivered is very inconsistent. This can translate
to rough user interface experiences as the system performance is bounded to a wider
range and in the worst case scenario the system runs too slow to be interactive.

6.4 Chapter Summary and Conclusion

The object in the second part of this thesis is to develop suitable interaction mech-
anisms for the version of HVR that runs on the display wall. Given that HVR user
interaction is done via a client program. We have developed a new client application
that can run on the tablets and enables interaction that can take advantage of the new
environment settings. The development of this new client application is motivated
by the lack of naturalness of desktop interaction mechanisms recurrently reported on
the literature and discussed in Section 3.3.4. Moreover, desktop user interfaces do
not leverage the ultra higher screen resolution and the enlarged physical form factor.
Therefore, interaction mechanisms for such systems should let the user freely move,
engage into discussion with other people and approach bigger audiences without com-
promising functionality. The focus should be on the visualization rather than on the
interaction mechanics.

The current implementation of the client application was developed for the iPad.
It lets the user steer the visualization by simple multitouch gestures and provides
visual feedback by displaying either a wireframe box or downsampled images from
the main visualization. It is also possible to change the opacity transfer function by
using the touch coordinates encoded as values for the density and opacity.

The overall design of the application is split into three layers: main, network and
display. The main layer manages the user interface and serves as a bridge between
the other two. The network layer handles all communication with the server via TCP
sockets. It can send and receive the various types of messages, as well as add and
extract the message contents. The display layer manages the user input and is the one in charge of displaying either the wireframe box or the low resolution images.

We reported our initial interactivity tests that measured the overall times since a request has been done until a response has been emitted on both the client and the server. The preliminary results show highly variable frame rates. When the system is running on wireframe mode, it can reach an average frame rate of 8 to 16 fps. The low resolution image mode drops this rate to 2 to 9 fps. This performance loss is due to the extra computation caused by downsampling and compositing the smaller images. As a final note, it would be interesting to further show how the time spent is distributed among the different tasks and see if this stage can be further optimized.

6.5 Future Work

The next version of the framework, currently in development, will have a hybrid rendering approach. This means that it will combine the sort-first screen decomposition with the sort-last data decomposition similar to the work by Samanta et al. [73]. We expect the system to further improve the overall performance because the data does not need to be redistributed at the beginning of every frame. Moreover, this change will introduce a compositing stage and we expect to use the parallel compositing implementations developed in Chapter V. For the next version of the client there are a couple of improvements and ideas to explore that we explain next.

Non-empty pixels transmission

First of all, the drop in the frame rates when the low resolution image mode is activated is due to the extra computation of compositing and the transmission of the image that potentially includes a great amount the empty space. The latter can be further optimized by only putting together the non-empty tiles and sending this to the
client. This amount is potentially less than what the server is sending to the client in the current implementation. The client can then add the corresponding empty space or just display the non-empty image with a better resolution and usage of the iPad screen. Similarly, we would like to enable data compression and decompression capabilities to further reduce the network load.

Transfer functions

The current transfer function is high limited and we would like to add more sophisticated and usable interfaces. For now, it is only possible to change the opacity function. We would like to add some way to track the current values by adding some semitransparent cues on the screen that indicate the last position of the fingers on the screen and the values used. We believe in this way, the user will be able to have a better understanding on the way the interface works and how to change it accordingly to achieve the desired results. Moreover, we would like to add a way to specify RGB transfer functions. A potentially intuitive way to do this is by adding paintbrush-like capabilities. In this way, the user could choose a color from a palette and "paint" what is visible according to the current opacity. We believe the paint-brush metaphor is well understood and could improve the ease of use for transfer function synthesis.

Collaborative visualization

Additionally, we would like to explore collaborative visualization sessions by adding allowing multiple iPad clients to the system. In this way, different users could use their clients to manipulate the visualization and the parameters used, but also to enable paint-brush capabilities to circle areas of interest and make notes to specific parts of the visualization. Moreover, ideas similar to the work by Rekimoto [68] could also be implemented. There could be a way to pick and drop elements from the tablet screen to the display wall. Also, we would like to explore a similar idea to the one
by Ponto et al. 66 where the clients could join and leave the visualization session on-the-fly. In this way, the users could plan and prepare the session in an offline, then join an active session and finally pick and drop the prepared elements from the tablet to the display wall.

Finally, tablet devices like the iPad feature gyroscopes, accelerometers, cameras and microphones. We would like to investigate interesting ways to use them as multimodal interaction that combines direct and indirect interaction. We could possibly let the user tilt the device to perform rotation operations or use the camera and microphone for remote collaborative sessions.
6.6 Photo gallery

Figure 6.9: Screenshots of the low resolution images of the various datasets running on the iPad simulator
Figure 6.10: Screenshots of both visualizations: from the iPad client and the display wall. The images on the iPad were zoomed in using the built-in functionality.
Figure 6.11: Screenshots of the low resolution images of the various datasets running on the iPad simulator
Chapter VII

Summary and General Conclusions

The goal of this thesis is to improve the scalability and usability of volume rendering systems that target visualization on display arrays. In the first part, we implemented two parallel compositing algorithms: binary swap and direct send. The objective was to provide the framework with parallel compositing capabilities to reduce the time spent in the compositing stage. Previously, only serial compositing was possible. However, serial compositing does not take any advantage of the distributed architecture of the system and becomes a bottleneck when the processor count increases. In our implementation of the binary swap algorithm, the results of our tests revealed performance penalizations. We believe this is due to an expensive blending routine. This routine downloads the images into the graphics hardware memory. Downloading images to the graphics memory works better for bulky transfers. At every frame, binary swap uses the routine multiple times according to the current number of compositors. This characteristic had a very negative impact on the performance of the algorithm. On the other hand, the direct send behaved as expected and did result in improved timings over the serial compositing. Because the actual blending happens only once per frame, the performance is not affected as much as that of binary swap.

In the second part of this thesis, we presented a new HVR client that executes on
the iPad. The main goal was to provide the framework with interaction mechanisms that can better utilize the resources of display wall systems. Previously, the desktop client offered an interface for mice and keyboards. However, tiled display applications have different user requirements. Desktop interaction mechanisms are less applicable to the new problem constraints and motivated the use of tablet devices given their great degree of availability and interactivity. We found research that supports direct interaction (touch, body gestures, etc.) over indirect tracking (mice, keyboard, etc.) for ultra high resolution displays. Moreover, the success of devices like the Apple iPad, have shown the richer interactive possibilities one can find on them opening windows of possibilities to other fields. This new client provides the user with well known multitouch gestures to steer the HVR visualization on the display wall. The user is then free to touch, move and walk around.
REFERENCES


