Stony Brook University

Massive Data Management for Distributed Volume Visualization

a Dissertation presented

by

Susan Lavis Frank

to

The Graduate School

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

December 2008

Stony Brook University

The Graduate School

Susan Lavis Frank

We, the dissertation committee for the above candidate for

the Doctor of Philosophy degree,

hereby recommend acceptance of this dissertation.

Distinguished Professor Arie Kaufman, Dissertation Advisor
Department of Computer Science

Professor Klaus Mueller, Chairman of Defense
Department of Computer Science

Professor Michael Bender, Committee Member
Department of Computer Science

Professor Xiangmin Jiao, External Committee Member
Department of Applied Mathematics & Statistics
Stony Brook University

This dissertation is accepted by the Graduate School.

Lawrence Martin
Dean of the Graduate School

Abstract of the Dissertation

Massive Data Management for Distributed Volume Visualization

by

Susan Lavis Frank

Doctor of Philosophy
in
Computer Science

Stony Brook University

2008

As memory and compute power become more affordable, the size of data sets has grown rapidly due to the improved accuracy and memory capacity of data collecting devices. A massive volume is one or more orders of magnitude larger than main memory on a single visualization engine. We present a framework which automates load balancing volume distribution and ray-task scheduling for parallel rendering of massive volumes. Our solution combines static network distribution with dynamic ray-task scheduling within each render node. Our framework is designed to automate the volume distribution process for parallel ray directed algorithms including ray casting and ray tracing on a heterogeneous mix of hardware. This parallelization may occur within a new custom-designed hardware, on multiple processing units, on a distributed super-computer, or on a visualization cluster.

High resolution and very large scale volume data sets frequently contain large portions of empty space. The main bottlenecks for a distributed volume rendering system involve moving data across the network and loading memory into rendering hardware. Space leaping techniques are traditionally used to reduce the impact of wasted space at rendering time. We remove empty space early in the pipeline in order to reduce the bandwidth required to send volume data across a cluster, as well as processing cycles required for loading volume

data into local memory during rendering. We introduce the *flex-block partition*, which gives an unambiguous ray traversal order for any view direction. The scene is partitioned into cells which each contain one or more closely cropped blocks of volume data, or *flex-blocks*. Scene partitioning is driven by tightly cropped subvolumes to allow volume boundaries to be followed in a natural manner.

Managing massive volumetric data requires the redesign of traditional algorithms for out-of-core implementation. The *out-of-core region growing* algorithm performs region growing on a consecutive group of slices, or slab. The output of is a mask volume which is used for volume segmentation and empty space cropping. The last slice of the mask volume supplies a set of independent region seeds for region growing on the next slab. Special consideration is taken to deal with regions splitting and/or merging within a slab. Our *slab-projection slice* is used to gather non-empty region information for one slab of data at time. Our out-of-core bricking is used to create cropped subvolumes, or bricks, sized for target memory. A directed acyclic graph representing the relative distance of the bricks for a given viewpoint and direction is built concurrently. Our *slab-projected kd-tree partitioning* uses a series of slab-projection slices to partition data between nodes for parallel rendering applications.

The problem of load balancing network distribution (LBND) is defined as that of partitioning volumetric data for distribution across a cluster to achieve good load balancing without violating priority order constraints required by image composition in parallel ray casting. We have explored dynamic programming (DP) solutions to the LBND problem that are particulary applicable to scenes with large portions of unevenly distributed empty space. The first, *brick grouping*, inputs a directed acyclic graph (DAG) of data bricks and finds an optimal partition with respect to a particular view direction, set of hardware constraints, and cost model. We attempt to minimize a cost function that reflects the end-to-end rendering cost on a cluster. The algorithm average time complexity is reduced by limiting the average number of bricks, which is a function of the brick size. The second DP solution,

moving walls, inputs slab projection slices and incrementally builds a flex-block partition by locally optimizing for a given set of constraints and cost model. The output is a view-independent, approximate solution. The potential size of the problem is greatly reduced by imposing restrictions on the cut plane ordering. Empirical results indicate that good load balancing is achieved using these algorithms.

We present a dependency graph scheduling algorithm for distributed ray tracing. Our *cell-tree* gathers clusters of eye, shadow, reflected and refracted rays into a compact description of all ray dependencies. Our *cell-tree peeling* algorithm exploits frame-to-frame coherence. It determines a memory caching schedule from ray traversal order of the previous frame, which is encrypted in the cell-tree to predict a good schedule. This algorithm works with any scene partition which allows unambiguous priority order for ray traversal.

Our flex-block partition approach removes empty space early in the pipeline. The cell-tree is constructed with virtually no overhead at run-time during ray tracing. It is a concise representation of ray-traversal dependencies between arbitrary blocks of data. In our framework it is used for local ray-task scheduling within each render node. The dynamic programming approach automates load balancing for a given set of system parameters. Our algorithms have been designed to be independent of any specific cluster configuration. Algorithms have been verified with several large high resolution volumetric data sets on the Stony Brook Computing Cluster.

To my husband, Ken,

and my children, William, Tommy, Bobby, Katrina, and Michael.

Contents

List of Tables **xii**

List of Figures **xiii**

Acknowledgments **xvi**

List of Publications **xviii**

1 Introduction **1**

1.1 Massive Data Applications 2

1.1.1 Anthropology Data 3

1.1.2 Geological Data 4

1.1.3 Medical Imaging 4

1.2 Stony Brook Visual Computing Cluster 5

1.3 Thesis Contributions 8

1.3.1 Out-of-Core Data Management 8

1.3.2 Load Balancing 9

1.3.3 Dynamic Programming 10

1.3.4 Dependency Graph Acceleration 11

1.4 Thesis Organization 12

2 Background **13**

2.1 Direct Volume Rendering 14

2.1.1 Ray Casting 14

2.1.2 Segmentation 16

2.1.3 Volume Rendering Architectures 16

2.2 Parallel Ray Casting 18

2.2.1 Visualization Clusters 18

2.2.2 Image Compositing 19

2.2.3 Hardware Compositing 20

2.2.4 Scene Partitioning 21

2.3 Volume Rendering on Graphics Accelerators 23

2.3.1 Multipass Partitioning 24

2.3.2 DPMPP 25

2.4 Ray Tracing 25

2.4.1 Rendering Model 26

2.4.2 Global Illumination 27

2.4.3 Ray Traversal Acceleration 28

2.4.4 Parallel Ray Tracing 28

2.4.5 Ray Tracing Systems 31

2.5 Summary 31

3 Framework for Interactive Massive Volume Visualization **33**

3.1 System Functionality 34

3.2 Data Management 35

3.2.1 Early Data Reduction 37

3.2.2 Dependency Graph Data Structures 38

3.2.3 Flex-block Partition 39

3.3 Load Balancing Data Distribution 40
3.4 Dynamic Task Scheduling 41
3.5 Test Environment 42
3.6 Interactive Visualization Results 45
3.7 Discussion 46

4 External Memory Solutions **48**
4.1 Challenges in Massive Data Management 48
4.1.1 Distributed Preprocessing 49
4.1.2 Empty Space Cropping 50
4.2 Out-of-Core Data Management Pipeline 50
4.3 Slab-projection Slice 52
4.4 Out-of-Core Bricking 54
4.5 Slab-projected Kd-tree Partitioning 55
4.6 Region-of-Interest Cropping 57
4.7 Out-of-Core Region Growing 58
4.8 Early Data Reduction Results 60
4.9 Summary 62

5 Load Balanced Network Distribution **64**
5.1 Distributed Volume Rendering Overview 65
5.2 Load Balanced Network Distribution Problem 66
5.2.1 Graph Partitioning Software Packages 68
5.2.2 GPU Multipass Partitioning 68
5.3 Dynamic Programming 69
5.3.1 Brick Grouping DP Overview 69
5.3.2 Objective Function 71

5.3.3 Locally Optimal Solution 72
5.3.4 Example Stage 74
5.3.5 Algorithm Analysis 74
5.4 Results 75
5.4.1 Test Environment 76
5.4.2 Timing 77
5.4.3 Load Balancing Results 79
5.5 Summary 82

6 Moving Walls DP Data Distribution 84
6.1 Overview 84
6.2 Cost Function 85
6.3 Moving Walls Algorithm 87
6.3.1 Building the Flex-block Tree 88
6.3.2 Efficiency Considerations 88
6.3.3 Comparison to Brick Grouping 89
6.4 Results 90
6.5 Summary 95

7 Cell-tree Scheduling for Ray Tracing 98
7.1 Ray Traversal 99
7.1.1 Ray Queues 100
7.1.2 Problem Definition 100
7.2 Cell-tree 101
7.2.1 Cell-tree Construction 102
7.2.2 Cache Savings Links 104
7.3 Cell-tree Peeling 106

7.3.1 Definitions . . . 107
7.3.2 Algorithm Description . . . 107
7.3.3 Algorithm Correctness . . . 109
7.3.4 Worst Time Bounds . . . 110
7.4 Architecture Simulation . . . 110
7.4.1 GI-Cube Ray Tracing Architecture . . . 111
7.4.2 DSP Cell-tree Scheduling Simulation . . . 112
7.5 Results . . . 114
7.6 Summary . . . 118

8 Conclusions **120**
8.1 Summary of Contributions . . . 121
8.2 Summary of Results . . . 122
8.3 Near-Term Future Work . . . 123
8.4 Extended Vision . . . 125

Bibliography **126**

List of Tables

1.1 Comparison of visualization clusters 6
4.1 Preprocessing time 63
5.1 Partition preprocessing time 78
5.2 Empty space and slice preprocessing time 79
6.1 Rendering time for different partitions 91

List of Figures

1.1 High-resolution X-ray Computed Tomography Data 3
1.2 Low resolution Visible Male 5
1.3 Visible Korean raw data 5
1.4 Stony Brook Visual Computing Cluster 6
1.5 Data sizes compared to cluster memory 7
2.1 Phong shading 15
2.2 Parallel ray casting 20
2.3 Ray tracing physics 27
2.4 Pseudo-random ray traversal 30
2.5 Hybrid object image ray tracing. 30
3.1 Visualization framework block diagram 34
3.2 Distributed rendering data management pipeline. 36
3.3 Flex-block tree 40
3.4 HP MDS Cluster block diagram. 43
3.5 VolumePro1000 44
3.6 Image composition 44
3.7 Volume rendering of seismic data 45
3.8 Scalability of frame rates 46
4.1 Distributed preprocessing compared with single node preprocessing . . . 51

4.2 Out-of-core preprocessing pipeline 52
4.3 Slab-projection of the Visible Korean lungs 53
4.4 Out-of-core region growing seed slice 59
4.5 Composited full resolution four-channel Visible Male 61
4.6 Volume rendered teeth images. 62
4.7 Volume rendered fossil images. 62
4.8 Data storage reduction 63
5.1 Distributed ray cast rendering block diagram 65
5.2 Partition of a DAG of bricks into a DAG of cells 70
5.3 Stage candidate transitions 74
5.4 GPU-rendered segmented regions of the Visible Korean 77
5.5 VolumePro1000-rendered images of the Visible Korean 78
5.6 Data distribution load balancing 80
5.7 Load balancing scalability 81
6.1 Comparison of partitions with different algorithms 89
6.2 Sample partition slabs 93
6.3 Preprocessing time comparison 94
6.4 Load balancing and scalability 94
6.5 Visible Male distribution 95
7.1 Ray-cell dependencies 102
7.2 Cell-tree construction 103
7.3 Redundant dependencies 105
7.4 Cell-tree peeling 108
7.5 GI-Cube ray tracing PCI board. 111
7.6 Proposed ray tracing architecture PC board. 112
7.7 Proposed ray tracing architecture data structures. 114
7.8 Ray traced images for algorithm testing. 115

7.9 Cell-tree sizes 115
7.10 Cell-tree scalability 116
7.11 Performance comparison 117

Acknowledgments

I would like to begin by thanking my advisor, Distinguished Professor Arie Kaufman, for his continual guidance, advice and support. Through years of dedication and hard work, Professor Kaufman has developed a world-class research environment that has given students the opportunity to work with top researchers and state-of-the art equipment. I am thankful for the freedom he has given me to pursue interesting research problems. I am proud to join his rich legacy of Ph.D. graduates. I am also thankful to my committee members, Klaus Mueller, Michael Bender, and Xiangmin Jiao for their constructive critique, assistance, and support.

I want to thank all my colleagues, friends and staff members at the Computer Science department who have helped me in these years. I want to give special thanks to Stella Mannino for her generous help. I owe special thanks to Bin Zhang for all his support with the Stony Brook Visualization Cluster. I want to thank Brian Tria, Kathy Germana, Betty Knittweis, Cynthia Scalzo, and Shakeera Thomas for their help and support. I want to thank all my research colleagues for all the fruitful discussions and encouragement: Feng Qiu, Zhe Fan, Joseph Marino, Kaloian Petkov, Abhijeet Ghosh, Sarang Lakare, Huamin Qu, Kevin McDonnell, Kevin Kreeger, Nan Zhang, Xiaoming Wei, Suzi Stover, Kevin Kreeger, Frank Dachille, Kevin McDonnell, and Wei Hong. I want to thank Professor Anita Wasilewska, Professor Leo Bachmair, and Professor IV Ramakrishnan for their guidance in supervising my recitation class teaching. I also want to thank the faculty at Stony Brook

for all the interesting courses and for their encouragement, especially Professor Esther Arkin, Professor Hussein Badr, Professor Joseph Mitchell, Professor Hong Qin, Professor David Smith, Professor Eugene Stark, and Professor Larry Witte. I am also thankful to my research grant providers: National Science Foundation, and the people at Hewlett Packard, IBM and TeraRecon.

I am deeply thankful to my parents, who instilled in me the desire to learn from a young age. I would also like to thank my brothers and sisters, who have encouraged and motivated me over the years. I would like to thank my children, William, Tommy, Bobby, Katrina and Michael, who have taken this journey with me. I cannot begin to enumerate all the sacrifices they've made on my behalf, yet they have been there with words of encouragement at every turn. Finally, I want to thank my husband, Ken without whom this Ph.D. would not be possible. I cannot write in words how much support and encouragement I have received from him through these years and I am truly thankful to have him by my side.

List of Publications

S. Frank and A. Kaufman, Out-of-Core and Dynamic Programming Strategies for Data Distribution on a Volume Visualization Cluster, *Computer Graphics Forum*, 2009.

S. Frank and A. Kaufman, Dependency Graph Approach to Load Balancing Distributed Volume Visualization, *The Visual Computer, International Journal of Computer Graphics*, 2009.

S. Frank, Framework for Interactive Massive Volume Visualization, *Grace Hopper Celebration of Women in Computing, PhD Forum*, October, 2008, Denver Colorado.

S. Frank, A Dynamic Programming Approach to Kd-Tree Based Data Distribution, *Supercomputing, Student Poster*, November, 2007, Reno, Nevada, USA.

S. Frank and A. Kaufman, Distributed Volume Rendering on a Visualization Cluster, *CAD/Graphics*, December, 2005, Hong Kong, China, pages 371-376.

S. Frank and A. Kaufman, Distributed Volume Rendering on a Visualization Cluster, *Computer Graphics International, Poster*, June, 2005, Stony Brook, NY, USA.

S. Frank and A. Kaufman, Dependency Graph Scheduling in a Volumetric Ray Tracing Architecture, *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, September, 2002, Saarbruecken, Germany, pages 127-135.

Chapter 1

Introduction

The goal of this research is to provide mechanisms which improve the efficiency and scalability of distributed volume visualization. Volume visualization is a key technology for data exploration. Realistic lighting and interactivity enhance the value of the visualization process. State-of-the-art data sizes have consistently exceeded available memory even on the most up-to-date computer systems. Increased memory capacity opens tremendous potential in what can be visualized, but also introduces challenges in managing data on distributed systems. Good load balancing requires careful distribution of data and run-time task scheduling, and these processes must adapt to different cluster configurations. For a scalable system, each processing unit, or node, performs an approximately equal portion of the illumination and/or rendering tasks.

Parallelization may occur within custom-designed hardware, on a distributed supercomputer, on a multi-core PC, or on a visualization cluster. In parallel ray casting, a partial image is rendered for each subvolume, and composited into a the final image using alpha blending. The contribution of each image depends on the relative location with respect to the eye, so a partial ordering of these images must be determined for any given view

direction. Data distribution should partition render-node contributions so that each render-node operates independently, avoiding inter-node dependency latency.

In parallel ray tracing, a recursive operation is used to compute reflective, refractive and shadow rays. Each ray trace node calculates the ray contributions for the cell assigned to it. Ray queues hold pending rays for each cell. The load balance problem results from the non-uniform distribution of work, which arises because ray trajectories are influenced by several object and lighting characteristics.

For many volumetric data sets, there is a large portion of voxels that are never intended to be rendered, or empty space. This occurs, for example, in photographic data sets the object of interest is suspended in a material, such as a gel, which has no relevance to the subject. In our solutions, we take advantage of empty space. At the center of this research is a framework designed to render very large scale, high-resolution volumes at interactive rates. Our dependency graph data structures enable our load balancing algorithms to efficiently use information about the relationships between data and processes.

In this chapter, we first describe several example applications of high resolution massive data sets, used to motivate and test our data management strategies, in Section 1.1. We then describe our test environment in Section 1.2. Finally, Section 1.3 gives an overview of the contributions of this dissertation.

1.1 Massive Data Applications

Some sources of increasing data sizes include large-scale physical simulations, seismic data, and the increased accuracy of CT technologies and medical imaging. Managing such massive data sets requires careful planning in all stages, from long-term storage to the moment it is cached into local memory for rendering. We rendered several very large scale, high-resolution volumes on the Stony Brook Visual Computing Cluster in order to discover the problems associated with massive data sets.

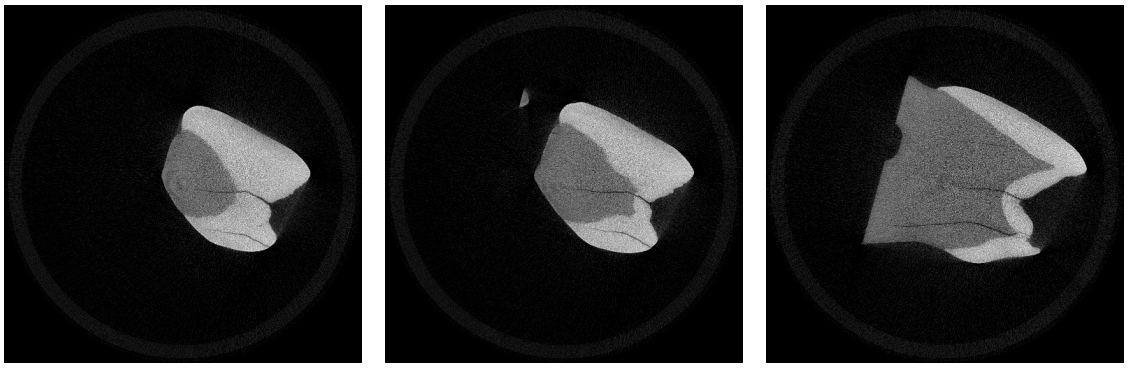
Volume data sets frequently contain large portions of empty space. We use early data

reduction to remove this empty space at the beginning of the pipeline in order to prevent using bandwidth on sending the empty volume data across a cluster. This has the additional benefit of avoiding the need to load empty volume data into local memory for each frame during rendering. In this chapter we present some example applications of massive data.

1.1.1 Anthropology Data

Data sets grow faster than memory capacity due to the improved accuracy and memory capacity of data collecting devices such as CT technologies and medical imaging. For example, High-resolution X-ray Computed Tomography (HRXCT) is a new imaging approach with a resolution in the tens of microns. The accuracy of HRXCT is similar to the destructive method of sectioning, which has been used to study fossils, teeth and bones. Studies have shown that the high resolution is necessary for the accurate reconstruction of bone in order to correctly quantify structural parameters [24].

Several HRXCT data sets have been obtained from the Stony Brook Anthropology Department for our research. The very high resolution teeth and fossil data sets consist of micro CT scanned images of 2048×2048 pixels, each with a slice thickness of around 10 microns (see Figure 1.1. The accuracy of tooth enamel measurements using micro-CT is comparable (within $3 - 5\%$) to that of destructive physical sectioning [85].



(a) Slice 115 (b) Slice 128 (c) Slice 215

Figure 1.1: High-resolution X-ray Computed Tomography Data

1.1.2 Geological Data

Visualization of geological data has the potential of saving millions of dollars in the oil exploration industry by aiding geophysicists in choosing sites for drilling. Seismic data is like a giant sonogram of the earth. A team of geophysicists, engineers and others study seismic data in conjunction with land contour maps, existing drill-hole information and other data to determine the best place to drill. For example, Tracy Stark [107] creates a *Relative Geologic Time Volume* in a preprocess step, which is used to segment layers of the seismic data during interactive rendering.

1.1.3 Medical Imaging

Photographic volumetric data sets such as the Visible Male and Visible Female data sets from the National Library of Medicine [80, 106], have become more prevalent. The size of photographic data grows substantially due to multichannel information in photographic data. The full Visible Male data set, from the Visible Human Project, is a sequence of axial anatomical images of 2048×1216 pixels at 1 mm slices. The Visible Male color data set has 1879 slices. With the addition of an alpha channel the data set is 18.7GB, more than double the capacity of our MDS System VolumePro1000 boards. The typical solution to is to volume render the data at a lower resolution. Figure 1.2 is a volume rendering of the Visible Male down-sampled to a voxel resolution of $512 \times 256 \times 883$. In order to utilize all available information in the data set, and down-sampling may result in losing critical information and resolution required for scientific and medical purposes. We do not use any down-sampling, but instead remove empty space, or areas of the data with no valid information, prior to distribution and rendering.

The Visible Korean male data from the Visible Korean Human Project [87] includes $8,590$ digitally captured photographic anatomic images of serially sectioned surfaces with photographic image resolution $2,468 \times 1,407$ and 24 bits color, for a total of 120GB. It is accompanied by a corresponding 40GB set of 8 bit color mask slice (see Figure 1.3).

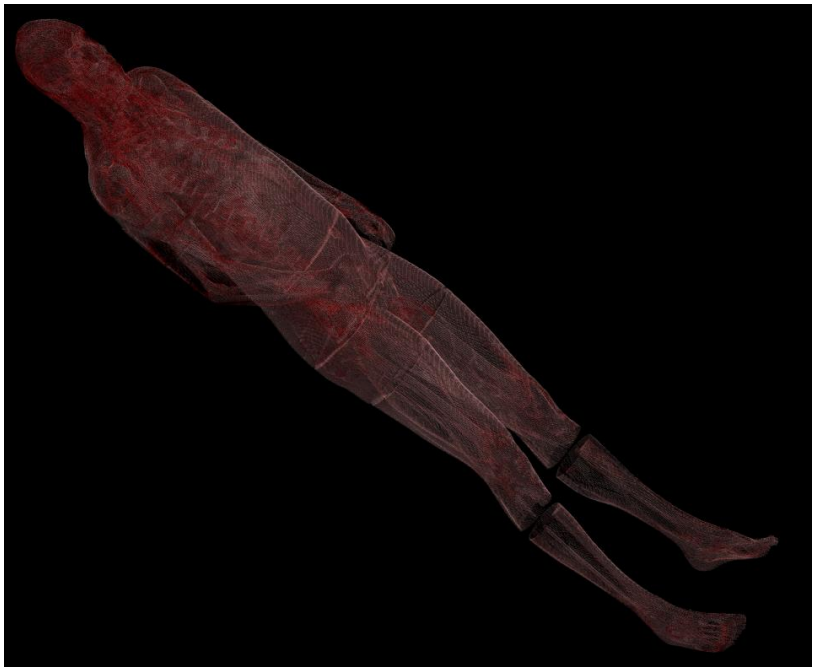
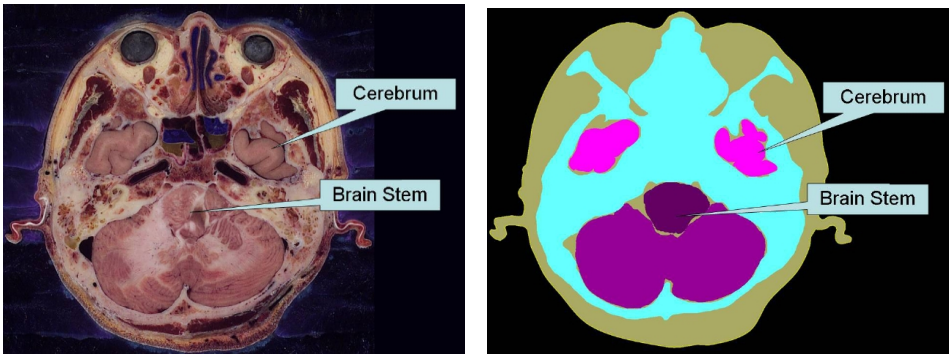


Figure 1.2: Low resolution ($512 \times 256 \times 883$) volume rendering of Visible Male data set.



(a) RGB Slice (b) Mask Slice

Figure 1.3: Visible Korean raw data.

1.2 Stony Brook Visual Computing Cluster

The Stony Brook Visual Computing Cluster, shown in Figure 1.4, is the platform used for developing our visualization framework. It is is an ever-evolving cluster environment. The GPU cluster contains 66 dual-boot compute nodes connected with a gigabit Ethernet frontend network and a 10 Gbps InfiniBand backend network, with a portion of the nodes

(a) IBM Deep Computing Visualization Cluster

(b) HP Market Development System Visualization Cluster

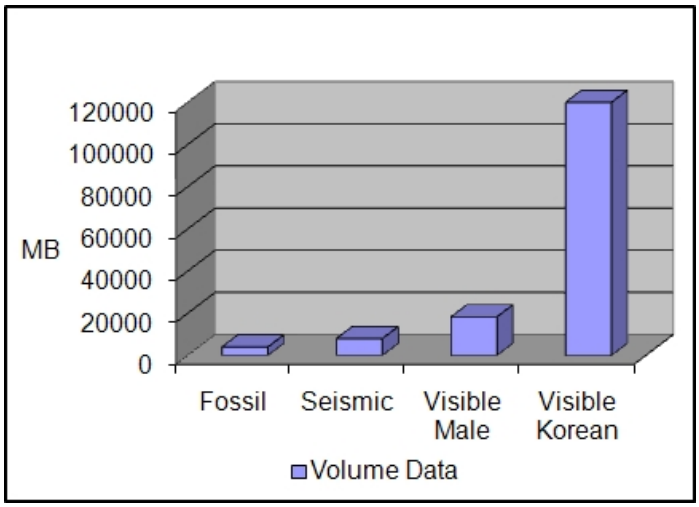
Figure 1.4: Stony Brook Visual Computing Cluster.

Table 1.1: A comparison of visualization clusters.

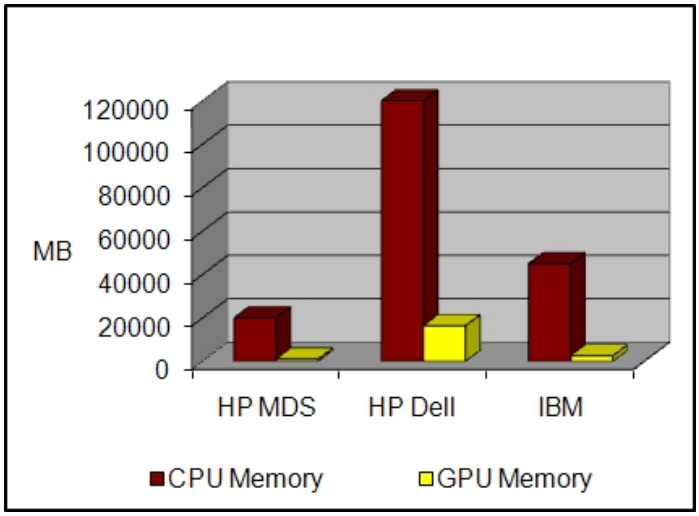
	GPU Cluster	HP MDS	IBM
Render Nodes	65	8	5
Interconnect	Infiniband	ServerNetII	Infiniband
RAM	2GB	2GB	9GB
Texture Cache	512	256	256
VolumePro Memory	1GB	1GB	n/a

connected to HP ServerNetII network through HP Sepia-2A card. Each node in the first 34 nodes contains a nVidia Geforce FX5800 Ultra graphics card, and a TeraRecon VolumePro 1000 volume rendering board. Each node of the other 32 nodes contains a nVidia Quadro FX4500 graphics card or better. Nine nodes of the Stony Brook Visual Computing Cluster are configured as a HP Market Development System (MDS) Visualization System. The IBM Deep Computing Visualization cluster contains five IBM IntelliStation Z Pro Z20 workstations with a Gigabit Ethernet frontend network and each node is connected to

Topspin 120 InfiniBand Server switch (10Gbps) through Topspin InfiniBand Host Channel Adapter. Each node contains dual-Intel EM64T 3.6GHz/800MHz CPUs, 9GB memory and a nVidia Quadro FX3400 PCI-Express graphics card. Table 1.1 is a summary of our current cluster configurations.



(a) Data sizes



(b) Memory capacity

Figure 1.5: Data sizes compared to cluster memory (in log scale).

As memory and compute power become more affordable, the size of data sets which are available has grown rapidly. Figure 1.5 shows a comparison of some the data sets rendered

in this research with the Stony Brook Visual Computing Cluster hardware. We have developed algorithms that help bridge the gap between available resources and visualization requirements.

1.3 Thesis Contributions

The techniques presented in this dissertation aim to advance research in distributed volume visualization. We have developed a volume visualization framework which is independent of the underlying physical configuration, and can take full advantage of all available hardware on a given system. This ray-directed system provides global illumination and volume rendering of massive volumes on a heterogeneous mix of hardware. Special purpose volume architectures have been developed for both ray casting and ray tracing. PC clusters have become increasingly popular for volume visualization. Recently, multi-core PCs have emerged as yet another parallel platform. Each of these options, as well as various combinations of them, offer great potential for volume exploration. At the same time, they open up new research areas as challenges are faced in process and data management. The contributions of this dissertation are listed below in increasing level of significance.

- **Out-of-core region growing and bricking**
- **Slab-projected kd-tree partitioning**
- **Dynamic programming solutions to load balanced network distribution**
- **Cell-tree for ray dependency encryption**
- **Dependency graph ray-task scheduling for ray tracing**

1.3.1 Out-of-Core Data Management

A massive volume is one which is one or more orders of magnitude larger than free memory. This includes free memory on rendering hardware or in the CPU main memory.

We introduce an out-of-core region growing algorithm [29], which is used to find a mask volume for segmentation. We introduce the *slab-projection slice* [31] as a tool for out-of-core data processing. A consecutive series of data is gathered into an orthographic projection. Our *slab-projected kd-tree partitioning* finds a kd-tree partition using non-empty voxel information represented by a series of slab-projection slices. We use out-of-core bricking to create bricks of volumetric data along with a directed acyclic graph (DAG) of the relative distance of each brick along a single view direction. The bricks are aligned with cropped subvolumes. A heuristic is used to evaluate the coherency of a series of data slices, which are parsed into a set of bricks sized for rendering hardware.

1.3.2 Load Balancing

In our framework, preprocessing is used to remove empty space from data slices and to find a good scene partition for distribution of the scene. Ray casting is typically performed by special purpose hardware or a graphics card (GPU), and ray tracing is done in software. The master node handles initial distribution of the data and subsequent scene updates. The user defines view and lighting parameters. As view information is updated, the master sends the new viewing parameters to each rendering node. Both software ray tracing and hardware ray casting are supported.

In distributed ray casting, a *brick* is a subvolume that is sized for rendering hardware. Data is distributed between nodes using a flex-block partition. A local flex-block partition is used to create and schedule bricks within a render-node. The *load balanced network distribution (LBND)* problem [31] is an optimization problem in which data is partitioned to minimize end-to-end rendering time. This time is a function of the slowest render-node, image composition time, and/or network communication time. Images are composited into a final image using an alpha blending operation, which requires the relative depth of each image. This equation is associative. If every image rendered on a node is composited into a single image prior to blending it with images from other nodes, then the bandwidth required

for transporting images between nodes is minimized. Traditional space-subdivision methods can be adapted to achieve this goal by using sufficiently large partition cells. However, load balancing suffers due to the granularity required. If empty space is not uniform within the scene, then some render-nodes are assigned large amounts of empty space, requiring little or no computation, and leaving a disproportionate amount of computation for other nodes.

Dynamic programming (DP) is a cost optimization technique. We have developed two dynamic programming solutions to the LBND problem. We attempt to minimize a cost function that reflects the end-to-end rendering cost on a cluster without violating the image composition priority constraints. The first one, *brick grouping*, starts with a directed acyclic graph of segmented and cropped bricks, and the second one, *moving walls*, applies dynamic programming to the slab-projection slices. We use our out-of-core bricking to produce a DAG of bricks for input to the first approach. We compare the load balancing using our approaches to results obtained using traditional subdivisions for several segmented regions of the Visible Korean data set.

For distributed ray tracing, we use a hybrid scheme for load balancing. The data is distributed using a flex-block partition, and ray-task scheduling is updated dynamically within the local partition of each render-node. Flex-blocks are used for cells. Space skipping within each cell is automated using the data extents of each flex-block. As rays are traced through the scene, one cell is worked on at a time. A ray queue [94] holds pending rays for each cell. When a cell is in main memory, each ray in the corresponding ray queue is traced through that cell. When a spawned ray exits the current cell, it is placed on the queue of the next cell it intersects.

1.3.3 Dynamic Programming

Our brick grouping DP algorithm for LBND [31], presented in Chapter 5, evaluates a cost function to create a load balanced network distribution. This solution is based on a DP solution for the multipass partition problem (MPP) introduced by Heirich [41]. MPP is a

problem that arises when a computation targeted for the GPU exceeds resource constraints. The computation is divided into multiple passes that do not violate resource constraints, in order to minimize total compute time.

The input to our algorithm includes a scene with volumetric data, a description of the distributed system configuration, including the number of render-nodes, rendering and local composition costs, network transfer costs, and the memory capacity of each render-node. The output is a render-node assignment, which minimizes the total runtime cost, does not violate any physical resource constraint of the system, and observes the precedence order for image composition. The depth of the search tree is limited to prevent the recursion from becoming intractable, which limits the size of the data bricks with respect to the whole data set. In addition, the partition is optimized with respect to a single viewpoint.

Our moving walls DP algorithm [30], presented in Chapter 6, uses dynamic programming find a solution to a relaxed version of the LBND problem. The output is a flex-block partition, a view-independent partition of cells which contain a combination of empty space and cropped subvolumes.

1.3.4 Dependency Graph Acceleration

In our cell-tree peeling algorithm [28], an efficient cell-processing schedule for the next frame is determined using a ray dependency graph, the cell-tree. When a render-node becomes available, the cell with the longest queue is assigned to it. As long as a cell of data is in local memory, all rays on the corresponding queue are processed before it is replaced with another cell. As with image composition of ray casting algorithms, there is a priority order relationship in ray traversal, which must be preserved in parallelized ray tracing. However, this is more difficult for parallel ray tracing. The rays travel through the scene in a pseudo-random manner. The number of times each volume block is cached depends on the order in which data cells are cached and rays processed.

Our cell-tree represents ray dependencies between cells. Each node in the cell-tree has an associated cell ID, and more than one node may have the same cell ID. The tree

is processed starting with the leaves by peeling the tree nodes with the same cell ID and adding it to the reverse schedule. An interim tree is maintained by keeping a list of nodes that have not been included in the schedule. The cell-tree peeling algorithm exploits frame-to-frame coherence by using the cell tree to find potential cache savings links. A schedule with any combination of non-conflicting link groups is a feasible one, and the optimal schedule for a single render-node includes a maximal group of cache savings links.

Cache savings links are found by gathering leaf nodes which correspond to the same cell. The cell-tree peeling algorithm performs well and has polynomial worst time. Ray tracing efficiency improves using ray dependency encryption.

1.4 Thesis Organization

This dissertation is organized as follows. In Chapter 2 we introduce the key concepts used in volume visualization, including the basic concepts of volume rendering and segmentation. We discuss applications and advantages of ray tracing and give a brief overview of the theory behind volume rendering and global illumination. We also discuss some of the algorithms that have been developed to accelerate ray traversal. We present an overview of special purpose ray casting and ray tracing hardware, along with a brief history of volume rendering on graphics hardware. We then explore some recent volume visualization cluster systems and image compositing issues. In Chapter 3 we describe our framework. We introduce the flex-block tree and cell-tree dependency graph data structures and their advantages. In Chapter 4 we present our external memory solutions, including out-of-core region growing, out-of-core bricking, the slab-projection slice and slab-projected kd-tree partitioning. In Chapter 5 we present a theoretical description of the LBND problem, along with our brick grouping DP solution. The moving walls DP algorithm is presented in Chapter 6. In Chapter 7 we present our ell-tree for ray dependency encryption, and our cell-tree peeling algorithm for ray-task scheduling. We conclude in Chapter 8 with a discussion of the impact of our work and directions for future work.

Chapter 2

Background

Volume rendering encompasses techniques that allow the visualization of 3D volumetric data. Volume data consists of information at sample locations in some space. The information may be a scalar value such as density, a vector such as color in a photographic data set, or a combination such as energy, density, and momentum in computational fluid dynamics. The space is usually three-dimensional, either consisting of three spatial dimensions or another combination of spatial and frequency dimensions.

The two predominate types of volume rendering techniques are indirect volume rendering and direct volume rendering. Indirect volume rendering involves the extraction of a surface from the input data, followed by projection of the extracted surface onto a 2D image. Internal information is not rendered because only the polygon mesh of the region's surface is maintained. This dissertation is concerned with ray-directed volume rendering techniques, which fall under the category of direct volume rendering, and so we present an overview of these in this chapter.

2.1 Direct Volume Rendering

Direct volume rendering techniques [67, 84] project the entire volume directly onto a 2D image. Object-order algorithms iterate over the volume data and determine the contribution of each voxel to the screen pixels. Splatting, introduced by Westover [116], convolves every voxel in object space with a 3D reconstruction filter and accumulates its contribution to the image plane. Three dimensional convolution is usually replaced by less computationally expensive 2D convolution filters in practice, and lookup tables are used for filter weights.

Image-order algorithms cast rays from the viewpoint through screen pixels into the volume and determine the contributions of voxels towards the pixel currently being composed. Ray casting is an image-order technique which simulates optical projections of light rays through the data set to find the pixel color. In our framework, we support ray casting and ray tracing, which both fall into the category of image order techniques.

2.1.1 Ray Casting

Direct volume rendering models the physics of the interaction of light with particles in a volume. Ray cast rendering, or ray casting uses a local illumination model and does not include effects of ray reflections or refractions. The interaction is modeled by a volume rendering integral [35].

$$I(x,r) = \int_0^L C(s)\mu(s)e^{-\int_0^s \mu(t)dt}ds \tag{2.1}$$

where $I(x,r)$ is the amount of light of wavelength λ coming from ray direction r that is received at location x on the image plane. L is the length of the ray, $\mu(s)$ is the extinction coefficient at a location s along the ray, $C(s)$ is the light of wavelength λ emitted at location s toward x. The extinction coefficient is set by the user to make an object transparent or opaque. The value of $C(s)$ is calculated using the illumination equation [27]:

$$C(s) = C_a k_a + C_l C_o(s) k_d \cdot N(s)L(s) + C_l k_s \cdot (N(s)H(s))^{ns} \tag{2.2}$$

where k_a, k_d, k_s are the ambient, diffuse, and specular material components, respectively. N is the normal vector (determined by the gradient), L is the light direction vector, H is the halfvector, and ns is the Phong exponent (see Figure 2.1). C_a specifies the ambient light color and C_l is the color of the light source. The parameter $C_o(s)$, is the color of the object. Color components are usually represented as a red, green and blue vector value. Ray cast rendering quickly creates images that look good without focussing on accuracy [38].

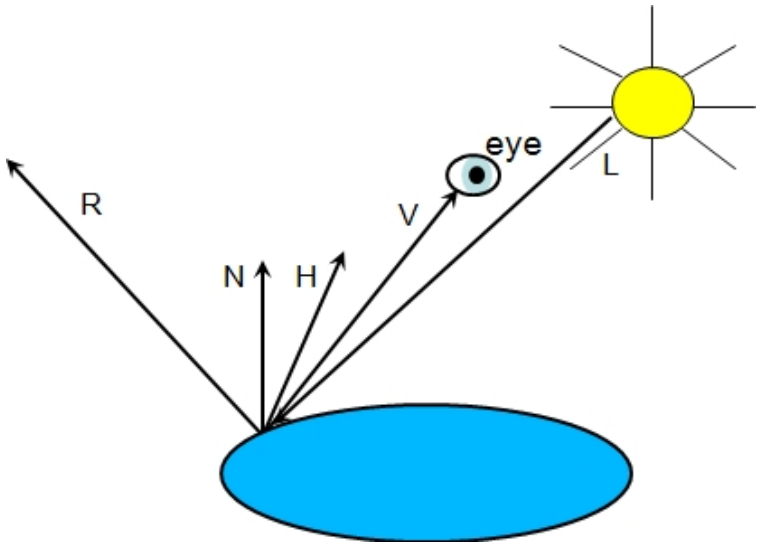


Figure 2.1: Phong shading

In ray casting techniques, the color at each pixel or partial pixel (fragment) is determined by shooting a ray through the scene and blending color and opacity of intersected objects to approximate the lighting equation. This is known as alpha blending. A ray can be described in terms of its origin *O* and direction ray *R*, where $R(t) = O + tD$. The volume rendering integral gathers the effect of the light particles along the parameter *t*. The assignment of color and opacity to a sample location in the data set is called classification. With ray casting, the user is able to selectively examine different regions of the data set by setting some of the voxel opacities to transparent or translucent values.

Various acceleration techniques are commonly used to reduce the computational cost of ray casting [70]. Some of these techniques include early ray termination and empty space skipping. Li et al. [69] achieve space skipping by partitioning volumes based on voxel attributes.

2.1.2 Segmentation

Visualizing a specific region of interest using direct volume rendering can be achieved using a mask volume. The mask is set only for voxels which belong to the region of interest. The technique of isolating a region of interest in a volumetric data set is commonly referred to as *segmentation*. Segmentation is achieved by marking the voxels that belong to the region of interest in the volume [64]. Methods for masking a region of interest have been researched for many years.

Region growing is a technique used to extract a connected region from a 3D volume based on some pre-defined connecting criterion. In the simplest form, region growing requires a *seed* point to start with. From the seed point, the region is expanded to include voxels within the target value range until the connecting criteria has been met. Some volume rendering implementations do not use the mask information directly; instead they use a transfer function indexed by the mask volume produced by segmentation. Several techniques have been developed for automating the transfer function development process and for enhancing the visual effects of a volume rendering [56].

2.1.3 Volume Rendering Architectures

Volume rendering architectures have been developed since the early 1980's. The concept of 3D scalar field *voxels* is introduced in the GODPA/Voxel, the underlying architecture of a physician's workstation [37]. This prototype uses a hierarchical pipelined hardware design. The processors independently render their subcubes by traversing them in back-to-front order and mapping each voxel onto image space using the 3D painters

algorithm [32]. Image-space shading [39] and pseudo-coloring are performed by a post-processor.

The PARCUM system [49], is based on a specially organized 3D memory called the *Memory Cube*, which allows simultaneous read/write of *Macro Volume Elements*. The memory interface of the VERVE [58] architecture uses eight different memories to hold all voxels necessary for trilinear interpolation. VIZARD [59] is a PCI-based volume-rendering accelerator that uses DMA to access the volume from main memory. A second-generation VIZARD system, VIZARDII [76] is a PCI card that performs shading and illumination calculations and uses a local look-up-table for pre-calculated gradients. Doggett et al. [22] have proposed a buffering scheme that prevents a second memory stall when a ray crosses into a new sub-cube in a skewed memory architecture. Smart Memories is an architecture which closely couples data with processing [74].

Igehy et al. [48] evaluate the effects of load imbalance on bandwidth requirements in parallel texture caching architectures. They define the working set size as the amount of memory that is being processed at a particular moment in time. Their results show that as the number of texturing units increases, the working set size for each texturing unit decreases. The point of diminishing returns for cache size is well correlated with working set size.

Cube-1 [54] performs the first opaque parallel projection of $16 \times 16 \times 16$ data sets using the Cubic Frame Buffer, 3D skewed volume memory, and a voxel multiple write bus for ray projection. The skewed memory organization allows for conflict-free access to partial beams from any major direction. Cube-2 is a full-scale VLSI-based volume visualization system based on Cube-1 technology, and Cube-3 [93] performs ray casting with various composition algorithms. The Cube-4 architecture technology [92] uses the skewed memory scheme and local communication between processors to implement the shear warp volume-rendering algorithm. Each voxel of the data set is accessed exactly once per projection. Beams of two adjacent data slices of voxels are processed simultaneously to compute a new slice of interpolated sample values in between these two slices. The *ahead,*

current, behind buffers store the samples one slice ahead and one slice behind in order to take advantage of data coherency. The samples are composited onto the base-plane, then transformed onto the viewing plane.

The Cube-4 architecture technology was developed at Stony Brook University. It has been licensed and produced by TeraRecon as the chip in the VolumePro 500 board [91], and has also been incorporated in the U-Cube ultrasound visualization system produced by Japan Radio Co. The next generation VolumePro 1000 is also commercially available, and is used as a volume rendering engine in our research.

2.2 Parallel Ray Casting

Ray directed volume rendering algorithms are well suited for parallel implementation in a distributed cluster environment. In volumetric ray casting, one or more rays emanate for each pixel and each ray accumulates the color and opacity contribution of a series of voxels along the ray in the volume data. Parallel volume rendering has been studied extensively [33, 71, 72, 77, 78, 82].

2.2.1 Visualization Clusters

Clusters of commodity hardware play an increasingly important role in visualization. A cluster can be scaled up as new hardware is introduced or as increased computing power or memory needs arise. A visualization cluster is a network of PC nodes which each have volume rendering hardware and/or one or more GPU. With the rapid development of graphics hardware and network switching technologies, several cluster-based visualization architectures that support parallel rendering algorithms have been proposed or developed [45, 83, 108]. The advantage of such systems is that they are easy to build from commodity components. Various methods have been proposed for splitting a scene among processing units with graphic models [13, 25]. Samanta et al. [102] have introduced a view dependent, hybrid 2D image and 3D polygon data partitioning scheme.

2.2.2 Image Compositing

Distributed ray cast rendering involves placing an image plane at the face closest to the camera in the major axis direction, for each scene cell and taking a snapshot of that cell. Image compositing is used to blend these snapshots together. Associative image composition allows any geometrically coherent group of sub-volumes to be rendered independently then blended together. The relative distance to the viewpoint is needed for alpha blending, and distance (priority) order must be respected. This result allows us to treat distributed ray casting as an extension of classic ray casting. If we place bounding boxes over our subvolumes, the operation is equivalent to ray casting in a geometric scene, with geometric surfaces being replaced with images from subvolumes.

Lombeyda et al. [71] have shown the arithmetic equivalence of a single ray casting composite computation and the combined result of a set of (smaller) composite computations. They propose the following concurrent composition operator, F, for alpha blending subimages. They demonstrate that this operator is associative, but not communicative and that it yields a parallel composition result, which is arithmetically equivalent to a serial composition of sequential images using the ray casting alpha blending function:

$$F(i_1, i_2) = (C_{i_1} + (1 - \alpha_{i_1})C_{i_2}, \alpha_{i_1} + (1 - \alpha_{i_1})\alpha_{i_2}) \tag{2.3}$$

where i_1 and i_2 are subimages, and C_s is the accumulated color, and α_s is the accumulated opacity of a subimage.

Figure 2.2 illustrates parallel ray casting. When we composite images rendered by different render-nodes using ray casting, it is equivalent to tracing rays from one node into another in the same manner described in Section 2.1.1. The rendering time for a given image size is roughly proportionate to the size of volume rendered.

Associative image composition implies that any geometrically coherent group of sub-volumes may be rendered together as long as the overall priority order is respected. The problem becomes analogous to the ray traversal problem through a triangle mesh.

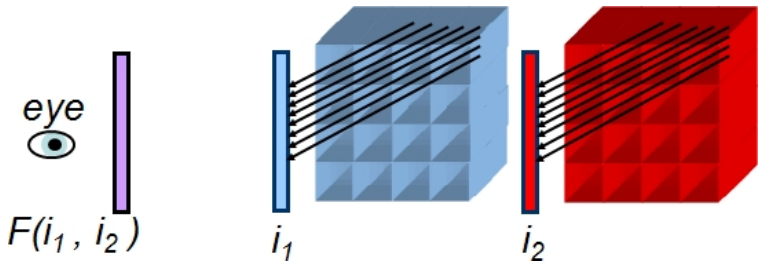


Figure 2.2: Parallel ray casting. Partial images are generated for each subvolume, and composited using a concurrent blending operator F (see Equation 2.2.2).

Software compositing has been used to composite geometric primitives distributed and rendered on different renderers [78]. Compositing can also be performed on a GPU [33]. In the VG-Cluster system [82], a special image compositing hardware composites images from eight PCs. These interim images are composited to produce the final image. In the HP MDS Visualization cluster [77], a series of images are composited for each frame by a special purpose board, HP Sepia-2a, located on each render-node. The Sepia-2a composites a local image with one received through DVI acquisition from another node. Image composition priority order is the relative distance along the view direction. This can be obtained from our flex-block tree.

The Chromium system [46] uses a stream-processing framework for interactive rendering on clusters. SIMD arrays provide the power of an ASIC implementation with the flexibility of a dynamic solution on a general-purpose machine. Special purpose hardware has been used for compositing as well. In the VG-Cluster system [82] a special image compositing hardware composites the images from eight PCs. The VG Cluster uses a tree structure to direct composition.

2.2.3 Hardware Compositing

In the HP Market Development System (MDS) Visualization cluster [42], a series of images are composited for each frame by composite hardware, HP Sepia-2a, located on each render-node. The Sepia-2a [77] composites a local image with one received through

DVI acquisition from another node. The Sepia-2a architecture is a special purpose compositing hardware which combines the images produced from individual data subsets and supports user mouse and keyboard interactive rendering. The Sepia board gets its graphical image data through the DVI display port of a commodity OpenGL graphics card when configured as a render-node. The Sepia board connects to a ServerNet II high-bandwidth, low latency network.

The Sepia-2a daughter board allows DVI output to the display board further increasing speed-up; faster image acquisition for alpha blending is enabled by avoiding the frame buffer read-back bottleneck. Composition order is not static and compositing operations are not required to be communicative. The depth information required by alpha blending is provided by the CPU. It can change from frame to frame as the camera position moves. The volume correction matrix is used to place each subvolume in its correct position within the whole volume. Compositing of images between nodes is executed either on a GPU [33], or on special purpose compositing hardware compositing.

2.2.4 Scene Partitioning

Optimized octrees have been successfully used for multipass rendering task scheduling [60]. Optimized octree-based rendering techniques that skip empty spaces directly on the GPU can be used for local GPU rendering pass scheduling in conjunction with our methods. However, the goal of the load balanced network distribution problem is different in that the cost of moving data between nodes and inter-node image composition is more significant. The smallest resolution cell is sized to fit the target rendering hardware because data bricks are available in local disc memory. In contrast, for network distribution, only a subset of bricks is available to each node, and the partition generally should minimize inter-node communication. If a static partition is used, the granularity required to force all local images to have consecutive depth priorities is typically orders of magnitude larger than multipass partitioning cells. As a result, the rendering assignments are not evenly allocated among resources if the empty space distribution is not uniform.

In parallel volume rendering the scene is typically subdivided using an acceleration structure such as a grid, octree or kd-tree. Lombeyda et al. [71] have shown the arithmetic equivalence of a single ray casting composite computation and the combined result of a set of (smaller) composite computations. They demonstrate that the alpha compositing operator is associative, which means portions of the scene can be rendered individually and composited together as long as the overall priority order, or relative distance to the viewpoint, is respected. In the VG-Cluster system [82], special purpose hardware composites images from eight PCs. These interim images are composited to produce the final image. In the HP MDS Visualization cluster, a series of images are composited for each frame by composite hardware, HP Sepia-2a, located on each render-node. The Sepia-2a [77] composites a local image with one received through DVI acquisition from another node. Compositing is also performed on a GPU [33].

There are three main approaches to parallelization used in volume rendering: demand-driven, data parallel and hybrid. Demand driven ray-parallel techniques divide the screen into a number of regions where each region represents a task, a number of processors execute these tasks and whenever a task is completed the processor requests a new one from the master. With ray-parallel techniques, all voxels along a ray are processed simultaneously. Data parallel approaches partition the object space. The *ray-slice-sweeping* algorithm [10], is a slice-parallel technique, which processes consecutive data slices that are parallel to a face of the volume data set. Sectioning has been introduced by de Boer et al. [20]. The volume is divided into horizontal sections, which are each processed in turn. This reduces the slice face area and hence the size of the slice buffers. PAVLOV [61] is a two-dimensional array of SIMD processing elements used for parallel segmentation. The volume is distributed so that a complete beam of voxels along the z-axis is stored on each processing element to allow conflict-free access to any z-slice so there is only a single clock cycle delay between slices.

If the data is partitioned so that the priority order is maintained with respect to render-node assignments, then each render-node contributes exactly one image to the final image.

In this case, the end-to-end rendering time is restricted by the slowest render-node. For distributed ray casting, the scene must be partitioned between nodes for good load balancing, and a strict view dependent priority order is required for image composition. Kd-tree, octree and grid partitions all meet the criteria of producing a deterministic priority order for any given viewpoint. However, these schemes do not achieve good load-balancing for data sets with large variations in sparsity because partitioning is not guided by the distribution of empty space throughout the data. The problem of partitioning volumetric data for distribution across a cluster to achieve good load balancing for parallel ray casting is the *load balanced network distribution (LBND)* problem. The goal of the LBND is to minimize end-to-end render time in a distributed rendering system within resource and priority order constraints. The input is a volumetric data set, along with render and network cost information. The precedence order is defined as the relative distance of each voxel with respect to a given view direction.

2.3 Volume Rendering on Graphics Accelerators

Volume rendering on graphics cards has become increasingly popular [3, 11, 19, 23, 57, 62, 75, 78, 119]. The functionality at the core of a graphics processor (GPU) uses single instruction multiple data (SIMD) techniques with image-space partitioning. Researchers capture the relatively cheap compute power of GPUs, using various techniques which leverage GPU optimizations originally designed for rasterization. Volume data is stored as 2D or 3D texture for GPU rendering. A ring network of 20 graphics processors and eight renderers is used in the Pixel-Planes system [119]. The ring network is bandwidth limited and does not scale well. Akeley [3] propose storing the volume as a solid texture on the graphics hardware, and then to sample the texture using planes parallel to the image plane and composite them into a frame buffer using blending hardware. This method quickly produces unshaded images, but with two main drawbacks: the volume must be re-shaded and re-loaded every time any of the viewing parameters changed, and non-linear transfer

functions are not interpolated correctly by the texture hardware.

Cabral et al. [11] use texture mapping in combination with an accumulation buffer on four Raster Manager Reality Engine Onyx machines to implement the filtered back projection CT algorithm. PixelFlow [78] uses image composition to composite geometric primitives distributed and rendered on different renderers. This approach has high network bandwidth requirements for composition since every pixel is transferred multiple times for every frame. Dachille and Kaufman [19] use a shear-warp [63] style method where the texture mapping hardware performs the shearing and perspective scaling. In recent years GPUs have been enhanced with features such as programmability and texture access, and have become increasingly useful for general purpose programming [75]. Algorithmic speed-ups are implemented using GPU features. For example, early ray termination takes advantage of the z-buffer [62].

2.3.1 Multipass Partitioning

Shader programs allow complex problems to be solved on a GPU as long as the underlying problem can be reformulated to fit the SIMD paradigm. However, most complex applications, including texture-mapped volume rendering, exceed the resource capacity of even the most up-to-date GPU hardware. The solution is to partition the problem into multiple passes. Our brick grouping DP solution to the load balancing, described in Chapter 5, is based on a dynamic programming (DP) solution to the *multipass partitioning problem (MPP)*, or the DPMPP solution, which is briefly described here.

The input to the MPP problem is a valid shader program in the form of a DAG. The costs of each operation and GPU pass are included as well. The output is a schedule of DAG operations partitioned into passes which minimize the total runtime cost, where the schedule observes the precedence relations of the DAG, and no pass exceeds the physical resource constraints of the GPU. Several solutions to the MPP problem [14, 41, 100] have been proposed. These algorithms each evaluate the cost function of a proposed GPU pass

by generating the code for that pass. Chan et al. [14] have introduced a minimum-cut approach, the recursive dominator split (RDS) which optimizes to minimize the total number of passes. RDS computes a limited search of a subregion of the solution space, and chooses the solution with least cost. The runtime is $O(n^3)$. This time is reduced to $O(n^2)$ in RDS$_h$, which uses a heuristic to replace the subregion solution search with a tradeoff in the partition quality. These algorithms allow multiply-referenced nodes to be either recomputed or saved. RDS considers only a single connected region of its input DAG at any time. Riffel et al. [100] have observed that the MPP problem is an instance of the job-shop scheduling problem. They introduce the *MIO* algorithm, which is a greedy algorithm based on list scheduling. MIO gives an approximate solution with average time $O(n \log n)$ which is optimal locally but may be suboptimal globally.

2.3.2 DPMPP

The algorithm proceeds backwards from the last stage to the first. Each stage evaluation is initialized with a set *T* of transitions, where transition t consists of $t.precondition$, $t.postcondition$, $t.operation$, and $t.cost$. For each transition, *t*, *t.postcondition* is the machine state in *t.precondition* following application of *t.operation*. For each stage, the principle of optimal substructure is used to determine the transitions which may be optimal. These are used as input to the next subsequent (earlier) stage. The resulting solution is globally optimal because it observes the principle of optimality [8], which states that the optimal solution for the current stage is optimal regardless of what policies or conditions led to this stage. The time-complexity of the DPMPP algorithm could potentially be intractable. However, an average time of $O(n^{1.14966})$ is achieved experimentally.

2.4 Ray Tracing

Volumetric ray tracing methods include global illumination and higher-order light-material interactions [36], which are excluded from the ray cast rendering. In ray tracing,

the same rays generated for ray casting (eye rays) are cast. In addition, rays are spawned at ray-object intersection points to approximate reflected, refracted and shadow rays. A common element in the ray casting and ray tracing techniques is that the contribution of a data intersection point is dependent on the accumulated opacity of the ray at that point from prior intersection points along the ray path.

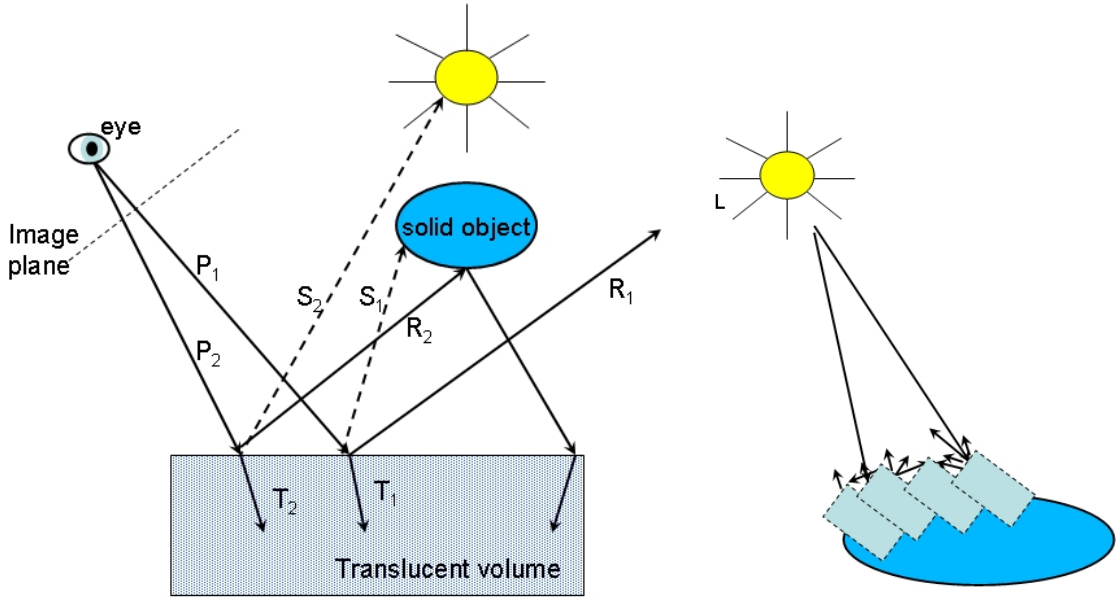
Ray tracing methods are used for multiple purposes including rendering, intersection testing, and global illumination with bidirectional reflectance distribution function (BRDF) splatting. Ray tracing is a unifying method for rendering volume data, geometric primitives, parametric Bezier or NURBS patches, implicit surfaces, point clouds, and more. It is based on global illumination models of classic physics [35], and so it results in more physically correct images, and is extensible to immersive environments [2]. The ray traced rendering and the BRDF splatting algorithms both simulate the interaction of light with the environment, either in a forward (splatting) or backward (gather) direction. The main disadvantage of ray tracing is that it requires out of order memory access and high computing power.

2.4.1 Rendering Model

Most light particles will have no effect on the final image. It is more efficient to determine the effect of light at each point (pixel or sub-pixel) on the image plane because it doesn't require simulating unseen light particles. The property of reciprocity states that the path a light source takes from a light to the image plane is reversible. Ray traced rendering takes advantage of this property and traces the paths of particles, or light rays, which actually do intersect the image plane.

An image is formed by finding an approximate solution to the rendering equation for each pixel. Classic, or Whitted-style, ray tracing [117] captures shadows, reflection, refraction and diffuse surface shading. The ray tracing model places an image plane in the scene and captures the light arriving the image plane, see Figure 2.3(a). One or more rays are sent through each pixel of an image plane and each ray gathers lighting information as

it hits objects in the scene. As the ray interacts with the environment, additional rays are spawned as it is reflected and refracted through objects. Shadow rays are shot from each intersection point toward each light source to indicate whether that point is blocked from the light source.



(a) Classic ray tracing (b) BRDF

Figure 2.3: Ray tracing physics. (a) Rays P_1 and P_2 are sent through the scene, spawning reflection rays R_1 and R_2, refraction rays T_1 and T_2 and shadow rays S_1 and S_2. Spawned reflection and refraction rays are recursively traced; (b) Bidirectional reflectance distribution function (BRDF). When light hits a surface, it is scattered and bounces off many micro-facets. This process is approximated with the BRDF.

2.4.2 Global Illumination

The basic principal behind most global illumination algorithms is the physics of the interaction of light with particles in a volume. In the physics model, light particles, called photons, are emitted from one or more light sources and are scattered through the environment. Photons are either absorbed or reflected by the objects they intersect. When light hits a surface, it is scattered and bounces off many micro-facets as shown in Figure 2.3(b).

Global illumination models these light particles, and is used to model indirect lighting effects, such as indirect illumination are color bleeding and caustics. These occur when light bounces off of one surface to illuminate another.

Much research has been done for the generation of bidirectional reflectance distribution function (BRDF) [6]. With the instant radiosity algorithm [55], a scene illuminated by a small number of virtual point lights. The irradiance at any given surface point is approximated by the sum of the contributions from all power of each point light. Photon mapping is a related technique which splats energy around a scene [50, 51].

2.4.3 Ray Traversal Acceleration

For decades research has been done on acceleration of ray tracing. Several techniques reduce the number of rays to be traced. They include early ray termination, using fewer samples on the image plane and reducing the number of rays to be traced for each sample. For early ray termination a ray recursion is terminated once its pixel contribution drops below a certain threshold. However, simply terminating such rays results in biased images and an overall decrease in illumination. Russian roulette termination [5], help to remove this bias at the cost of adding noise to the image. The main drawback of these methods is that high-frequency details tend to get lost. Progressive ray tracing is a technique for producing lower quality images which are then improved with time [15].

2.4.4 Parallel Ray Tracing

Image space partitioning techniques divide the screen space (rays or pixels). Ray traced rendering is easily parallelized on scenes in which all data can be replicated and accessed by all compute entities, or ray tracers. Yagel et al. [97] propose a discrete ray tracing method that leads to fast implementations because all ray-traversal calculations can be performed in parallel. More often than not, however, data sets are too large for this to be feasible, and BRDF splatting requires unique scene access for each ray tracer. Therefore, there has

been much research on distributed ray tracing. For clarification, the original use of the term *distributed ray tracing* was for rays generated with a distribution function [17]. This is generally referred to as *distribution ray tracing* to avoid ambiguities, and is not discussed here.

Object subdivision techniques are data-driven as with the bounding volume hierarchy [79, 101]. A bounding volume is a simple geometric primitive such as a box or sphere that can be intersected very quickly to avoid costly ray-primitive intersection computations for more complex objects. A B-kd-tree [118] is a hybrid approach which uses a kd-tree partition with bounding volumes as leaf nodes. This approach is similar in concept to our flex-block tree, except that our approach is designed for massive volumes, and the total size of data distributed in our system is a function of the flex-block partition.

Space partitioning techniques organizes the scene into a set of non-overlapping cells, and divide the scene among processing units [13, 25]. Each cell contains a list of references to all the primitives that fall partially or completely within the cell bounding box. Ma et al. [73] perform space subdivision in parallel. Space subdivision techniques include binary space partitioning (BSP) trees [9], octrees [34], kd-trees, and regular grids. Acceleration data structures are also used to mark empty space for skipping during ray casting and ray tracing [16, 68, 69, 90, 112].

Efficient ray traversal is a well studied problem [4, 66, 90, 94].Figure 2.4 illustrates the pseudo-random ray dependencies that result when cells are revisited by spawned rays.

Shadow rays for a point light source that are occluded by the same object are inherently coherent. This coherence is exploited by the shadow caching [40] technique. Fernandez et al. [104] subdivide the scene, and keep a list of light sources that will be fully occluded, fully visible, or partially visible to each region of space. They use an approximation for these, based on shadow rays, so that data structure can be built at run time for dynamic scenes. Recently much attention has been focused on efficient building of kd-trees [47].

Pharr et al. [94] introduce the use of ray queues associated with each portion of the scene volume, or cell. The scene is divided into an acceleration or scheduling grid as

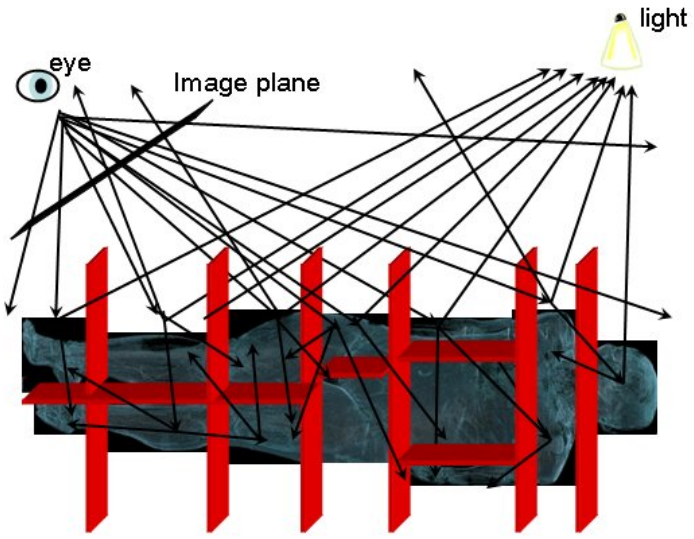


Figure 2.4: Pseudo-random ray traversal. Arrows show possible ray traversal paths. Each path may be followed by thousands of rays.

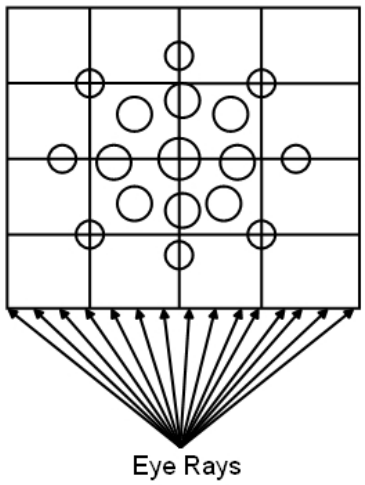


Figure 2.5: Hybrid object image ray tracing.

illustrated in Figure 2.5. When the queue for the current cell is emptied, another cell is chosen for processing. A greedy *max-work* approach is used to choose the next cell. The difficulty with optimizing ray traversal for ray tracing stems from the fact that the same cell may be revisited by a ray and by its descendants any number of times, and these descendants may not be generated until it has been processed within another cell.

Reinhard et al. [98, 99] use a pyramid clipping scheduling scheme and the demand-driven scheduling of shadow ray tracing. Each processor handles both types of tasks, but data parallel tasks are given higher priority. The geometry is distributed and the spatial subdivision structure is replicated. A pyramid is constructed around a bundle of rays and intersected with an octree subdivision of the volume. Clusters of rays are gathered to improve coherency [90].

2.4.5 Ray Tracing Systems

Sobierajski and Kaufman have developed a complete framework for volumetric ray tracing that includes global illumination effects such as shadows, reflections of the scene in mirrors, light interaction between multiple volumetric and geometric objects, and volumetric rendering effects such as fog and transparency [105]. The results have been incorporated into VolVis, a comprehensive volume visualization system developed at Stony Brook University. GI-Cube [18] is a single PCI board volume ray tracing coprocessor with parallelization based on cubic cells. Interleaved cubic cells are used to improve load balancing. Ray tracing in our framework, described in Chapter 7, is based on the GI-Cube simulation.

The SaarCOR Realtime Ray Tracing Engine [103, 111, 113, 114, 115] is a hardware developed for ray tracing triangles. The Galileo system uses kd-tree space subdivision for ray tracing [88, 89]. It has been parallelized on a CPU cluster system [21]. Recently, ray tracing has been implemented on graphics hardware [95, 96]. Vertex tracing [110] targets primary rays directly towards the vertices of visible triangles, computes the color of these vertices by recursive ray tracing, and uses graphics hardware to perform the interpolation between the vertices.

2.5 Summary

Much research has been done to improve algorithms for high quality rendering and interactive visualization. Volume rendering hardware has been developed to speed up this

process. The market has driven the production of low-cost, high compute power GPUs and switching technologies. The result is an ever-changing environment of visualization clusters, which need resource management which is not tightly coupled to the specifics of any particular configuration. This is the environment which has driven our research.

Chapter 3

Framework for Interactive Massive Volume Visualization

In this chapter, we present a framework which automates load balanced volume distribution and ray-task scheduling for parallel visualization of massive volumes. The main bottlenecks in distributed volume rendering involve moving data across the network and loading memory into rendering hardware. Our load balancing solution combines static network distribution with dynamic ray-task scheduling. At the core of the dependency graph approach are the flex-block tree and the cell-tree.

We first present the system functionality that our framework addresses in Section 3.1. In Section 3.2 we describe our data management philosophy and argue the case that early data reduction is an important aspect of managing massive volume data sets. We then introduce the dependency graph data structures that are the basic elements of our framework in Section 3.2.2. In Section 3.3 we introduce an overview of our data distribution and dynamic ray-task load balancing. In Section 3.5 we describe our test environment, and in Section 3.6 we show some interactive rendering results.

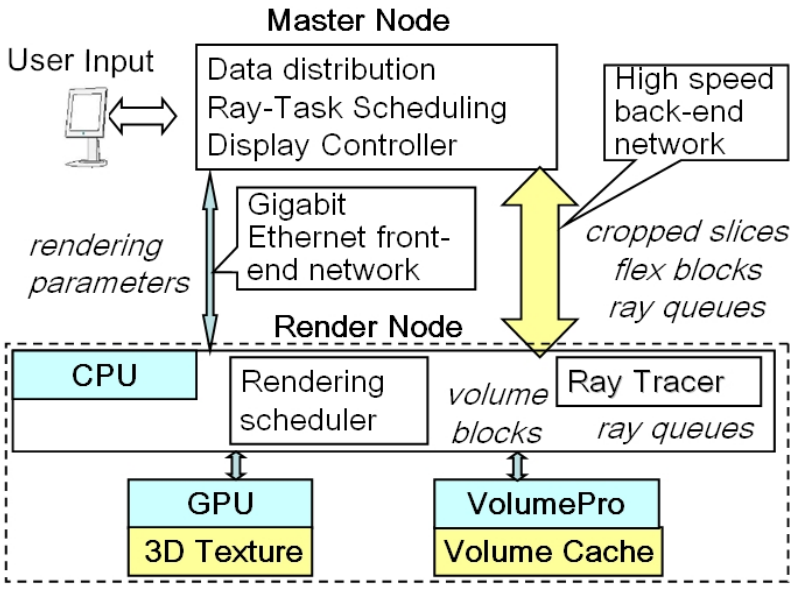


Figure 3.1: Block diagram of our visualization framework.

3.1 System Functionality

An overview of our framework is shown in Figure 3.1. The master node handles initial distribution of the data and subsequent scene updates. As the user updates view information, the master sends updated viewing parameters to each rendering node. Our framework improves load balancing for both ray casting and ray tracing. Ray cast rendering is performed by hardware and ray tracing is performed in the CPU.

Each node renders the image of its portion of the scene for the current frame. Wherever these partial images overlap in screen space they must be composited. This requires the bandwidth-intensive task of transporting images between compute nodes. By restricting scene partitions to those which yield an unambiguous priority rule for any direction, all compositing occurs between neighboring nodes, which significantly reduces overall network communication compared with global image communication. The relative depth of each image is an input to the alpha blending equation used for compositing.

Ray casting is typically performed by special purpose hardware or a graphics card (GPU), and ray tracing is done in software. The master node handles initial distribution

of the data and subsequent scene updates. The user defines view and lighting parameters. As view information is updated, the master sends the new viewing parameters to each rendering node. Both software ray tracing and hardware ray casting are supported.

3.2 Data Management

Our data management pipeline is shown in Figure 3.2. The preprocessing consists of the slice preprocessing phase, followed by the data distribution process. Slice preprocessing is used for creating cropped slices to be distributed across the network for rendering. Each raw data slice is read once during this phase. Non-empty regions are segmented using a mask volume, cropped and written to disc memory. Out-of-core region growing is used if no mask has been provided with the data set. Concurrently, data extent information is consolidated into a series of slab-projection slices. Partitioning uses these slab-projection slices to avoid any further raw data reads. Interactive rendering is done in parallel as viewing parameters are parsed and distributed by the master node. Render pass scheduling, hardware ray casting and compositing all take place on the render-nodes.

In distributed ray casting, a *brick* is a subvolume that is sized for rendering hardware. Data is distributed between nodes using a flex-block partition. A local flex-block partition is used to create and schedule bricks within a render-node. End-to-end rendering time is restricted by a combination of the slowest render-node, image composition time, and/or network communication time. Images are composited into a final image using an alpha blending operation, which requires the relative depth of each image. This equation is associative. End-to-end render time increases with the maximum number of rendering passes required by any node in the cluster. There are two ways to keep the maximum number of rendering passes down: by assigning an equal number of rendering passes to each node, and by decreasing the total amount of memory assigned. If every image rendered on a node is composited into a single image prior to blending it with images from other nodes, then the bandwidth required for transporting images between nodes is minimized.

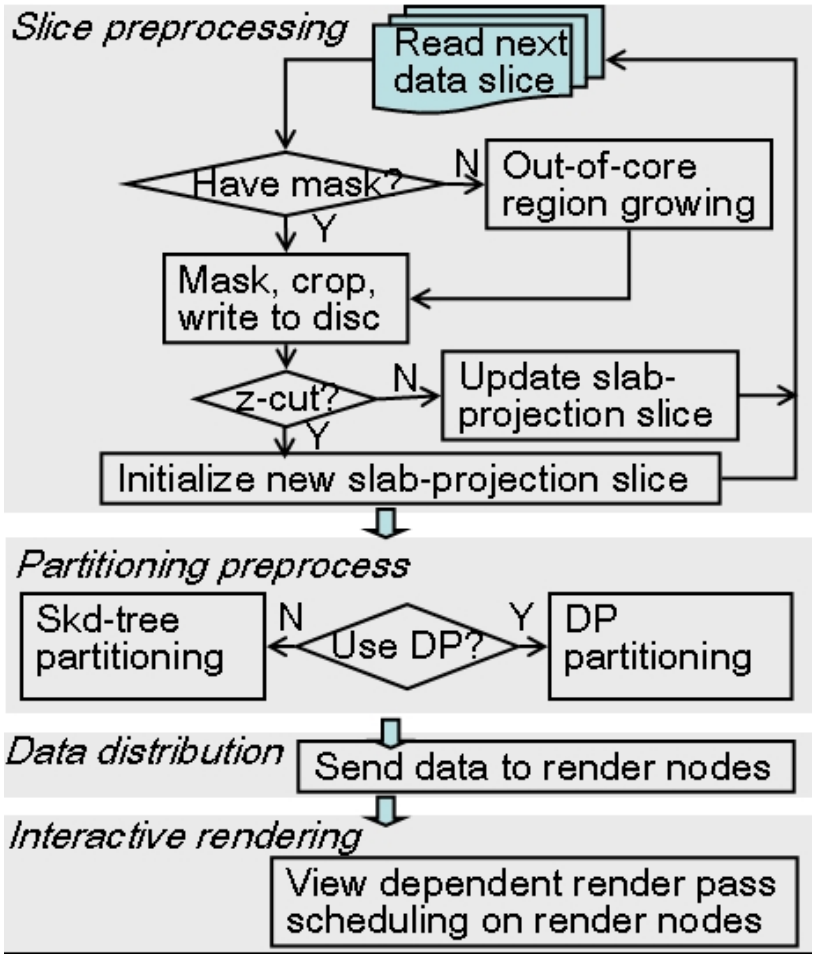


Figure 3.2: Distributed rendering data management pipeline.

Traditional space-subdivision methods can be used to reduce total memory assigned, but data distribution requires partition cells to be sufficiently large so that each render-node contributes a single composited image per frame. However, load balancing suffers due to the granularity required. If empty space is not uniform within the scene, then some render-nodes are assigned large amounts of empty space, requiring little or no computation, leaving a disproportionate amount of computation for other nodes.

Intra-node compositing takes place on the rendering engine. Multi-pass rendering is used whether rendering is performed on volume rendering hardware or GPUs. If rendering takes place on the VolumePro 1000 board, we leverage the capability of rendering multiple volumes in a single context with on-board image compositing.

For distributed ray tracing, we use a hybrid scheme for load balancing. The data is distributed using a flex-block partition, and ray-task scheduling is updated dynamically within the local partition of each render-node. The problem of determining the optimum order in which cells are cached is NP-Complete, so several optimizations have been proposed [18, 90, 94, 98, 99]. A ray queue [94] holds pending rays for each cell. Space skipping within each cell is automated by using flex-blocks for cells by advancing the ray forward according to the data extents of each flex-block. As rays are traced through the scene, one cell is worked on at a time. When a cell is in main memory, each ray in the corresponding ray queue is traced through that cell. When a spawned ray exits the current cell, it is placed on the queue of the next cell it intersects. It is not generally possible to create a schedule which fetches each cell only once for ray tracing algorithms.

3.2.1 Early Data Reduction

A volume is defined as massive when it is one or more orders of magnitude larger than the size of main memory available in a single visualization engine. Massive data requires either out-of-core (external memory) or distributed preprocessing. We propose external memory solutions, which attempt to minimize the number of times a piece of data is read from disc memory. For example, we use the bounding boxes of bricks, rather than physical data as an external memory approach to our dynamic programming algorithm. Bounding boxes are used to create volume texture bricks, sized to fit the rendering hardware, from pre-cropped slices at render time.

Space leaping is an important optimization used for ray directed techniques. However, runtime space leaping does not alleviate the inefficiency of sending empty volume data across a cluster, or bringing this data into texture or cache memory. We use preprocessing to remove empty space from data slices and to find a good scene partition for distribution of the scene. The problem of running out of memory occurs in all parts of the pipeline, from the initial reading of raw data slices to moving data across the network. Frequently volumetric data contains a large portion of empty space, or data with no useful information.

We can reduce the size of the distributed data by modifying the input volume such that it only contains the voxels which belong to one or more regions of interest.

Large-scale data management requires an out-of-core approach because all data will not fit into local memory. Preprocessing large-scale data for distribution must be handled without the caching all data at once. We use out-of-core segmentation to determine the bounding boxes and dependency graphs of volume boundaries for automated volume splitting. If segmentation is provided, as with the Visible Korean data set, then we read one slice of raw data along with the corresponding segmented slice. Otherwise, a small slab of raw slices is read at a time and out-of-core region growing is used to determine the segmented area to crop. The slices are processed in slabs sized to fit the target rendering hardware texture or cache size. We reduce data sizes to achieve space skipping prior to texturing by cropping the empty or unused space. Out-of-core segmentation is used to determine the bounding boxes and dependency graphs of volume boundaries for automated volume splitting.

If segmentation mask data is provided, then we read one slice of raw data along with the corresponding segmented slice. Otherwise, a small slab of raw slices is read at a time and out-of-core region growing [29] is used to determine the segmented area to crop. Slice data is kept in cache memory only long enough to do slice-to-slice comparisons for detection of region overlaps, splits and merges. The most recent slice is used to seed the region-growing process on the current slab. Each slice is cropped to the bounding rectangle of region of interest.The goal of segmentation here is to segment out empty space, as indicated by the user; any or all valid regions in the data are included.

3.2.2 Dependency Graph Data Structures

We use dependency graph information for both ray tracing and ray casting. The flex-block partition is used for data distribution. The corresponding flex-block tree contains the image order dependencies for ray casting. These dependencies are also used to accelerate ray traversal in ray tracing by indicating potential neighbors a ray may enter. The cell-tree

is a compact representation of ray dependencies between cells, where a cell is a flex-block. It is several orders of magnitude smaller than the number of dependencies represented. The cell-tree is used for ray-task scheduling and for ray traversal speedup in ray tracing.

3.2.3 Flex-block Partition

Traditional space subdivision methods do not address the load balancing problem for distributed rendering. Our flex-block partition is designed for this purpose. It is derived from a kd-tree partition, but the leaf nodes are flex-blocks. In addition, the flex-block partition does not restrict the order of partition cuts to alternate between axis. This is particularly important for a scene with disproportionate lengths on each side.

We introduce the *flex-block tree*, our dependency graph data structure, derived from a kd-tree, which describes a *flex-block partition*. The flex-block partition is used to define a render-node assignment. The flex-block is the data structure used to define a flex-block partition and is illustrated in Figure 3.3. Each cell in the partition is a flex-block. The flex-block tree node, shown in Figure 3.3(a), contains cut plane and child node information as in a traditional binary tree. The leaf nodes in the tree are flex-blocks. A flex-block is a cell containing a tightly cropped subvolume, which does not necessarily occupy the whole cell (see Figure 3.3(b)). Prior to distributing data for rendering, we reduce the overall size of the data. We crop empty space by aligning a tight axis-aligned bounding box around a cohesive non-empty area of the volume.

Partition walls are moved to simultaneously accommodate non-empty subvolume boundaries and render-nodes for good load balancing. A *flex-block* is a cell containing a tightly cropped subvolume, which does not necessarily occupy the whole cell. Empty space skipping is generally treated as a render time solution. However, reducing the size of data early in the pipeline has the advantages of reducing the run-time of later preprocessing and processing steps as well as data movement across the network. The flex-block partition is designed to reduce empty space, facilitate early data reduction, and to provide a deterministic depth order of flex-blocks for any given view direction. The flex-block tree

(a) Tree Node (b) Flex-block

Figure 3.3: Flex-block tree data structure captures view-dependency order and data position information. Leaf nodes are flex-blocks. (a) Flex-block tree node; (b) Flex-block data structure.

is similar to a kd-tree except that leaf nodes are cells containing a combination of empty space and tightly cropped subvolumes, or *flex-blocks*.

3.3 Load Balancing Data Distribution

Static data distribution is used for load balancing both for ray casting and ray tracing. Dynamic task scheduling is used to schedule subvolumes, or flex-bricks, within each render-node. For ray casting, this involves traversing the flex-block tree structure to determine the order for compositing (and rendering) images. For ray tracing, this involves scheduling flex-bricks for pending rays.

Our data management preprocessing pipeline for massive volumes proceeds as follows. One raw data slice is read at a time and out-of-core region growing creates a mask for each slice. Region growing is is skipped if a mask volume is is provided with the raw data, as with the Visible Korean data set. Slices are cropped to cropped to one or more bounding rectangles of regions of interest using the mask volume. Cropped slices are stored on disk memory for distribution to render-nodes.

The bounding rectangle parameters of cropped slices are used as input to our moving walls algorithm, which partitions the scene with a kd-tree space partition. Dynamic programming creates flex-blocks within a kd-tree like partition of the scene. The kd-tree is extracted from the output of the moving walls procedure. The image priority order for each

of eight quantized view directions is extracted from the kd-tree and stored in a look-up table. This table represents a dependency graph of flex-blocks, which is used for image composition and ray-task scheduling. Flex-block parameters, including the flex-block dependency graph, are distributed across the cluster along with cropped slices. Volume blocks are created locally by render-nodes. Alternatively, at the request of the user, volume blocks are created during preprocessing for distribution.

The user has the option of creating flex-block volumes to be distributed, or to distribute cropped slices along with flex-block bounding boxes. For a stable configuration volume blocks are stored locally on the render-nodes. However, it is generally better to store cropped slices rather than volumes which have been targeted to a specific system. If the configuration changes, the data distribution algorithm is run using the new constraints, and cropped slices need not be regenerated. Blocks are never created until the end of pre-processing because memory used by pre-processing unnecessarily restricts volume block sizes.

We have introduced the use of dynamic programming for volume distribution and render task scheduling. The moving walls algorithm partitions the scene into cells which each contain one or more closely cropped blocks of volume data. The total size of each group of data blocks is closely matched to the target memory size. Scene partitioning is driven by tightly cropped subvolumes to allow volume boundaries to be followed in a natural manner. Our dynamic programming approach gives the user control over the tradeoff between distribution quality and algorithm run time.

3.4 Dynamic Task Scheduling

In ray tracing mode, each node calculates the ray contributions for the portion of data assigned to it. Ray queues in each node hold pending rays. Each ray tracing node communicates ray information to neighboring nodes. The flex-block dependency graph is used to determine the next neighbor to enter, as selected by the direction of the ray segment. In

ray-cast mode, the ray queues are not used, but instead an ray cast image of the assigned volume portion is rendered locally and sent to the next neighbor node for the view direction of the current frame. Ray-dependencies are collected into a cell-tree which is used to create an efficient cache schedule for the next frame. Each ray has an ordered sequence of cells which it depends on. We call this its ray-cell dependencies. If ray r_2 is spawned from ray r_1, it is defined to be a child of r_1. The ray cell dependencies of ray r_2 are the same as those of r_1 with the addition of any new cell r_2 enters. For each eye ray r, the dependencies of all rays spawned from r form a ray tree. For a 512^2 image, there are over $250,000$ ray trees, each with its own set of dependencies.

We gather these ray dependencies into coherent groups to create a single, consolidated, cell-tree [28]. All rays which traverse from one cell to another at the same relative time are represented by a single link in the cell-tree. Initial rays are projected from the eye through the scene. As these hit the scene objects, reflection, refraction and shadow rays are spawned. A ray dependency occurs when a ray traverses from one cell to another. Clusters of eye, shadow, reflected and refracted ray-cell dependencies are gathered into a compact description; the first ray dependency of each cluster of rays adds a branch to the cell-tree. In our experiments, cell-trees were more than 100 times smaller than the average number of rays represented.

3.5 Test Environment

Preprocessing has been implemented using the Open Volume Library, OpenVL [65]. OpenVL is an extensible plugin-based volume library, which offers support for image and volume file reading and writing which has been developed at Stony Brook University. The flexibility offered by this library has proven invaluable for unique situations which occur in a wide variety of data. For example, the Visible Male data contains non-interleaved RGB data. An incremental change to the OpenVL raw file I/O plugin allowed us to create an interleaved RGB volume from this data. Another small change enabled us to use a

single channel for the region grow algorithm, rather than an extremely slow three channel segmentation.

Distributed ray casting is done on our HP MDS Visualization System. Figure 3.4 shows a block diagram. The MDS cluster has eight render-nodes and one display node with Sepia-2a boards. Each node is an HP dual-processor Pentium Xeon 2.4GHz with 2.5GB memory. The nodes are connected with a high speed backend network for fast sharing of partial images and/or data, as well as a frontend network for process communication and control.

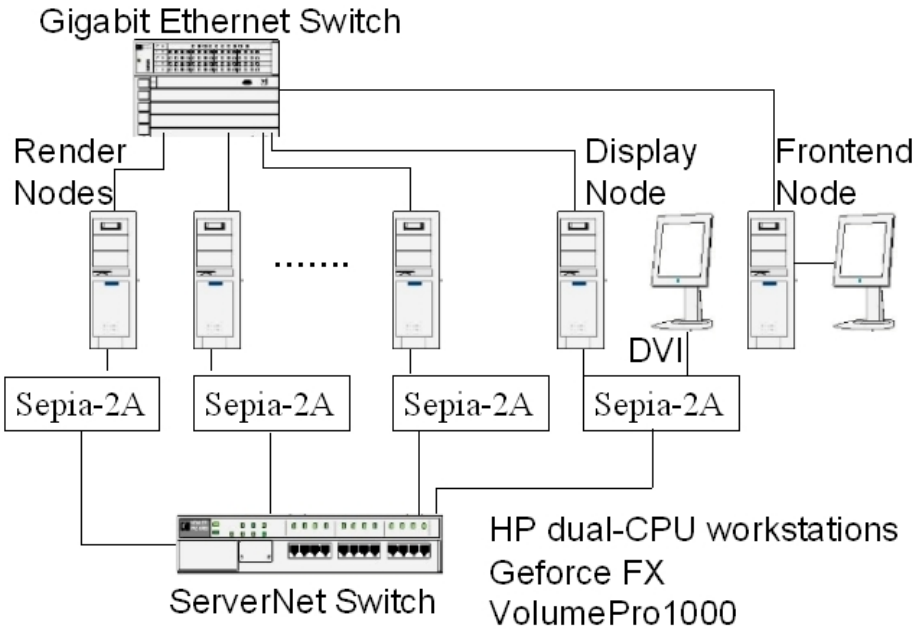
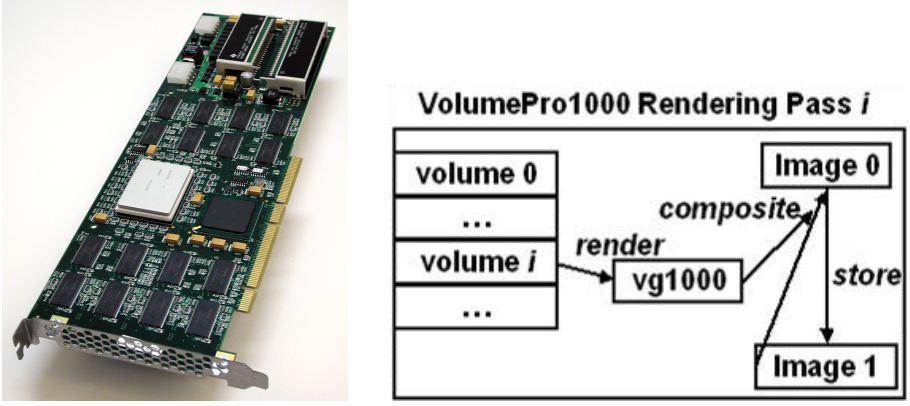


Figure 3.4: HP MDS Cluster block diagram.

Each node is also equipped with a VolumePro 1000 volume rendering accelerator, shown in Figure 3.5(a), with 1GB of memory and a GeforceFX 5800 GPU with 128MB memory. Figure 3.5(b) illustrates the process of compositing an image on the VolumePro 1000. Multiple subvolumes can be stored in the VolumePro 1000 memory. Each subvolume is rendered as a separate pass, allowing subvolumes to be tightly cropped.

The resulting images are accumulated into an image buffer which is located in the VolumePro board memory. The end result is sent to the GPU frame buffer for acquisition by the Sepia-2a card at the end of each frame. Priority order along each axis is either



(a) VolumePro 1000 PCI board (b) Multipass rendering on the VolumePro 1000

Figure 3.5: VolumePro 1000. (a) PCI board; (b) Prior to rendering pass i, *Image 1* contains an image that is the composite of all prior rendering pass results.

strictly front-to-back or back-to-front, as determined from the dot product of the composite direction with the view direction, as shown in Figure 3.6. An extension of this is to compose a grid of images by compositing along one axis direction at a time.

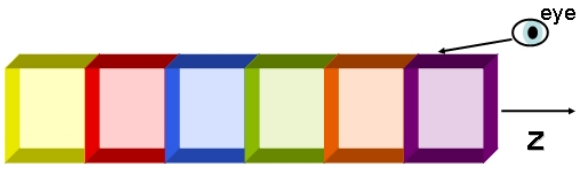


Figure 3.6: Image composition. Image sequence order of a serial composite chain determined using dot product.

VolumePro 1000 rendering on an MDS cluster proceeds as follows:

1. Local processor updates viewing parameters.
2. VolumePro1000 renders, composites subvolumes.
3. Graphics card renders image to video memory.
4. Sepia receives pixels from local video memory and from upstream node.
5. Sepia combines pixels, transmits to next node.
6. Display node receives fully composited image.

3.6 Interactive Visualization Results

Interactive distributed volume rendering is demonstrated in our visualization of seismic data, the Visible Male, and several teeth and fossil data sets. The seismic data (see Figure 3.7) is 8.3GB, consisting of $1834 \times 1382 \times 783$ floating point with four bytes/voxel. Following standard practice of the Society of Exploration Geophysicists, it has been reduced to a 2.1GB greyscale data set. Three VolumePro 1000 boards were used for volume rendering, and Sepia-2a boards [82] for compositing. The end-to-end render time average is 0.45 seconds.

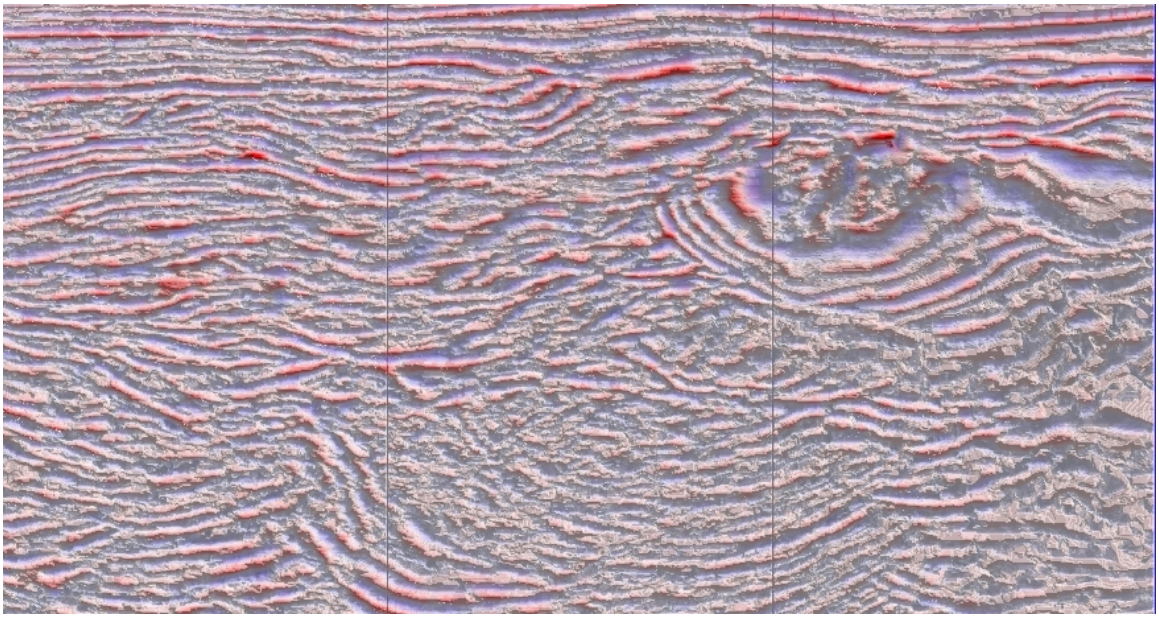


Figure 3.7: Volume rendering of seismic data on a cluster.

The full Visible Male data set, from the Visible Human Project, is a sequence of axial anatomical images of 2048×1216 pixels at 1 mm slices. The Visible Male color data set has 1879 slices. With the addition of an alpha channel the data set is 18.7GB, more than double the capacity of our MDS System VolumePro1000 boards. The 6.7GB homo sapiens molar contains 1589 slices, and the 8.5GB diademodon tooth fossil contains 2028 slices. Figure 3.8 shows frame rates for a varying number of nodes, each rendering a full screen 1280×1024 resolution image of volumetric data which is nearly the same size as

the VolumePro1000 memory capacity.

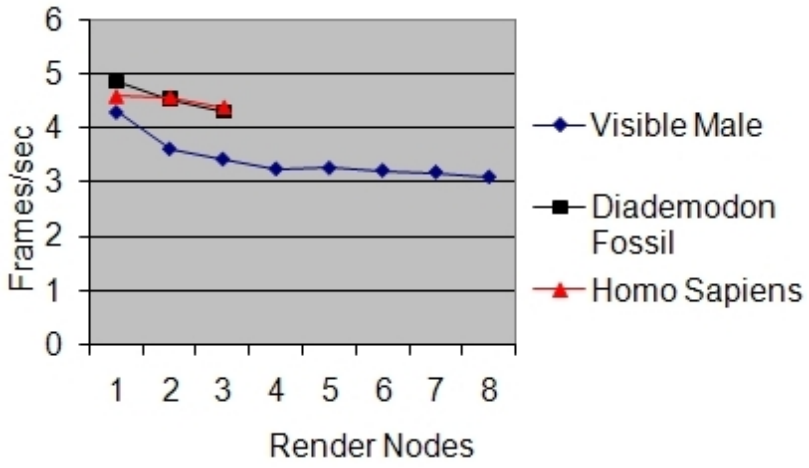


Figure 3.8: Rendering frame rates for 1280×1024 image resolution and approximately 1GB data per node.

3.7 Discussion

We have developed a framework for distributed parallel visualization of massive volumes. Distributed interactive volume rendering poses two related problems. The volume must be split between the nodes to allow for the best load balancing as the camera parameters and other viewing parameters change. At the same time, restrictions imposed by each part of the system on such things as GPU memory and I/O speed must be taken into account for scalability. Data should be distributed to achieve maximum load balance while maintaining a strict view dependent priority order for image composition. The image composition order must be determined quickly in parallel with rendering, in order to avoid reducing the frame rate. Massive volume data sets must be split up, or bricked, to fit within memory capacity prior to rendering.

Our novel dependency graph approach to load balancing allows flexible and efficient render-node assignments. Flex-blocks give more control over allocation of volumetric data

between render-nodes. Image composition order is pre-computed in the flex-block dependency graph. This is extracted from a kd-tree partition of the scene, which is driven by the flex-blocks. By using the dependency graph of tightly cropped flex-blocks for large-scale data management, block-level space leaping effectively occurs prior to data distribution. This is true for both ray casting and ray tracing algorithms. Our approach is general enough to be utilized in any distributed system.

Chapter 4

External Memory Solutions

In this chapter we discuss strategies we have developed for managing out-of-core data In Section 4.1 we discuss the challenges, and we describe the parallel preprocessing used for comparison to our out-of-core methods. In Section 4.2 we give an overview of the data management pipeline. We introduce our *slab-projection slice* in Section 4.3, and our out-of-core bricking in Section 4.4. Our slab-projected kd-tree partitioning is presented in Section 4.5. We describe out-of core cropping in Section 4.6, and our out-of-core region growing algorithm in Section 4.7. In Section 4.8 we present the results of cropping and bricking.

4.1 Challenges in Massive Data Management

Managing massive volumes presents many challenges which arise due to the sheer size, complexity and variability of the data sets. For example, the process of transferring data from one to another is not only tedious, but it is error prone as well. Frequently there is insufficient memory on an interim system, such as the master node on a visualization cluster, and data must be moved in stages. Although the task of moving data can be automated using scripts, precautions must be made to detect when all data has not been moved due to lack of storage or another failure, such as a network interruption.

We address the following specific problems encountered when rendering massive volumetric data sets. The full data set may not fit in the disc memory of a single render-node, so that data must be distributed, rather than replicated on each render-node. In order to have a scalable system that efficiently utilizes the available resources, a load balancing scheme is needed to insure that all render-nodes perform an equitable amount of work during rendering. In addition, data typically does not fit in main memory, so all preprocessing must use either out-of-core or distributed methods. Finally, the data size may exceed total rendering hardware memory available across all rendering nodes, requiring multiple rendering passes. The likelihood of such problems increases with data size. This has motivated us to reduce data as soon as possible in the preprocessing pipeline.

4.1.1 Distributed Preprocessing

Massive data requires either out-of-core (external memory) or distributed preprocessing. We compare these approaches. For distributed preprocessing data must be replicated on render-nodes, and moved again if non-static data distribution is used. Static data distribution doesn't achieve good load-balancing for data sets with large variations in sparsity because partitioning is not guided by the distribution of empty space throughout the data. A full portion of data might be assigned to one or more nodes while some nodes have little or no data to render, so that the speedup obtained by parallel rendering is limited by poor allocation of resources. If this is to be avoided, then data must be moved again after preprocessing.

Data should be distributed in a manner which avoids having a disproportionate amount of the rendering assigned to a few render-nodes. Distributed preprocessing uses a static data structure, such as a grid, for distributing slices prior to reading. In order to avoid unnecessary inter-node image transfers, the granularity of the grid is set so that an equal contiguous portion of the data, one cell, is assigned to each render-node. The grid cell size is equal to the whole scene extents divided by the number of render-nodes. Each raw data slice is sent to every render-node that overlaps it. One drawback to distributed

preprocessing is that empty space is unnecessarily sent to render-nodes.

4.1.2 Empty Space Cropping

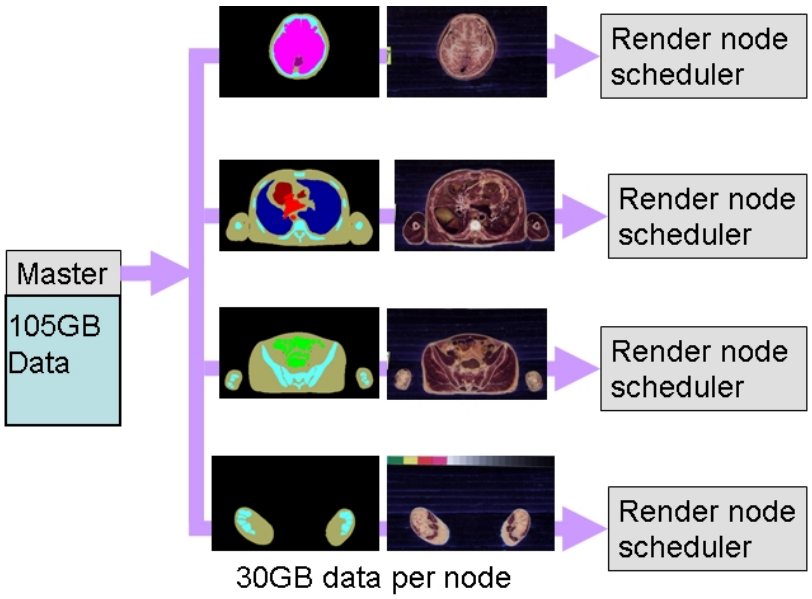
In many data sets, such as the lobster, the Visible Human, Korean Visible Male, and the teeth and fossil data sets used in our experiments, the data is surrounded by some material which is irrelevant to the object being visualized. The challenge is to efficiently skip this empty space. We can reduce the size of the distributed data by modifying the input volume such that it only contains the voxels which belong to one or more regions of interest. We use masking to distinguish this material from the areas of the data which may ever be examined during interactive rendering.

Figure 4.1 illustrates that an order of magnitude is saved by cropping the Visible Korean data set prior to distribution compared with sending full slices. The distributed preprocessing data size shown represents the best case scenario. In addition, slices are replicated for each overlapping cell in the grid. This allows early data reduction.

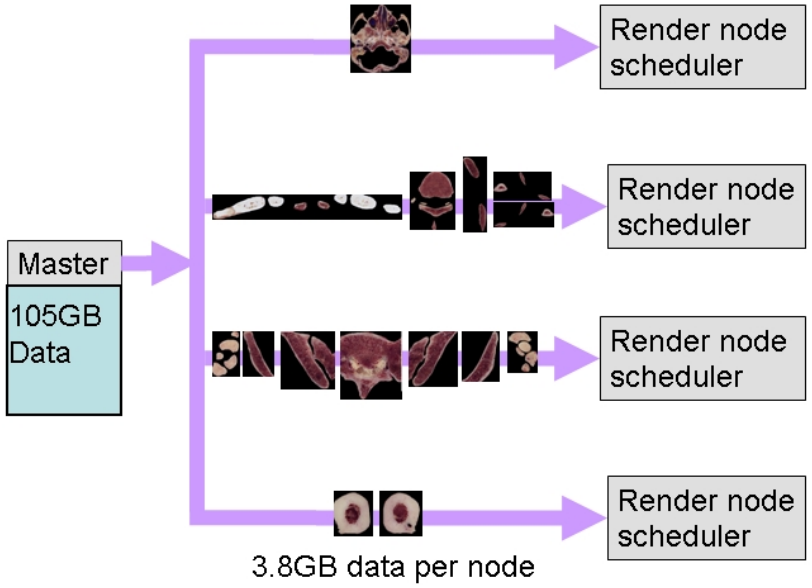
4.2 Out-of-Core Data Management Pipeline

We propose out-of-core preprocessing to minimize the number of times a piece of data is read from disc memory. A volume is defined as massive when it is one or more orders of magnitude larger than the size of main memory available in a single visualization engine. We use early data reduction to remove empty space at the beginning of the pipeline. In order to incorporate empty space information prior to distribution and reduce the amount of empty space distributed to render-nodes, each slice is read, masked and cropped during preprocessing. Traditional seeded region growing [1] runs out of memory in processing a small portion of a massive data set. This algorithm is adapted for external memory implementation. We introduce an incremental slab-to-slab version of seeded region growing, where we read one small z-slab at a time.

This reduces the amount of empty space distributed to the render-nodes and loaded for



(a) Distributed preprocess



(b) Preprocess prior to distribution

Figure 4.1: Distributed preprocessing compared with single node preprocessing: (a) Full slices distributed prior to preprocessing; (b) Cropped slices distributed after preprocessing. Data is reduced by nearly an order of magnitude using empty space removal.

rendering. Data is distributed at the completion of the out-of-core preprocessing. Data is not moved between nodes during rendering. An overview of the pipeline for out-of-core

preprocessing is shown in Figure 4.2.

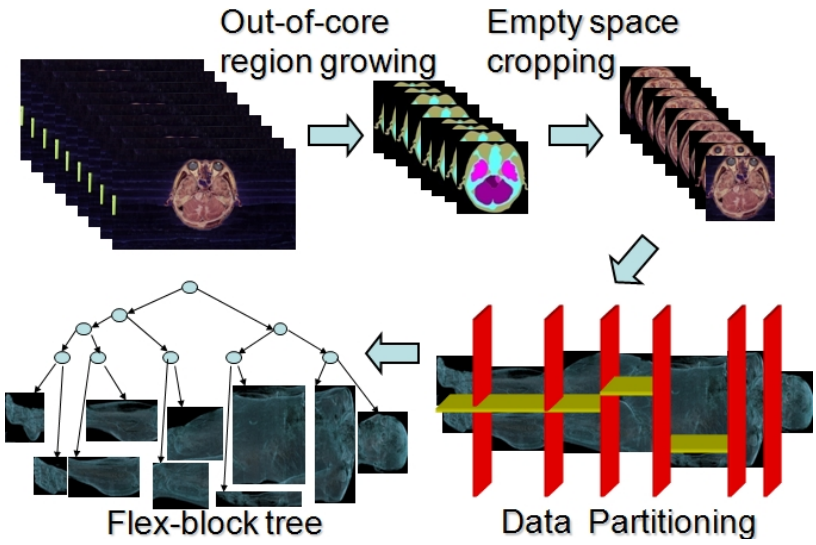


Figure 4.2: Out-of-core preprocessing pipeline. By masking and cropping data early in the pipeline, we are able to used non-empty region bounding boxes for the data partitioning portion of preprocessing"

The preprocessing consists of the slice preprocessing phase, followed by the data distribution process. Slice preprocessing is used for creating cropped slices to be distributed across the network for rendering. Each raw data slice is read once during this phase. Non-empty regions are segmented using a mask volume, then cropped and written to disc memory. Out-of-core region growing is used if no mask has been provided with the data set. Concurrently, data extent information is consolidated into a series of slab-projection slices. Render pass scheduling, hardware ray casting and compositing all take place on the render-nodes during rendering.

4.3 Slab-projection Slice

We have developed an algorithm which selects non-empty voxels in a massive volume data set using out-of-core region growing, and gathers groups of slices into slab-aligned volume blocks for distribution across a volume rendering cluster. Our *out-of-core region growing* algorithm [29] is used to create mask volume slabs. These are assembled by our

region of interest cropping into small volume blocks, which are gathered into spatially coherent groups.

We gather a slab of data into an orthographic projection, the *slab-projection slice* as shown in Figure 4.3. This is used to gather non-empty voxel information for the current slab. Slices are freed from main memory once they are added to the slab-projection slice.

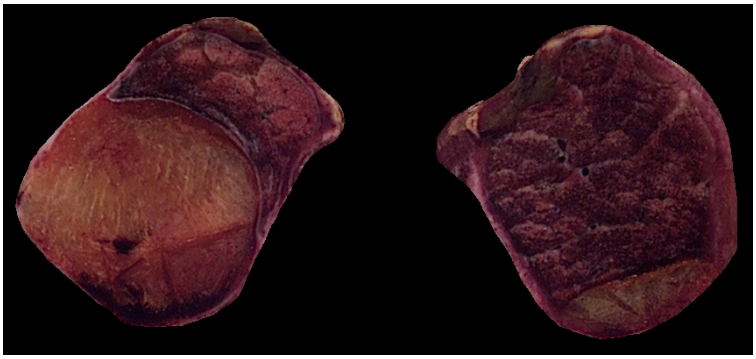


Figure 4.3: Slab-projection of the Visible Korean lungs.

A series of slab-projection slices are created during the initial preprocessing step of masking and cropping. These are used to avoid any further raw data reads during the data distribution preprocessing. We gather a consecutive series of slices, a slab, into an orthographic projection of slice data information, the slab-projection slice. Slices are masked, cropped, and written to disc memory for use during rendering. The number of non-empty voxels within the current slab of data is stored in the corresponding x-y position of the slab-projection slice. This maintains a count of non-empty voxels for the current slab.

A *z-cut* is a cut across the z-axis. If the z-length of the slab is larger than a given slab minimum, and either a split or merge is detected, or if the z-length is equal to a given slab maximum, then a z-cut is made. The concepts of a split and a merge are introduced in [29]. A split occurs when a slice contains two distinct non-empty regions that both overlap a single region in the current slab. A merge occurs when a single region in the slice overlaps two or more regions in the slab. Whenever a z-cut is made, a new slab-projection slice is initialized using the current slice.

We use a series of slab-projection slices for out-of-core preprocessing. The first step

is to gather data extent information from slices. These are used either directly for slab-projected kd-tree partitioning and brick grouping DP algorithms or to find the input brick DAG for DP partitioning.

Data distribution uses either slab-projected kd-tree partitioning or our DP partitioning [30, 31]. For brick grouping DP partitioning, a separate partition is derived for each viewing octant by running DP with a DAG representing the corresponding visibility ordering. Data bricks are replicated as needed across the rendering system, rather than moving data across the network during rendering. The portion of data assigned to a render-node for each unique partition, is stored in local disc memory. A kd-tree partition has the advantage of giving a view-independent solution, but has several disadvantages for massive data sets. We must restrict the number of render-nodes to a power of two, or either allow some render-nodes have more data assigned than others using a non-balanced kd-tree or by assigning more than one leaf node to a render-node.

4.4 Out-of-Core Bricking

Corresponding to the slab-projection slice is a list of *flex-blocks*, which are coherent subvolumes within the slab of volume data which change shape to accommodate the boundaries of adjacent areas in new slices as they are added to the projection slice. A z-cut involves calculating brick bounding boxes and dependencies from the candidate-block list, and the current slab-projection slice is re-set to the current slice. The precedence constraints are defined with respect to the view direction along the positive z-axis. A z-cut is indicated either with a merge or split, or when the footprint of data is close to a multiple of the z-length. A merge is when data from more than one flex-blocks of data overlap within the current slice. A split is when a new region is detected in the data, or length limit. A proportionate footprint indicates that a z-cut would result in well-portioned bricks. A z-cut is also made if the z-length of the slab is larger than a user-specified maximum.

Out-of-core bricking finds a DAG of bricks using the series of slab-projection slices

obtained in the first step of preprocessing. The brick precedence order, with respect to a specific viewing vector, is stored in a DAG, along with brick bounding box extents.

For each slab-projection slice, we find the set of candidate-blocks, or cropped, non-empty regions. From these candidates we cut bricks of a predetermined size. For GPU rendering, the bricks should be sized to fit the maximum texture memory and satisfy any hardware-specific dimension requirements. However, bricks of this size may be too small to control the algorithm run time for DP partitioning. In this case, larger bricks are used for data distribution, and these are cut to the target GPU texture size prior to rendering. We use bounding boxes of bricks, not physical data, in our brick grouping DP. At render time these bounding boxes are used to create volume texture bricks, sized to fit the rendering hardware, from pre-cropped slices.

4.5 Slab-projected Kd-tree Partitioning

A traditional kd-tree is obtained through recursive splitting of data using empty space information. Each recursion requires one or more scans of every data slice. In order to avoid these memory accesses during the partitioning process, we introduce the slab-projected kd-tree partitioning algorithm.

The input to each recursion of the kd-tree partitioning includes a set of volume data slices, current sub-scene Q, the current depth of the recursion, d, and a cut plane. The center of a sub-scene is the plane that splits the non-empty voxels into nearly equal numbers across the cut axis. At each recursion, the sub-scene is split with the center cut plane, and then each side is recursively partitioned. The kd-tree root represents the whole scene, and the depth is initialized to 0. The recursion ends when the number of leafs at the current level is the same as the number of render-nodes, B. This requires the number of render-nodes to be a power of two. The final depth of the kd-tree is $b_d = \log_2 n$.

We use the number of non-empty voxels as the criteria for finding the center of a sub-scene. The set of data slices that overlaps the current sub-scene is S. For each voxel

position along the cut axis there is a data slice, s, that is orthogonal to the current cut axis. The set of these cut crossing slices is Q, and $Q = S$ for the z-cut axis. The number of non-empty voxels in slice i is $i.voxelCount$, where i is either a data slice or a cut crossing slice.

For each recursion of the partitioning algorithm, every data slice, s, that overlaps S is examined. If it is not in main memory it is read from disc memory. Traditional kd-tree partitioning proceeds recursively as follows:

$\forall s \in S$, read s and scan to get $s.voxelCount$
  Add $s.voxelCount$ to sub-scene total
  $\forall q \in Q$ update $q.voxelCount$ from intersection with s
    Find center cut plane using voxel count array
    if $d < b_d$ then find kd-tree for each side of cut

The problem with this approach is that every slice in the current scene needs to be read in order to determine the splitting axis. Instead, we use a series of slab-projection slices in the slab-projected kd-tree partitioning algorithm. The slab-projected kd-tree partitioning solution proceeds in the same way as a traditional kd-tree except that a series of slab-projection slices provides non-empty voxel information instead of raw data slices. The slab-projection slices are produced during the first step of preprocessing, as described in Section 4.3.

At the start of the algorithm, all slab-projection slices are assumed to be in main memory. The minimum slab width in the slab-projection slice preprocessing step is used to insure that this criteria is met. This circumvents the need to read any data from disc memory during kd-tree partitioning. The main advantage of using slab-projection slices is that empty space information is consolidated prior to the partitioning process. This has a major impact on the time spent in determining the splitting plane at each iteration of the recursion.

4.6 Region-of-Interest Cropping

When the data manager processes a data slice for the first time, it is reduced by cropping it to the bounding box of the non-empty voxels. Storing the cropped slices is an important step in managing massive data. The time required for creating a block from cropped slices rather than full slices is reduced proportionately to the slice size reduction. Data is loaded into rendering hardware memory from pre-cropped slices according to the data bounding boxes in the flex-block data structure. By pre-determining the size of each block, memory allocation is efficient. The memory required to fit a volume block with full slices may not be available for the prescribed z length, since only a small portion of the full slices fit in memory at once.

As part of our out-of-core strategy, the data inside the flex-blocks is not loaded into memory for our load balancing data distribution preprocessing. Only the bounding boxes of bricks, not physical data, is used to drive the algorithms. Data distribution is re-calculated for different cluster configurations, and stored in the form of a flex-block tree. The leaf nodes, flex-blocks, are used to create physical volume blocks at render time.

In our approach, the volume is broken into small, segmented non-overlapping volume blocks, which are recombined into larger subvolumes. We save the cropped slices to disk memory and gather virtual volume blocks by comparing only the most recent slice with the current slice and incrementally updating volume block bounding boxes. Our data distribution management algorithm uses the bounding boxes of these volume blocks, and only creates the actual blocks at run time. The region grow algorithm is run on each distinct region of interest within a slab. A threshold range and the initial seed for each separate region within the whole volume are provided by the user.

4.7 Out-of-Core Region Growing

Methods for masking a region of interest have been researched for many years. Region growing is a technique to extract a connected region from a 3D volume based on some pre-defined connecting criterion. In the simplest form, region growing requires a *seed* point to start with. From the seed point, the algorithm grows until the connecting criteria is satisfied.

Our *out-of-core region growing* algorithm [29] creates a mask volume for a massive data set. Our algorithm selects non-empty voxels in a slab of data using seeded region growing [1]. The output mask volume which is used for volume segmentation in our region of interest cropping. We crop the segmented volume to the minimal axis-aligned cuboid which contains non-empty voxels. The last slice of the mask volume supplies a set of independent region seeds for region growing on the next slab. Special consideration is taken to deal with regions splitting and/or merging within a slab. Slabs of approximately 20 slices work well. These are each segmented using seeded region growing. For each detected region a volume *block* is created, which is cropped to the region's axis-aligned bounding box. The block's position and size are retained for later volume distribution steps.

Out-of-core region growing requires detection of overlaps, splits and merges at slab boundaries. The quality of a volumetric data set is not always reliable. There may be missing or damaged slices, which may cause region of interest boundaries to be indistinct and disconnected. This makes it necessary to write robust code for segmentation which can detect or adapt to these problems without compromising the integrity of the data set. If a specific data set has a known problem, processing may be tweaked as a work-around. However, a work-around for one data set will probably not process others correctly. For example, there are several empty slices in the Visible Human data, which must be ignored. However, there may be a data set where it is known to have empty regions that include full empty slices. We set the default to treat empty slices as empty regions, but allow a flag to be set to indicate that empty slices are ignored for region continuity purposes.

A mask volume is produced during region growing. The last slice of this mask volume, the *mask slice*, is used for seeds in region growing on the next slab. If this slice contains

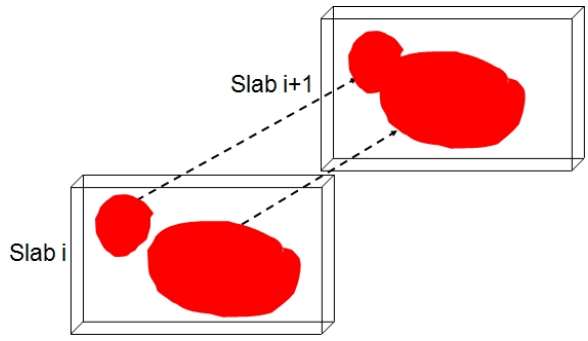


Figure 4.4: Out-of-core region growing. Continuity between slab seeds is maintained by using the final mask slice output from region growing on the previous slab, as a seed slice. Separate regions merge within the region defined by slab i. This is detected by region growing in slab $i + 1$.

more than one distinct region, we say that the volume has a *split* within that slab. We must insure that a separate region grow process is initiated for each distinct region of interest in the next slab which is connected to this mask's region of interest. If there is only one region, it means that the two regions *merge* in the next slab. In this case, we must be careful not to produce duplicate copies of the same region. Thus, before using a seed voxel from the mask slice on slab s we verify that it has not already been included in a segmented region for s.

Volume blocks and their dependency graphs are built as raw data slices are gathered into volume blocks. A link is created between two volume blocks whenever a split or merge is detected within in a data slice and when a new volume block is generated because a user-specified size has been reached. The resulting dependency graph is used for data distribution and for image composition order computation or ray-task scheduling. As volume blocks are created they are grouped together.

A *block group* is a consecutive group of volume blocks which have common faces, have contiguous nonempty voxels, and do not have a natural breaking point, such as a local minimum slice or empty voxel region. Each block group bounding box and volume block file list is maintained for the later steps in data distribution. A block group is opened when a block is encountered which does not belong with an existing block group, and it is closed

when the volume defined by that block group encounters a local minimum x-y slice. If regions merge within a slab, then the block groups from each branch are closed and a new one is opened. An active block group list is maintained which contains all block groups which are still open.

Out-of-core region growing and cropping proceeds as follows:

For each slab s:
Use seeded region grow segmentation on s.
Create cropped volume block b for each region in s.
For each block b:
If b belongs with any active group g add b to g.
Else start a new group.
Close active groups that have no new blocks.
Use the final slice mask as seeds for next slab.

4.8 Early Data Reduction Results

The effectiveness of out-of-core region growing and cropping is demonstrated by rendering and compositing the Visible Male photographic image data, and several very high resolution CT scanned teeth and tooth fossils. Volumes rendered include the full Visible Male photographic image data set, a cebus apella (spider monkey) tooth, a diademodon fossil, a baboon fossil, and a homo sapiens molar.

Figure 4.5 shows a volume rendered and composited image of the full resolution Visible Male using eight render-nodes on our MDS cluster. By default, a volume is distributed over the minimum number of nodes required to fit it. Alternatively, the user may request that a particular number of nodes be used. Each node renders one or more subvolumes. Each subvolume requires a separate rendering pass. In addition, one or more slices of voxels must be replicated between adjacent subvolumes for correct interpolation and shading during

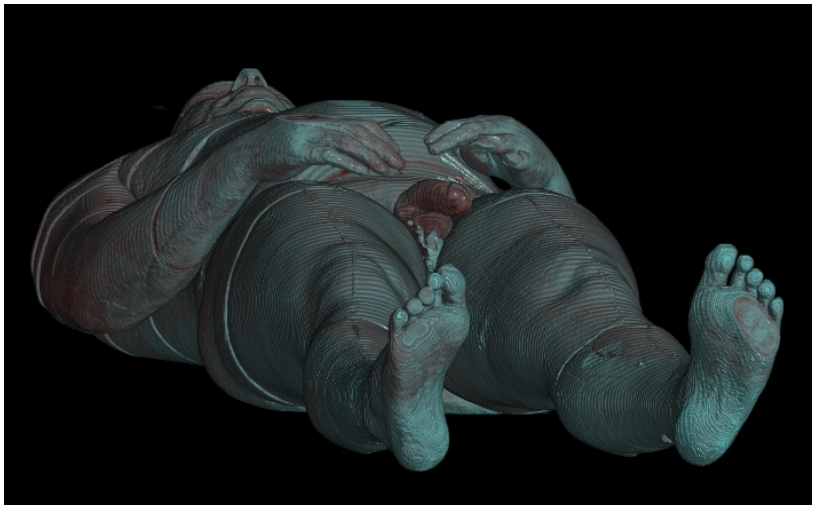


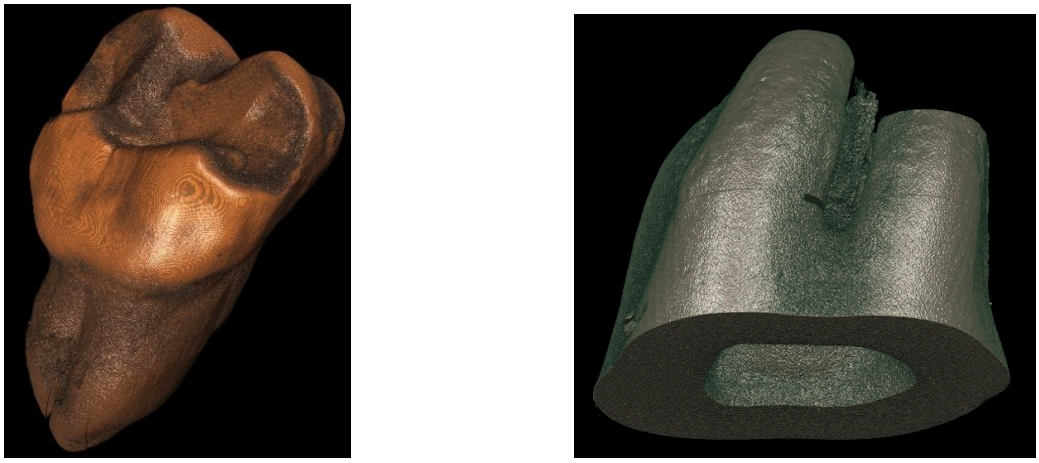
Figure 4.5: Composited full resolution four-channel Visible Male. Data resolution: $2048 \times 1214 \times 1879$. Image resolution: 1280×1024.

rendering. We use distributed volume rendering on these data sets. The Visible Male data is used to stress the MDS Visualization System, requiring all available VolumePro memory even after being cropped. The result of our out-of-core region growing and cropping allows the full-scale Visible Male data to fit within the VolumePro1000 memory of these eight-nodes without any down-sampling. Region of interest cropping reduces volume sizes for the full Visible Male set and several CT scanned teeth and tooth fossils.

Figure 4.6(a) is a volume rendering of a cebus apella tooth data set. The 2.8GB data set contains 663 slices. The 6.7GB homo sapiens molar contains 1589 slices, and is shown in Figure 4.6(b). An volume rendered image of the 8.5GB diademodon tooth fossil, containing 2028 slices, is shown in Figure 4.7(a). A baboon fossil data set is volume rendered in Figure 4.7(b). The data set is 6.6GB and contains 1570 slices. These were rendered on a the eight-node MDS cluster, with frame rates ranging from 6Hz to 20Hz. In our experiments, data is reduced by an average of 68%, and frame rates range from 6Hz to 20Hz.

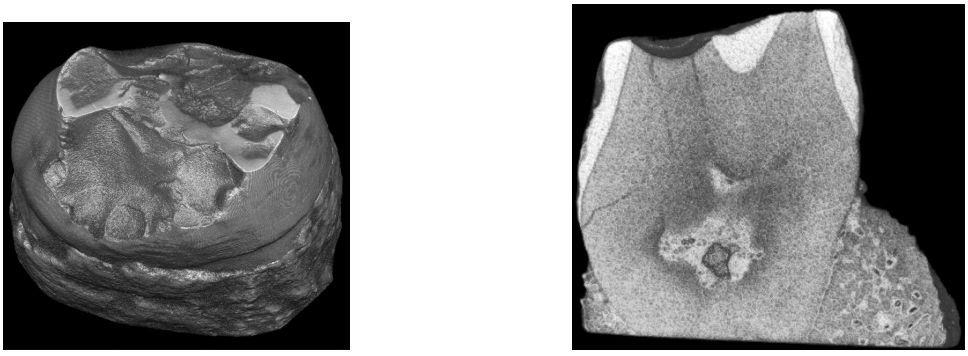
In our experiments, data is reduced by an average of 68% (see Figure 4.8).

Table 4.1 shows the preprocessing time, including slice reading, region growing, and cropping, for these data sets. We were unable to compare with traditional region growing because the algorithm memory requirements exceeded memory available on our CPU.



(a) Cebus apella tooth (b) Homo sapiens tooth

Figure 4.6: Volume rendered teeth images.



(a) Diademodon fossil (b) Baboon fossil

Figure 4.7: Volume rendered fossil images.

4.9 Summary

In our framework, we use early data reduction to reduce problems associated with distribution and rendering of massive volumetric data sets. External memory solutions are of paramount importance in managing massive data for distributed rendering. We have developed an out-of-core region growing algorithm for pre-processing massive volumetric data, and a scheme for distribution of this data on a volume rendering cluster. The region of interest cropping algorithm assembles raw data slices into small volume blocks and gathers these blocks into spatially coherent groups. We crop the segmented volume to the minimal

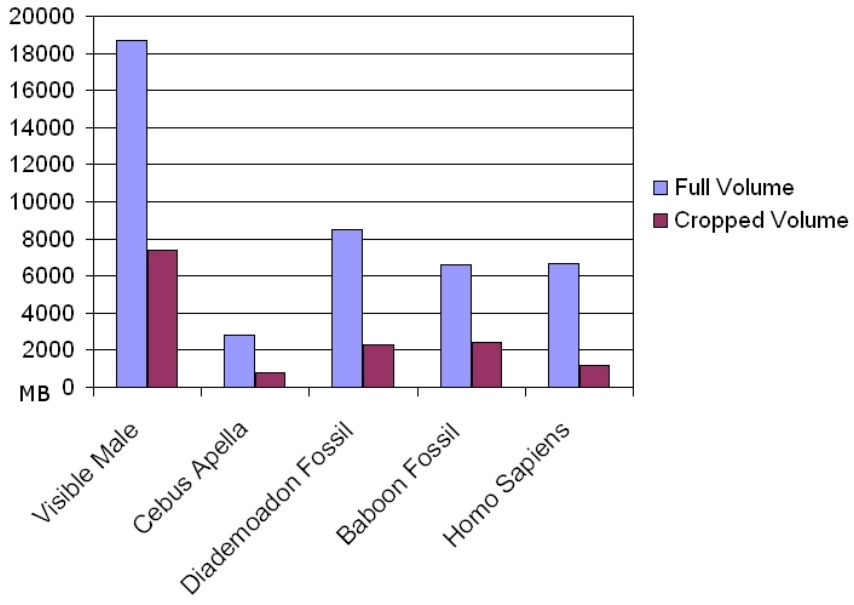


Figure 4.8: Data storage reduction after cropping and data distribution.

Table 4.1: Volume sizes (MB) and preprocessing (sec).

	4-Channel RGBA	1-Channel CT			
	Visible Male	Cebus Apella	Homo Sapiens	Diademadon Fossil	Baboon Fossil
Image	Fig 4.5	Fig 4.6(a)	Fig 4.6(b)	Fig 4.7(a)	Fig 4.7(b)
Volume size	18,717	2,780	6,665	8,506	6,585
Preprocessing	7,800	921	1,605	2,440	1,398

axis-aligned cuboid which contains non-empty voxels. The *slab-projection slice* collects data extent information used for our out-of-core bricking algorithm. Our results indicate that this is a very effective strategy for data sets with large contiguous portions of empty space.

Chapter 5

Load Balanced Network Distribution

In this chapter, we define the *load balanced network distribution (LBND)* problem, and map it to the NP-complete precedence constrained job-shop scheduling problem. As systems, memory, and data sets continue to grow, so does the problem of managing these resources and the data sets they are capable of rendering. Dynamic programming (DP) optimizes a cost function while meeting a set of constraints. Due to the wide variety and the frequency of changes in visualization system hardware, we are motivated to find a solution to the LBND which can adapt to new hardware as it is introduced; this is achieved by defining an appropriate cost function and constraints for a DP solution. We present two solutions to the LBND problem that are particulary applicable to scenes with large portions of unevenly distributed empty space. The first, out-of-core *slab-projected kd-tree partitioning*, uses non-empty voxel information collected in a series of slab-projection slices. The second, *brick grouping*, inputs a directed acyclic graph (DAG) of data bricks and finds an optimal partition with respect to the given cost model. We attempt to minimize a cost function that reflects the end-to-end rendering cost on a cluster. Bricks are sized for rendering hardware, and the DAG represents their view-dependent precedence order.

The specific contributions discussed in this chapter are:

- Definition of the LBND problem

- Mapping of LBND to job-shop scheduling
- Brick grouping DP algorithm

We have rendered several segmented portions of the Visible Korean Human [87] on the Stony Brook Visualization Cluster. The segmented data does not preclude the use of a transfer function to render translucent or color-enhanced images, but is used because it provides a good cross-section of data with varying amounts of empty space.

5.1 Distributed Volume Rendering Overview

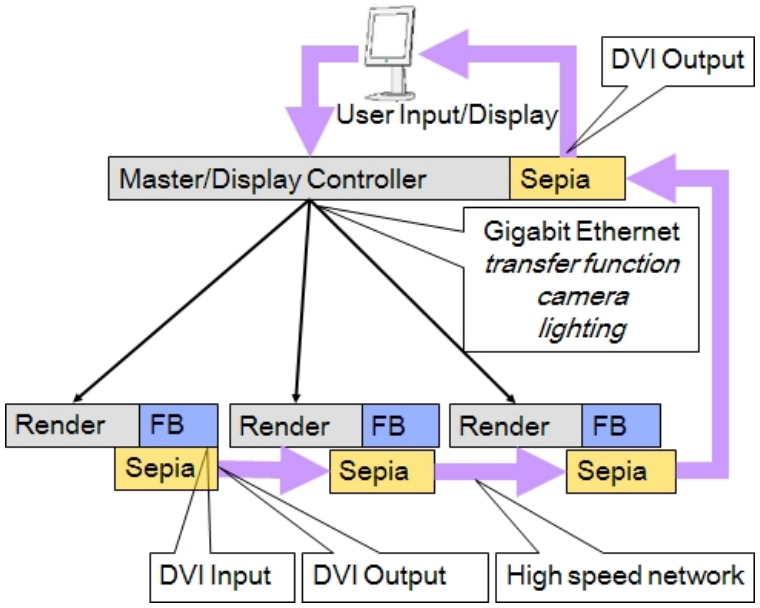


Figure 5.1: Distributed ray cast rendering block diagram for a sort-last architecture.

An overview of our distributed ray cast rendering is shown in Figure 5.1. A volumetric data set in this context consists of a 3D grid of information at sample locations, or voxels. The information is either a scalar value such as density, or a vector such as color in a photographic data set. Volumetric ray casting is a technique in which one or more rays emanates for each pixel and each ray accumulates the color and opacity contribution of a series of voxels along the ray in the volume data. Our visualization cluster is a network of

PC nodes connected with a high speed backend network for fast sharing of partial images, as well as a frontend network for process communication and control.

Interactive rendering is done in parallel as viewing parameters are parsed and distributed by the master node. Rendering is performed on individual render-nodes by VolumePro1000 hardware or a GPU, and the final image for each frame is a composite of the resulting images. The alpha blending equation used for image compositing requires the relative depth of each image. The visibility ordering is unique for orthogonal projections along one of the octants of a cube centered at the origin; images are composited in the depth order for the current view direction. Both the GPU and VolumePro 1000 use parallel, orthographic projection, ray cast rendering; perspective rendering requires more complex image alignment, and is not used in our system.

Inter-node image composition is controlled by the master node, but takes place locally on render-nodes (either in the GPU or in Sepia-2a compositing hardware). The rendering time on each node for a given image size is roughly proportionate to the size of volume rendered. Other factors include sampling rate and the cache coherency of the volume. Early ray termination is programmed into the rendering hardware. However, taking full advantage of this requires that data be redistributed between nodes as the opacity changes, and the network transfer cost is relatively high in our system, so we do not move data during rendering.

5.2 Load Balanced Network Distribution Problem

For a given scene of volumetric data and a given system configuration, our goal is to distribute data across the system to optimize available rendering resources. The problem input includes a scene with volumetric data, a description of the distributed system configuration, including the number of render-nodes, rendering and local composition costs, network transfer costs, and the memory capacity of each render-node.

We use multipass rendering within each render-node; the same hardware is used to render an image for each of several different bricks of data, each in a separate pass. Images within the same render-node having consecutive depth priorities are composited into a single image prior to inter-node composition. We define a *cell* to be any section of the scene which can be assigned to a render node and all sub-images produced from a cell can be composited without interleaving with images produced in another cell. The final image must wait for every rendering pass to finish on each render-node, so end-to-end rendering time is a function of the slowest renderer plus inter-node composition time. For every image requiring inter-node composition, additional network communication is required.

The input to the LBND is a volumetric data set, along with render and network cost information. A volumetric data set consists of a grid of voxels, each defining a scalar or vector field. The precedence constraints between voxels are defined by the relative distance to the viewpoint along the view direction. The output is a render-node assignment, which minimizes the total runtime cost, does not violate any physical resource constraint of the system, and observes the precedence order for image composition. The LBND problem is an instance of the job-shop scheduling problem as demonstrated by the following mapping.

The input to the job shop scheduling problem is a list of jobs, with associated resource requirements, and a DAG of job dependencies. The goal of the problem is to schedule each job to a shop, or resource, and minimize the overall job completion time, without violating resource or job dependency constraints. Data voxels are jobs, rendering memory is the limited resource required for each job, and the precedence constraints are given in a DAG which is derived from the relative distance of voxels to the viewpoint along the view direction. The goal is to create a scene partition, or schedule, which minimizes end-to-end rendering time, or job completion time.

5.2.1 Graph Partitioning Software Packages

The graph partitioning problem occurs in many different contexts in computer science, including applications such as differential equation solvers and finite element computation [26]. For parallel processing, a computational task is partitioned, and the sub-tasks are assigned to different processors. The communication cost for dependent results is a key issue in the load balancing efficiency of the partition. Several graph partitioning algorithms have been proposed. Algorithms including the hypergraph approach [86], terminal propagation [43], and the multiple constraint method [52], have been implemented in the software packages, including METIS [53], and Chaco [44]. The LBND problem is a graph partitioning problem. However, the edge-cut metric used in algorithms implemented in METIS and Chaco do not adequately describe the physical characteristics of the image composition constraint in the LBND problem. This requires each partition to define a plane aligned surface area, and the cost of each of these planes consists of a relatively high fixed cost and an often inconsequential per pixel cost for the data transfer. Two solutions to graph partitioning that address this are octree and kd-tree partitioning [7]. Algorithms for finding these partitions require external memory solutions for distributing massive volumetric data sets. Our slab-projection kd-tree partitioning 4.5is a solution that addresses this issue. However, the kd-tree solution has some shortcomings as well, as described in Section 5.3, so we propose a dynamic programming (DP) solution as well.

5.2.2 GPU Multipass Partitioning

Task scheduling for multipass GPU rendering requires local partitioning, and the smallest resolution cell is sized to fit the target rendering hardware. The drawback to using traditional partitions, such as an octree, for LBND is that a scene cannot be partitioned to the granularity needed for reducing sufficient empty space. Network distribution partition cells are typically orders of magnitude larger than those for local partitions. Such large partition cells result in poor load balancing for any data set with a non-uniform distribution of empty

space.

Although the underlying structure of these problems are the same, there are several differences between them. For example, the cost calculation used in the DPMP [41]P solution to the multipass partitioning problem is based on resource usage of compiled shader code; this is measured as part of the optimization decision in DPMPP. Partition costs are directly calculated from the DAG of bricks in our LBND solution. The cost function for end-to-end render time depends on the time of the slowest renderer in LBND, a function of the maximum number of bricks assigned to a render-node. The cost function for DPMPP is a function of the sum of *operations*, which reflects resource use in total across the whole shader program. We start with a high level overview, then describe our solution in terms that are comparable to those used in DPMPP.

5.3 Dynamic Programming

The kd-tree solution does not attempt to solve an explicit cost function but rather operates under the assumption that evenly distributing the data is always the best solution. It also restricts the number of render-nodes to a power of two. We propose a DP partitioning solution that can readily adapt to different system configurations by using a well-defined cost function, and does not have any restriction on the number of render-nodes.

5.3.1 Brick Grouping DP Overview

We propose a LBND solution which starts with a set of non-empty bricks that covers all regions of interest in the volume, connected with a DAG that represents the viewing order of these bricks. The general idea is to split the scene into cells, which each contain one or more connected bricks. Figure 5.2 illustrates a partition of a DAG of bricks into a DAG of cells. This is a feasible solution where each cell represents a render-node assignment. In the following discussion, we use upper case variables for input variables, and lower case for calculated variables and indices.

The solution is found incrementally in stages, where a stage corresponds to a brick in the input DAG, and with each additional stage, one additional brick is added. We use b_i to label the brick in stage i. Each feasible partition p, for stage i, is defined by a DAG of cells containing all bricks in stages from i to $B-1$, where there are B bricks. The overall optimal solution is the optimal partition for stage 0, which is actually the partition for all stages and represents a render-node assignment.

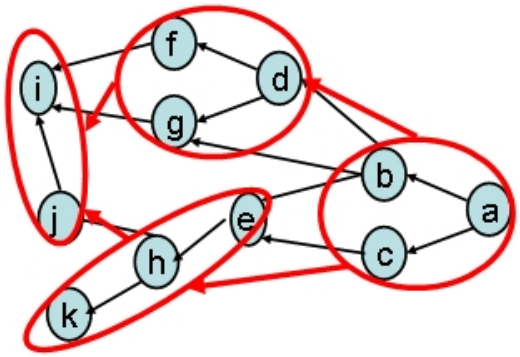


Figure 5.2: Partition of a DAG of bricks (blue) into a DAG of cells (red). This is a feasible solution where each cell represents a render-node assignment.

Cells in partition p are numbered 1 through p_r, where p_r is the number of cells in p. The number of bricks in the k^{th} cell in partition p is n_k. Each cell corresponds to a composited image to be produced by a render-node, where this image is not interleaved with any other render-node image. Interleaving occurs when the number of cells in the partition exceeds the number of render-nodes, so that there exists at least one render-node, a, such that every image rendered on a cannot be combined locally. In other words, for some images, v_1 and v_2, rendered on v, and image w_1 from a different render-node, the relative priority order is v_1, w_1, v_2. This results in an additional inter-node composition and network transfer, and may require additional image buffer space for storing v_2 until the v_1, w_1 composited image is ready.

If the network transfer time is high compared with the rendering time, then interleaving images between render-nodes results in a slower end-to-end rendering time. Good load balancing is achieved if each render-node is assigned a proportionate amount of the data,

and not necessarily by minimizing the sum of rendering times. If there are N render-nodes, then the total number of partition cells in the optimal solution with a high network transfer cost is X. We use this to prune the feasible solution search space by placing an upper bound, M, on the number of bricks per cell, where M is derived from the total number of non-empty voxels divided by N. This is the equivalent of the resource constraint of MPP, and allows us to define our DP solution using the same terms as the DPMPP solution.

5.3.2 Objective Function

We use a cost metric to choose the optimum partition at each stage. For partition p, the cost includes the direct rendering time per brick, R, plus local image composition time, L, plus network transfer time, X, which includes inter-node image composition. Rendering time is defined in terms of rendering hardware texture bricks. If the input bricks are a multiple of this size, as explained in Section 5.3.5, then the cost function is scaled accordingly. The correctness of the solution is determined by the accuracy of the cost model.

The cost function presented here reflects the costs of our current cluster implementation. An advantage to using dynamic programming is that the global cost formula can be adapted to reflect different implementations such as multi-threading. Each brick is projected locally and projection sizes do not vary significantly for orthogonal projection rendering with a given viewing vector, so a constant rendering time is a good approximation. Although a variable rendering time could be used, a constant simplifies the cost description. The cost of image composition depends on whether any contributing image is sent over the network or not. Local image compositing takes place in the rendering hardware. The most recently rendered image is composited with another image, which must be loaded into memory. Inter-node image compositing includes network communication time as well.

The input to our problem includes a volumetric data set consisting of a set of brick bounding boxes, a DAG representing the image composition priority order of these bricks, and the following set of rendering system parameters:

B: number of bricks

N: number render-nodes

M: maximum bricks per cell

R: rendering time per brick

L: local composition time

X: network transfer time

The network transfer cost for p is $X * (p_r - 1)$, the rendering cost is $max(R * n_k)$, $\forall k \in p$, and the local image composition cost is $\sum_{k=1}^{p_r}(L * (n_k - 1))$. The total cost of p is:

$$c_p = X * (p_r - 1) + max(R * n_k) + \sum_{k=1}^{p_r}(L * (n_k - 1)) \tag{5.1}$$

5.3.3 Locally Optimal Solution

Stages are processed in decreasing order. At each stage, a list of feasibly optimal partitions is determined. Each partition represents adding the brick associated with the current stage, to some partition encountered so far. We determine the cost of every feasibly optimal DAG of cells for each stage, i, and mark the lowest cost of these as the local optimum. A feasible cell is one that has a neighbor of brick I (corresponding to stage i), and contains no more than M bricks. At each stage, each feasible cell is considered. The algorithm compares a set T of transitions. Transition t consists of $t.postcondition$, $t.operation$, $t.cost$, and $t.solution$. Partition $t.postcondition$ contains bricks for stages $i + 1$ to $B - 1$, and $t.operation$ involves either concatenation of brick b_i to a cell in partition $t.postcondition$, or the creation of a new cell containing only b_i. The cost of each transition, $t.cost$, is computed as described in Section 5.3.2. Partition $t.solution$ contains bricks for stages i to $B - 1$. The notation here is similar to that used in DPMPP, except that we use $t.solution$ instead of $t.precondition$.

The goal is to find a partition which assigns approximately the same portion of the volume to each render-node, without violating the given precedence order. The optimal transition for stage i is $t^*[i]$, and $c^*[i] = t^*[i].cost$ is the cost of optimal solution for stage i. We use brackets to distinguish stage optimal transitions from the following interim values, which are calculated for adding brick b_i to partition $t.postcondition$:

$t_0.solution$ = $t.postcondition$ plus new cell containing only brick b_i

$t_k.solution$ = $t.postcondition$ with brick b_i added to cell k

Only feasible transitions are considered. The optimal groups of cells for different stages can overlap. A globally optimal solution is obtained using DP because the complete set of feasibly optimal solutions is evaluated at each stage, and the optimal path through these solutions is traversed from the initial stage for the final solution. The local optimality condition ensures that the largest cells are selected first. The algorithm processes the brick DAG starting with the last node.

In summary, DP partitioning stage i, entered with transition set T,where k is a cell, proceeds as follows:

$\forall t.postcondition \in T$
  Create $t_0.solution$
  Calculate cost
  $\forall k \in t.postcondition$ that contains a neighbor of b_i
    if $|k| >= M$
      Create $t_k.solution$
      Calculate cost
t_i^* = lowest cost transition

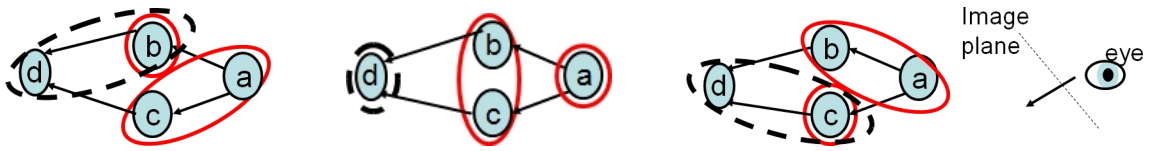
Stage 0 is the final stage evaluated, and the solution partitions the original scene. It minimizes the objective function, $c^*[0]$, while satisfying the following constraints:

1. $t^*[0].solution$ assigns each brick to exactly one cell

2. Input brick DAG precedence order is not violated,

5.3.4 Example Stage

At each stage, the set of feasibly optimal partitions is compared. Each connected set of bricks with up to M bricks, which includes the new brick, is considered. Figure 5.3 illustrates stage d, which compares feasible solutions for the placement of brick b_d, in a system where $M = 2$. The value of $M = 2$ is not typical, but is used here for a simple illustration. Stage d is entered with postconditions p_0, p_1, and p_2. Brick b_d can be concatenated to a cell in both p_0 and p_2, but for p_1 the only cell with a link to brick b_d already has the maximum number of bricks, so a new cell is started. In addition to the solutions shown are partitions with brick b_d added as a separate cell to p_0 and to p_1.



(a) p_0 (b) p_1 (c) p_2

Figure 5.3: Stage d with $M = 2$. Evaluate partitions that add brick b_d to postconditions p_0, p_1, and p_2. Not all feasible solutions are shown. (a) Brick b_d concatenated with b_b; (b) New cell is started; (c) Brick b_d concatenated with b_c.

5.3.5 Algorithm Analysis

The principle of optimality [8] states that the optimal solution for the current stage is optimal regardless of what policies or conditions led to this stage. If every feasibly optimal partition is considered for each stage, then the resulting solution is globally optimal because it observes this at each stage.

Stage i is entered with set p_{i+1}, where p_{i+1} is the set of partitions $t.solution\ \forall t \in t_{i+1}$.

The time-complexity for this algorithm is $O(\sum_{i=0}^{B-1}(|P_i|))$ for B stages. For a DAG constructed from a tree with the addition of a sink node that is the child node of all leaf nodes, where A is the average number of children per node in the brick DAG, there are A^m unique cells of an arbitrary size, m, adjacent to i, and each of these is part of a unique partition. Stage i solutions include each of these concatenated with b_i, and also includes each of these with an additional cell containing only brick b_i. This represents the worst case, since some of these solutions are not unique for a general DAG. If cells are restricted to contain no more than M bricks, then the number of unique solutions is $|P_i| = \sum_{i=1}^{M-1}(A^m) + \sum_{i=1}^{M}(A^m)$. This is controlled by the maximum number of bricks per cell, M.

In order to prevent the DP algorithm from becoming intractable we reduce the average number of feasible solutions. We do this by increasing the size of input data bricks to reduce the average number of bricks per cell in each feasible solution. The size of bricks is calculated so that the average number of bricks per render-node is less than the number, M, set by the user prior to building the DAG. Brick dimensions are each a multiple of the target GPU texture brick size. Bricks are further divided up into the final textures using a local grid prior to rendering. The DP solution obtained is optimal with respect to the input bricks.

5.4 Results

To validate our slab-projected kd-tree partitioning and brick grouping algorithm we have used them to distribute data for the Visible Korean. We compare these with a grid partition, preprocessed in parallel on the render-nodes. We use a 3D grid, which defaults to 2D for all clusters in our experiments except for the 64 node. The granularity of the grid used for distributing data between nodes is set so that the number of grid cells is equal to the number of render-nodes. For the Visible Korean data set, the z-axis is four times as long as the x and y axes, so the kd-tree algorithm is modified slightly to start with several z-cuts.

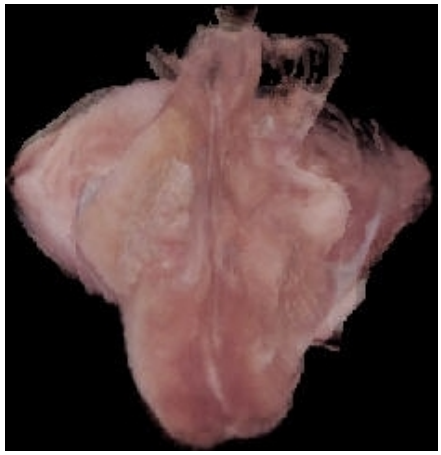
We compare out-of-core preprocessing to distributed preprocessing. Distributed preprocessing uses a static data structure, such as a grid, for distributing slices prior to reading. In order to avoid unnecessary inter-node image transfers, the granularity of the grid is set so that an equal contiguous portion of the data is assigned to each render-node. The grid cell size is equal to the whole scene extents divided by the number of render-nodes. Each raw data slice is sent to every render-node that overlaps it. Empty cells are marked in the grid as data is read and cropped.

5.4.1 Test Environment

Cost parameters used for the DP partitioning are based on using the ServerNetII and HP Sepia-2a card for image communication and composition, and the VolumePro 1000 or GeforceFX5800 for rendering. On the MDS cluster, there is a two-tier cost function for network transfer. For each frame, the first image transfer from each node occurs through a DVI output across the SeverNetII, and inter-node composition time is $X1$. All other transfers are done over the Ethernet and require a framebuffer readback, with inter-node composition time $X2$. If p_r, the number of cells in partition p, is more than the number of render-nodes, N, in the system, then the inter-node composition cost for p is $X1 * (N - 1) + X2 * (p_r - N)$.

The Visible Korean Male data includes $8,590$ digitally captured photographic anatomic images of serially sectioned planes with photographic image resolution of $2,468 \times 1,407$ and 24 bits color, with a slice-interval of 0.2mm. The data set is accompanied by a corresponding 40GB set of 8 bit color *mask* slices with several regions of interest marked with different colors, for a total of 120GB. Mask slices are used to index a transfer function at run time. We have rendered several segmented regions of data from the Visible Korean Human Project on our cluster. The goal of segmentation here is to segment out empty space for the sake of illustrating the load balancing problem on a scene with an uneven distribution of empty space. It does not preclude changing coloring and translucency using a transfer function during rendering. The cerebellum and brain stem are shown in Figure 5.4. They

(a) Cerebellum



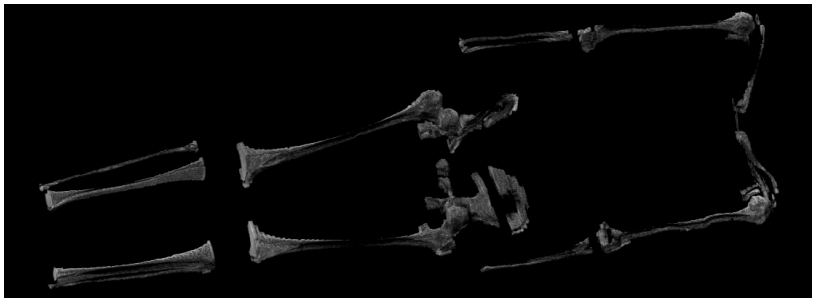
(b) Brain Stem

Figure 5.4: GPU-rendered images of segmented regions from the Visible Korean data set: (a) Cerebellum, (b) Brain Stem.

have been rendered on a GPU using a gradient mask volume and pre-segmented texture slices. The Visible Korean Male bones, cerebrum and lungs, rendered on VolumePro1000 hardware, are shown in Figure 5.5.

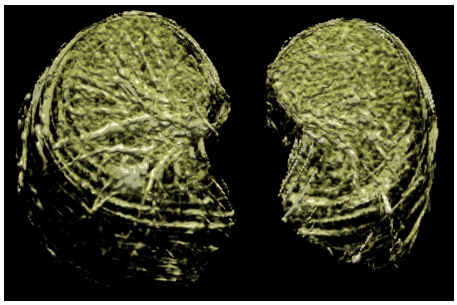
5.4.2 Timing

The partitioning times for traditional kd-tree, slab-projected kd-tree, and DP partitioning are compared in Table 5.1. We also ran experiments where cropped data was distributed with an octree subdivision, however the maximum data rendered is not generally reduced by using this strategy. The slab-projected kd-tree is a better choice because it allows the partition planes to adapt to the data, and avoids additional overhead in managing multiple



(a) Bones

(b) Cerebrum



(c) Lungs

Figure 5.5: VolumePro1000-rendered images of the Visible Korean data set: (a) Bones, (b) Cerebrum, (c) Lungs.

subvolumes assigned to a render-node. The slab-projected kd-tree uses an order of magnitude less preprocess time on average that the traditional one that requires multiple slice reads. The brick grouping DP partitioning preprocessing does not grow significantly as long as the size of the maximum number of bricks per render-node assignment is constant. Preprocessing for the grid partitioning is completed in the slice preprocessing phase, so is not included in this table. Grid preprocessing is significantly simpler and faster. However, for interactive rendering the improvement in rendering speed achieved by our partitions is more important.

Table 5.1: Partition preprocessing time (sec) for kd-tree and slab-projected kd-tree partitioning, and brick grouping DP (including bricking) for the Visible Korean bones on clusters with 8, 16, 32 and 64 nodes.

Cluster nodes	8	16	32	64
Kd-tree	792	1697	3508	7129
Slab-projected kd-tree	213	215	218	221
Brick grouping	251	252	253	255

Table 5.2: Empty space (percent) and slice preprocessing time (sec) on eight-nodes for the Visible Korean data set.

	Cerebellum	Brain Stem	Bones	Cerebrum	Lungs
Data Size (GB)	3.61	2.1	117.43	8.7	22.61
Empty space (%)	85	90	81	89	82
Slice preprocessing	4.7	0.45	109.2	8.01	21.03
Parallel slice preprocessing	0.8	0.34	18.4	1.36	3.54

A comparison of parallel and single node slice preprocessing, on an eight-node cluster, is shown in Table 5.2. The time required to segment, crop and distribute each data slice is the same for DP and slab-projected kd-tree partitioning and brick grouping DP algorithms. The time for data distribution is higher for the grid partition because it is done prior to cropping. However the slice preprocessing is done in parallel, and the additional overhead for marking grid cells during this process is negligible. Rendering times for all except the bones data were interactive (32 frames per second (fps)). The cropped bone data did not fit when the grid partitioning was used, and rendering was unsuccessful after several minutes. For these cases, a large amount of data is loaded into the VolumePro1000 memory during rendering, causing the system to hang up. Using a fine granularity local grid reduces rendering time on some nodes. However, it doesn't improve the end-to-end render time because it is bound by the slowest node, which contains no empty cells even within the finer granularity. For the DP partitioning and slab-projected kd-tree partitioning and brick grouping DP algorithms, the data fit into the total memory of the VolumePro1000 cards on our cluster. The average frame rate for the bones data rendered on 8 nodes was 1.35fps with slab-projected kd-tree partitioning and 2.6fps using DP partitioning.

5.4.3 Load Balancing Results

Data cropping is only useful in reducing end-to-end rendering time if it results in reducing the maximum data rendered on any node. Figure 5.6 shows the maximum data assigned per node using our algorithms compared with the grid partition. The largest decrease in data rendered compared with a grid partition is due to cropping because the maximum data

rendered with the grid is the same as the full portion of data assigned prior to cropping, with some nodes rendering no data. This is due to the high grid granularity required on the intra-node distribution level. All of the data sets show large reductions for both the slab-projected kd-tree partitioning and brick grouping algorithm and DP partitions compared with the grid partition. The smaller maximum data per node for DP partition is strictly due to using a cost function in incrementally finding the partition. Load balancing improvements over the slab-projected kd-tree partitioning and brick grouping algorithm partition is most significant for data sets that have large variations in empty space, which demonstrates the feasibility of using a cost-driven approach to reduce rendering time using DP.

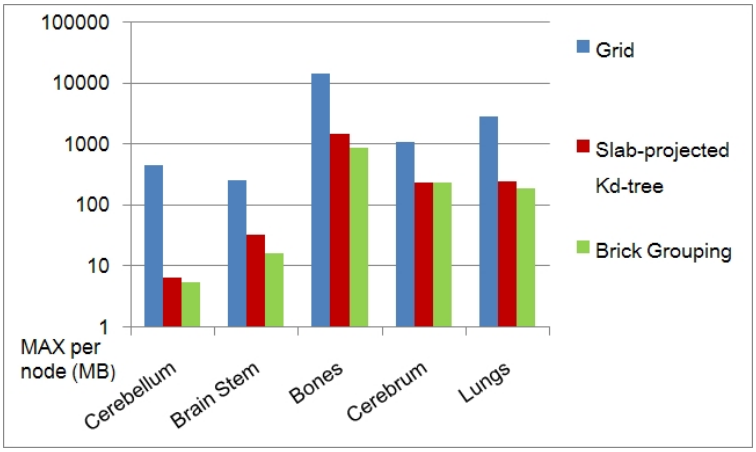


Figure 5.6: Data distribution load balancing. Maximum data assigned to any node for an eight-node cluster (in log scale).

Partitioning using DP was similar, and in some cases identical to, the slab-projected kd-tree partitioning and brick grouping algorithm partition for the cerebellum, lungs, brain stem, and cerebrum. This is because most of the empty space is cropped prior to data distribution. The speedup of DP compared with slab-projected kd-tree partitioning for the bones is probably due to the fact that we restricted the depth of the kd-tree to force the number of leaf nodes to be equal to the number of render-nodes to avoid interleaving of images between render-nodes. The speed up due to empty space comes from the fact that data is distributed in a manner that only requires a single image to be sent from any render-node. This is the result of the cost function which assigns a high penalty for data transfers.

In order for an octree to meet this criterion the data must be statically distributed prior to cropping, resulting in an uneven distribution of empty space. In a system where the transfer function cost is not dominant, multiple image compositions/transfers per node may be part of the DP solution. The load balancing was most improved for the bones because there is a wider variation in empty space distribution for the segmented bones in the original data set. For these cases, it is clearly advantageous to preprocess the data using one of the algorithms presented. When there is a large variation of empty space in the data set, static parallel partitioning results in poor load balancing. The grid assigns zero data to some nodes for several of the data sets because of the high grid granularity required on the intra-node distribution level.

The scalability of load balancing is illustrated in Figure 5.7 for the bones on different sized clusters. While the grid partition improves with a larger cluster because of the finer grid granularity, the number of render-nodes that remain idle also increases. The amount of data assigned per node does not reach our more adaptive techniques, and it is expected that this gap will persist for any system size.

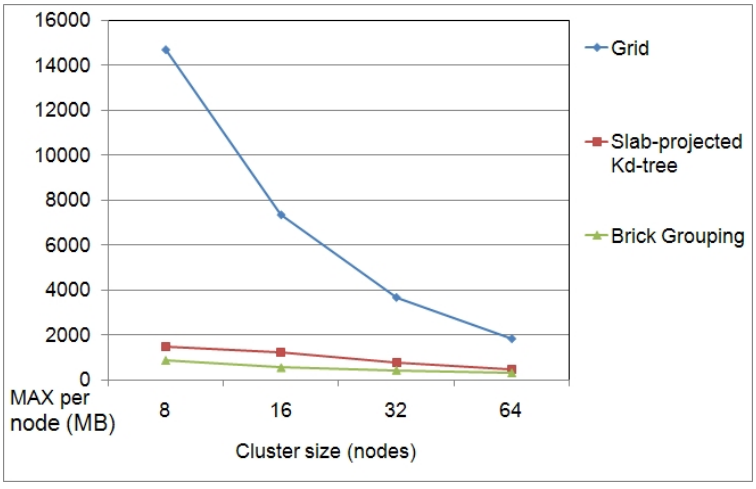


Figure 5.7: Load balancing scalability. Maximum data assigned to any render-node in 8, 16, 32 and 64 node clusters.

5.5 Summary

Managing massive data sets is a fundamental requirement for distributed volume rendering. We have introduced practical methods for reducing data early in the preprocessing pipeline and for out-of-core data distribution. Scalable priority-constrained data distribution is an open area of research. We have introduced a solution using DP, which allows a cost function to drive the distribution process, and a kd-tree solution. Our slab-projected kd-tree partitioning and brick grouping algorithms each find a good partition with a moderate preprocessing time using a series of slab-projection slices. As demonstrated by our test results, our solution allows the scene partition to be controlled so that data is distributed more evenly between render-nodes. The portion of the scene assigned to each render-node is dictated by the distribution of non-empty regions, resulting in better load balancing than with traditional partitioning methods.

In our slab-projected kd-tree partitioning and brick grouping algorithm and DP partitions, natural boundaries of non-empty regions are cropped prior to data distribution, resulting in a smaller maximum data assigned per node compared with a grid partition with the same number of cells. In our experiments, scenes with large portions of non-uniformly distributed empty space show more dramatic improvements using our approach than those with more homogeneous data distributions.

Slab-projected kd-tree partitioning has some advantages over the DP model. For instance, it is more suitability for perspective projection. The main drawback of the DP is that a solution for each octant is required, and some replication of bricks is necessary. The other drawback is that it is intractable if the number of decision branches at any stage is not controlled by the average number of neighbor bricks per brick. However, this algorithm demonstrates the potential to reduce rendering time using a cost-driven DP approach.

A promising area of future work is dynamic load balancing. Interactive transfer function updates are possible with our method. Recent developments in dynamic scanner technologies produce time varying data scans which are very well suited for these render clusters due to their tremendous size. Further research is needed for adapting the partition found

during our slab-projected kd-tree partitioning or DP preprocessing. One solution would use an incremental update approach to move partitions as regions of non-empty voxels move throughout the scene.

Chapter 6

Moving Walls DP Data Distribution

In this chapter we introduce the novel *moving walls* [30] dynamic programming (DP) solution to LBND, which effectively merges our out-of-core bricking with DP for a solution to a relaxed version of the LBND problem. In Section 6.3 we present a description of our moving walls DP algorithm. Implementation and experimental results are presented in Section 6.4. We compare our moving walls algorithm with partitioning using a more straightforward kd-tree partitioning as well as our brick grouping DP solution [31].

6.1 Overview

The input to our algorithm includes a scene with volumetric data, a description of the distributed system configuration, including the number of render-nodes, rendering and local composition costs, network transfer costs, and the memory capacity of each render-node. Without loss of generality, we label the longest axis z, and the shortest axis y, and use $x \times y$ slices. The output is a render-node assignment, which minimizes the total runtime cost, does not violate any physical resource constraint of the system, and observes the precedence order for image composition.

Moving walls is a DP algorithm that finds the optimal solution of a constrained version of LBND. This solution allows cost function evaluation to influence the data distribution,

but greatly reduces the potential size of the problem by imposing restrictions on the cut plane ordering. In addition, these restrictions allow a deterministic view-dependency order to be derived from the resulting partition for any viewpoint, whereas brick grouping DP solves the problem for a single viewpoint. The flex-block tree is derived directly from the flex-block partition to determine the image composition order.

A cost function is derived from a description of the distributed system configuration, which includes the number of render-nodes, rendering and local composition costs, network transfer costs, and the memory capacity of each render-node. For a given stage, each feasible solution divides the scene data from the current stage to the end of the scene. The optimal partition for each stage is independent of all partitions prior to that stage. Stages are processed starting with the last stage and proceeding to the first.

6.2 Cost Function

The solution to our DP problem is:

Minimize the objective function, c_0^*, while satisfying the following constraints:

1. The final partition, p_0^*, assigns each brick to exactly one render-node.

2. An image composition precedence order is well defined for all viewpoints;

where p_i^* is optimal partition for stage i, and c_i^* is the cost of the optimal solution for stage i.

For convenience, we list algorithm definitions here:

$b_{p,k}$: number of bricks in cell $\gamma_{p,k}$

M: average number of bricks per render-node

We use a cost metric to choose the optimum partition for each stage. The cost function is tailored to the specific system configuration. For this example, the cost function is for hardware ray cast rendering of blocks with relatively similar surface areas. For ray tracing, other factors such as the surface area of the blocks could be included in the cost function.

The rendering time cost per brick is R. Image compositing requires that one or more of the contributing images be loaded into composite hardware memory; compositing is scheduled so that the image in memory at the end of a rendering pass is one of the contributing images. The cost of image composition depends on whether any contributing image is sent over the network or not. The cost of each local composition, L, includes the time required for loading any contributing image which is not resident in memory at the time of composition. Network communication time, N, is the cost of sending an image across the network. The rendering hardware memory capacity constraint, C, is used to determine the brick sizes at input. The input to our problem includes a volumetric data set and the following set of rendering system parameters:

n: number of render-nodes

R: direct rendering cost per pass

L: local composition cost

N: cost of each inter-node composition

C: rendering hardware memory capacity

Each feasible solution p, for period i, is defined by a partition of the slab of data from stages from i to n. The cost of this solution is defined in terms of the number of render-nodes assigned and the number of bricks assigned to each render-node.

Parameters associated with partition p are:

r_p: the number of render-nodes required by solution p

b_{tp}: the number of bricks in render-node t for solution p

If the number of render-nodes for solution p is no more than the number of render-nodes in the system, then the inter-node composition cost for p is $N * r$. The rendering cost is $R * max(b_{tp})$, where t is any render-node in p. The image composition cost is $\sum_{t=1}^{r}(L * (b_{tp} - 1))$. The total cost of p is:

$$c_p = N * (r - 1) + max(R * b_{tp}) + \sum_{k=1}^{r}(L * (b_{tp} - 1)) \tag{6.1}$$

6.3 Moving Walls Algorithm

Our algorithm requires two sweeps through the data. In the first pass, original data slices are read and each slice is segmented to clear empty space regions, then cropped and written back to new file. The algorithm sweeps along the longest axis in the data and determines the best set of partitions along this axis. A stage corresponds to a consecutive group of slices, or slab of data. The number of slices in a slab is calculated from the given number of stages. In the second pass, the pre-cropped data slices are read, and the partition extents are obtained through a traversal through the optimal path.

The cost of a z-partition is determined by summing the cost of each slab in the partition. We further reduce the solution space by restricting plane cuts within a slab to occur on the longest slice axis, or the x-axis in our case. The cost of a slab is the cost of the optimum x-partition within that slab. For clarity, we refer to the partition along the z-axis as z-partitions and those along the x-axis as x-partitions.

A slab-projection slice is the accumulation of all slices between the first slice and the last slice of that slab. Each x-partition is found using DP on the corresponding slab projection slice, where stages are columns of data. To distinguish these stages from those along the z-axis, we refer to these as x-stages. The smallest group of bricks within a column is found by cropping out empty space and slicing off regions close in size to the target brick size.

In order to follow conventions of DP, stages are processed in reverse order. Thus, slices are read in reverse order as well. For each stage i, feasible solutions include a z-cut at each stage j, between i and the scene end. The slab with all data from the start of stage i to the end of stage j is $slab_{ij}$. The cost of the solution which cuts the scene at stage j is the optimal cost of $slab_{ij}$, or c^*_{ij}plus the optimal cost of stage $j+1$, or c^*_{j+1}. $SlabPartition$ determines the cost of the optimal partition for all slabs between i and j. Dynamic programming is used to determine the optimum partition for each stage. This is calculated using DP along x-stages on the group of slab projection slices, which corresponds to these slabs.

A summary of the moving walls algorithm follows:

For each stage i:

Read all slices in i and create slab projection slice p_i

For each stage j from i scene end:

$c^*_{ij} = SlabPartition(i, j)$

if $c^*_{ij} + c^*_{j+1}$ is optimal for i

Store partition and cost parameters

6.3.1 Building the Flex-block Tree

The output of the moving walls algorithm defines a partition of flex-blocks in terms of the cut planes. However, in order for this partition to be useful for ray traversal and image composition order, the flex-block tree must be constructed. The procedure for constructing the tree is straightforward. The solution to the moving walls is a series of z-cut planes. The top of the flex-block tree is defined by the center one of these cuts. The left and right sides of the cut are recursively added to the tree until all z-cuts are included in the tree. Each leaf node in this tree at this point contains a slab of the scene. Within each of these slabs, the optimal partition is traversed and added to the tree in the same way as the z-cut planes.

6.3.2 Efficiency Considerations

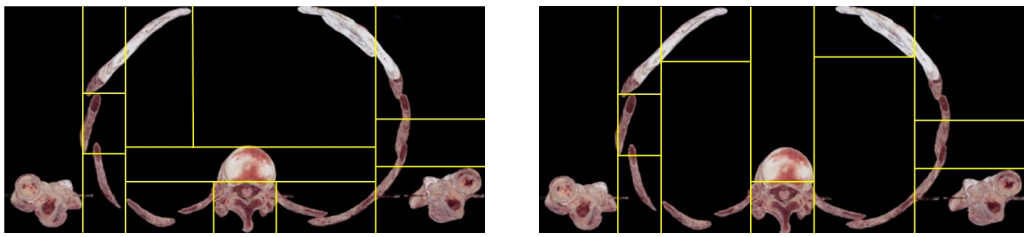
Since segmented and cropped slices are saved during the first pass of the algorithm, they are available in disk memory for subsequent executions of the algorithm. In this case these are used for the first pass as well as the second one in order to reduce the total data processed. Preprocessing could also be done in parallel across the cluster.

The number of stages used determines the granularity of the cost function analysis, and therefore the accuracy of the solution. At the limit, each slice of raw data represents a stage. We evaluated different sized slabs in order to determine a good trade-off between the algorithm run-time and accuracy. Moving walls preprocessing could also be done in

parallel by the render-nodes. For applications used in our experiments, the full data set did not fit in the hard drive of each render-node, so this would have required overhead of moving slices between nodes before and after preprocessing. In addition, cropping reduced data slice sizes by an average of 30%, reducing the inter-node communication required for initial data distribution.

6.3.3 Comparison to Brick Grouping

Our brick grouping DP algorithm for LBND evaluates a cost function to create a load balanced network distribution. This solution is based on a DP solution for the multipass partition problem (MPP) introduced by Heirich [41]. MPP is a problem that arises when a computation targeted for the GPU exceeds resource constraints. The computation is divided into multiple passes that do not violate resource constraints, in order to minimize total computing time. The depth of the search tree is limited to prevent the algorithm from becoming intractable, which limits the size of the data bricks with respect to the whole data set. With brick grouping DP, out-of-core bricking is done in conjunction with data partitioning. In addition, the partition is optimized with respect to a single viewpoint. We compare a typical decision stage for each algorithm in Figure 6.1.



(a) Brick grouping (b) Moving walls

Figure 6.1: Comparison of partitions with different algorithms. (a) Brick grouping DP considers partitions which alternate between cut planes; (b) Moving walls eliminates recursion by using parallel cut planes.

Since both algorithms attempt to minimize the same cost function, the solutions tend to converge. In Figure 6.1(a), the cost of every combination of cut planes for every stage is calculated, and recursion is needed. The problem size is reduced by simplifying the cuts,

as shown in Figure 6.1(b). The flex-blocks in each partition are the same in this case. This illustrates the reason the moving walls algorithm produces partitions which are comparable to those derived from the brick grouping DP algorithm. However, the partitions may differ because the brick grouping version compares more partitions overall.

6.4 Results

We compare our flex-block partition with both the brick grouping DP partition and a grid data distribution combined with a local kd-tree partition on each render-node on the Stony Brook Visual Computing Cluster. Several massive volumetric data sets have been volume rendered on the Stony Brook Visual Computing Cluster as part of this research. We tested our data distribution preprocessing for larger clusters to demonstrate scalability. We used the MDS cluster and VolumePro 1000 with 1GB memory for the volume rendered images.

The very high resolution teeth and fossil data sets consist of micro CT scanned images of 2048 by 2048 pixels, each with a slice thickness of around 10 microns. The 6.7GB Homo Sapiens tooth contains 1589 slices, the 2.8GB data set contains 663 slices, and the 8.5GB diademodon fossil data set contains 2028 slices, and the 6.6GB baboon fossil data set contains 1570 slices.

The Visible Korean male data [87] includes $8,590$ digitally captured photographic anatomic images of serially sectioned surfaces with photographic image resolution $2,468 \times 1,407$ and 24 bits color, for a total of 120GB. It is accompanied by a corresponding 40GB set of 8-bit color segmented masks.

The full Visible Male data set, from the Visible Human Project [106], is a sequence of axial anatomical images of 2048 by 1216 pixels at 1 mm slices. The Visible Male color data set has 1879 slices. With the addition of an alpha channel the data set is 18.7GB.

We use out-of-core region growing to create mask volumes for all data sets used in our experiments except for the Visible Korean. We take advantage of these slices and skip

the region growing part; each slice is segmented to the selected region using its associated mask slice. The bones from the Visible Korean data set are rendered as an application with a large portion of empty space. Alternatively, the segmented slices could be used to derive a transfer function to segment the bones during rendering.

We compare to a kd-tree built locally on each render-node after the data has been split at the top level with a grid partition. Prior to the top level splitting the data is reduced initially by cropping empty space that is exterior to all non-empty voxels. A traditional kd-tree could be used to distribute data between nodes as well. However, if the entire data set does not fit in main memory or cache (typically the case for massive data sets), it must be re-read from disc memory for each splitting plane. By partitioning the data prior to This is done in parallel on the render-nodes because the bulk of empty space cropping occurs in the inter-node partitioning step. Frame rates range from 6Hz to 20Hz using flex-blocks.

A comparison of the rendering times for the grid with kd-tree and the flex-block partitions, on an eight-node MDS cluster with ray cast rendering done on VolumePro 1000 boards, is listed in Table 6.1. The flex-block partitions generated by the brick grouping DP and moving wall algorithms vary slightly, as shown in the table. The jump in time for the Visible Male without using flex-blocks, as shown in Table 6.1, is explained by the fact that when assigned data exceeds the on-board memory capacity, local rendering stalls to load new data bricks. Frame rates achieved using the grid with kd-tree partition demonstrate inferior load-balancing compared with the flex-block partition.

Table 6.1: Rendering time (sec) on a ray casting hardware cluster using a grid data distribution with local kd-tree partitions, compared with flex-block partitions (generated by the brick grouping DP and moving walls algorithms). Data includes segmented Visible Korean Bones, the Visible Human Male, and the Teeth and Fossils.

	Korean Bones	Visible Male	Cebus Apella	Diademodon Fossil	Baboon Fossil	Homo Sapiens
Grid with kd-tree	13.1	13.1	2.7	8.5	6.6	6.7
Brick grouping	11.27	0.72	0.11	0.24	0.18	0.19
Moving walls	11.44	0.77	0.11	0.24	0.18	0.19

The partitions found using moving walls in our experiments were the same or close to

those found using our brick grouping DP algorithm. However, moving walls algorithm provides a partition that is view-independent, whereas brick grouping finds a view-dependent solution to LBND.The preprocessing time is comparable for both algorithms, but for brick grouping it depends on the data being pre-cut into bricks that are sized to control the algorithm branching. Moving walls operates directly on the slab-projection slices.

In Figure 6.2 we show several slab-projection slices with local partitions for a 64 -node distribution of bones using our moving walls. The memory read, composition and network transfer times used in the cost function were derived from prior rendering experiments on our MDS cluster. We used a range of brick sizes; further reduction in data size is achieved with smaller bricks that can closely crop the data, but at the expense of overhead in managing and loading bricks during rendering. The following parameters were used in both moving walls and brick grouping DP:

n: 8, 16, 32 and 64 render-nodes
$R = 0.001$ s
$L = 0.0002$ s
$N = 0.0005$ s
$C = 10$ bricks

In order to avoid floating point calculations, integers representing the relative values of these costs were used in both of the flex-block partitioning programs. We compared the algorithm run-time and maximum data size rendered per block for grid with kd-tree subdivision with both brick grouping DP and the moving walls algorithm. The preprocessing times are shown in Figure 6.3. The slowest running was the brick grouping DP algorithm. The dominating factor for both of these is the slice read-time. The difference in preprocessing time is due to the following slice-skipping efficiency used for the grid with kd-tree.

In Figure 6.4, we show load balancing results using flex-block DP partitions compared with grid distribution and local kd-tree partitions. The teeth and fossil data did not show

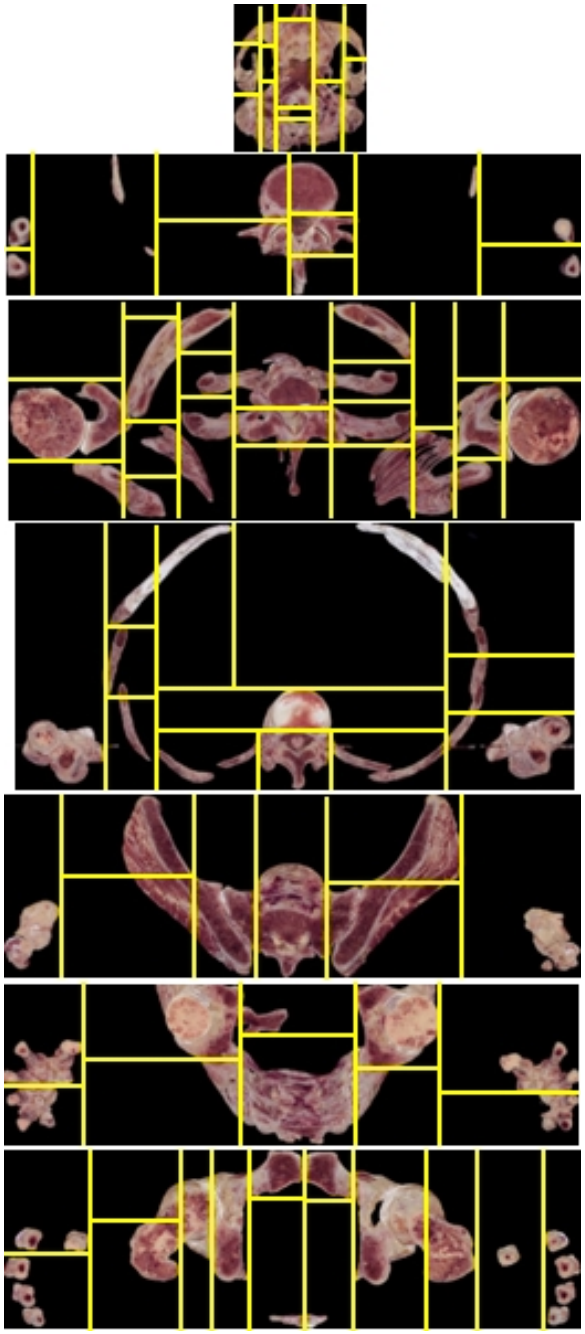


Figure 6.2: Sample partition slabs of the Visible Korean bone data partition for a 64 node cluster.

significant improvement using a flex-block partition compared with an grid with kd-tree, and are not included in the scalability results. The moving walls partitions have comparable data distributions to those found using brick grouping DP, with a reduced preprocessing

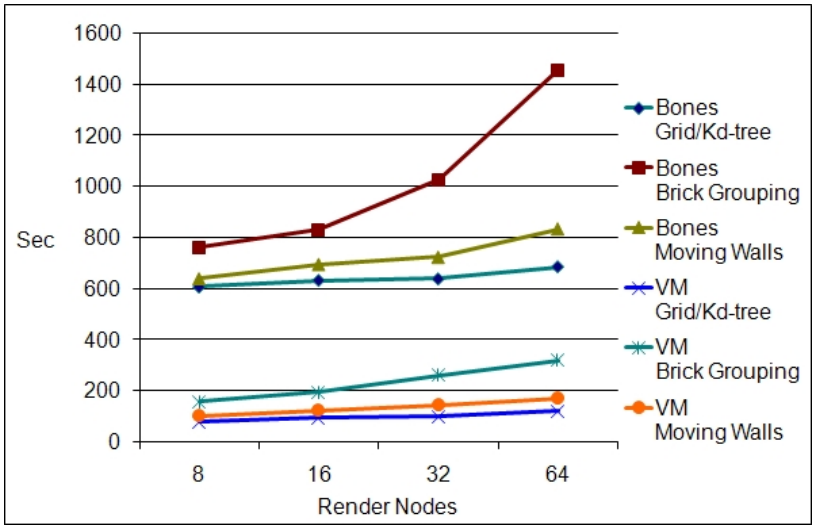


Figure 6.3: Preprocessing time (sec) for flex-block data distribution of the Visible Human (VM) and the Visible Korean bones (Bones) as a function of the number of render-nodes (in log scale).

time. The load balancing is measured at the maximum data assigned to any render-node.

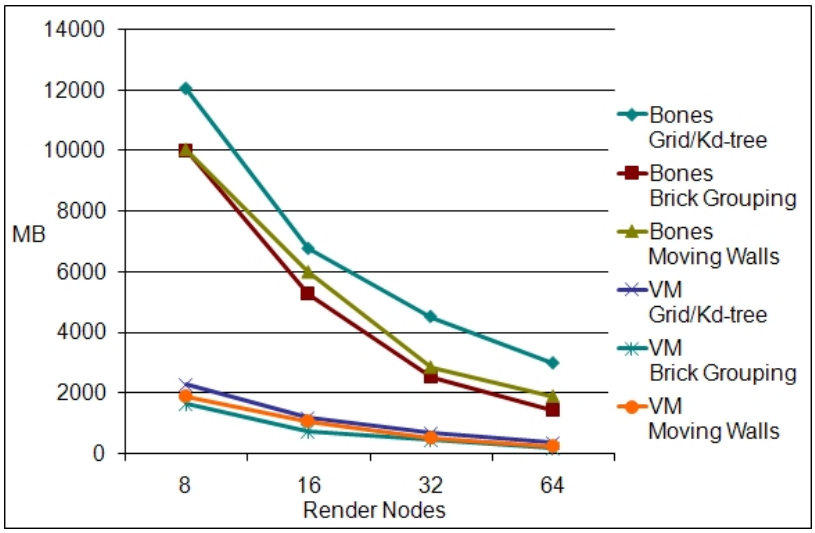
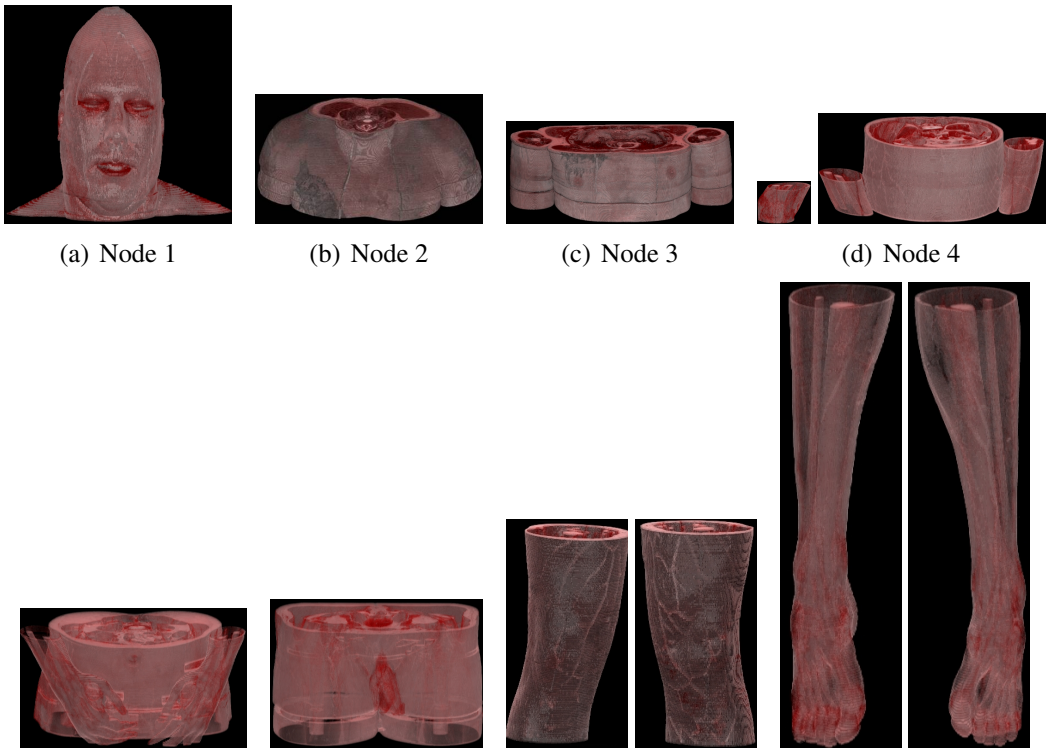


Figure 6.4: Load balancing and scalability results. Maximum volume data (MB) assigned to any node for the Visible Male (VM) and Visible Korean bones (Bones), using flex-block partition compared with grid with kd-tree partition on 4, 8, 32, and 64 nodes (in log scale).

The distribution of this data set on an eight-node cluster is shown in Figure 6.5. The separate arm piece in Figure 6.5(d) demonstrates that multiple data blocks are rendered by

the same render-node. By using DP to create our space partition, we are able to force render assignments to be more even. Although the grid with kd-tree partition is used to reduce data within each render-node, the size of data assigned to each node varies significantly. The maximum data size per node constraint, in conjunction with the cost function, results in even load balancing.



(a) Node 1 (b) Node 2 (c) Node 3 (d) Node 4

(e) Node 5 (f) Node 6 (g) Node 7 (h) Node 8

Figure 6.5: Visible Male distribution on eight rendering nodes.

6.5 Summary

We presented a framework for distributed parallel visualization of massive volumes. Our algorithms have been designed to be independent of the operating system or hardware of the cluster. Our novel dependency graph approach to load balancing allows flexible and efficient render-node assignments, and is general enough to be utilized in any distributed system.

We have introduced the flex-block partition, and corresponding flex-block tree. Flex-blocks give more control over allocation of volumetric data between render-nodes. The flex-block partition reduces empty space, facilitates early data reduction, and provides a deterministic depth order of flex-blocks for any given view direction. Our cell-tree automates ray dependency encoding for distributed volumetric ray tracing, and cell-tree peeling is used for ray-task scheduling. The flex-block partition can be used along with our cell-tree, to improve ray traversal and task scheduling for ray tracing. Image composition dependency order is pre-computed in the flex-block tree. Flex-blocks facilitate reducing the size of data early in the pipeline, thereby reducing network traffic and the run-time of later preprocessing steps.

We have also introduced a view-independent solution to LBND, the moving walls algorithm. Our moving walls algorithm uses the slab-projection slice to enable out-of-core processing of massive volume data. By using the dependency graph of tightly cropped bricks for large-scale data management, block-level space leaping effectively occurs prior to data distribution. This is true for both the ray casting and ray tracing algorithms. Load balancing data distribution based on empty space is only applicable for scenes in which a portion of data will never be rendered. The classification of empty space is defined by the user. In our experiments we have considered two situations: one, in which region growing is used to segment the object of interest, and the other, in which a set of mask slices have been provided. Although we illustrate our algorithm using bones from the Visible Korean, the same algorithm would be applicable if we chose to include all regions that are part of the actual body.

The moving walls DP algorithm has several advantages. Our method optimizes for resource allocation between render-nodes. It allows blocks to expand or contract for good load balance in systems with varying configurations. The primary advantage of our algorithm is that local processing leads to a scalable system. Different system configurations can be modeled by reformulating the objective function. Different viewing priorities are accommodated by changing the cost function.

Our z-cut stages take advantage of the somewhat regular nature of the human body. In particular, the slabs of data do not generally vary too much over long spans of slices. As a future work, we plan to run additional experiments on less regular data to determine the impact on the template approach. Future work also includes extending our algorithm for limited data replication. This would have the same effect as the RDS solution using the possibility of recomputing multiply referenced nodes. A parallel version of the moving walls algorithm in which each render-node partitions an equal portion of non-empty voxels is also a possible area of future research. The moving walls algorithm could also be extended to include ray densities in the cost function for dynamic load balancing.

Chapter 7

Cell-tree Scheduling for Ray Tracing

In this chapter we present our cell-tree [28], which concisely describes ray dependencies in a distributed parallel ray tracing environment. We also present our cell-tree peeling ray-task scheduling algorithm, which seeks to minimize the number of memory fetches in a distributed memory ray trace engine, or ray tracer. The term *memory* is defined to mean local memory with respect to the ray tracer. The ray tracer could be implemented in software, a special purpose ray tracing architecture, or in a GPU. We compare two algorithms for scheduling of cells to be processed. The first one is a greedy algorithm called *max-work*, and the second one is our cell-tree peeling algorithm. In our experiments, using the cell-tree algorithm reduced the number of memory fetches compared with using the max-work algorithm.

This chapter is organized as follows. We present our cell-tree ray dependency encryption algorithm in Section 7.2, and our cell-tree peeling algorithm in Section 7.3. In Section 7.4 we discuss our architecture simulation, and in Section 7.5 we summarize our results. We conclude in Section 7.6 with a discussion of potential research areas for cell-tree.

7.1 Ray Traversal

For distributed volumetric ray tracing, we divide the scene into cells. The ray tracer can operate on one cell at a time. When a ray is cast through the scene, the first cell it intersects with is detected. The ray is then placed on a queue of pending rays which is maintained for that cell. The ray tracer incurs a memory fetch cost each time it switches between cells; the aim is to minimize the number of these memory fetches. The basic assumption is that the memory fetch cost is very high compared to the ray contribution or splatting computation plus ray spawning and traversal costs. Our cell-tree peeling algorithm can be used for BRDF splatting and for ray traced rendering. We present it from the point of view of a renderer for clarity.

When the ray tracing process begins, one or more eye rays are shot into the scene for each pixel in the image plane. *Eye* rays are the rays which originate from the image space. Each ray accumulates opacity until either it reaches a predefined maximum opacity, it has reached a predefined maximum number of bounces, or it exits the volume. As a ray intersects the volume it can spawn additional reflection and/or refraction rays. A reflection ray is generated when a ray reflects off a surface, a refraction ray is generated when a ray is spawned through a translucent surface, and a shadow ray is shot from each point of intersection toward each light source to determine if the object is blocked from the light at that point. Every ray, with the exception of eye rays, is dependent on its parent ray as well as on all of its predecessor rays. As with image composition of ray casting algorithms, there is a strict relationship between ray segments defined by the compositing equation. It is not possible to spawn secondary rays, or calculate the contribution of scene cells to different ray segments, until we find the intersection positions of the rays that cause them to be spawned.

Each ray has an ordered sequence of cells which it depends on. We call this its ray-cell dependencies, or ray dependencies for short. However, the same ray may traverse through a cell more than once, resulting in cyclic ray dependencies, as illustrated in Figure 7.1(a). In ray tracing, a ray segment is the intersection of a ray with a scene cell. Note that spawned

rays do not define a new dependency until they exit the cell they are spawned from. For the remainder of this section, we use the word ray instead of ray segment for improved readability.

7.1.1 Ray Queues

Eye rays which intersect the volume are placed in the queue of the intersected scene cell at the start of ray tracing. Once a cell is in memory, all of the rays on the queue for that cell are processed. This includes any additional rays which are generated while processing the rays from that cell if they intersect the volume in the same cell which intersect the same cell.

When a ray exits the cell, it can either exit the volume or enter an adjoining cell. When a (non-shadow) ray exits the scene its contribution is added to the image buffer. Shadow rays which exit the scene without hitting an object in the scene have no effect, which those that do diminish or eliminate the contribution of rays which originated from the same intersection point. When a ray travels from one cell to another, it is placed on the queue for that cell. As long as a cell of data is in local memory, all rays on that cell's queue are processed before it is replaced with another cell. If a ray is reflected back into a cell it has previously exited, then the same cell is read from memory again. The order in which rays are processed influences the number of times each cell is cached from memory. Our goal is to minimize this number.

7.1.2 Problem Definition

The problem can be stated formally as follows: We define a job to be a group of rays requiring processing. For every job j there is an associated cell, c_j. If cell c_j is not available in local memory at the start of job j, then there is a cost of p for fetching it from a higher memory level such as main memory. Given a partial order of jobs, find the schedule that obeys the partial order and has the lowest overall cost. The priority order must be preserved

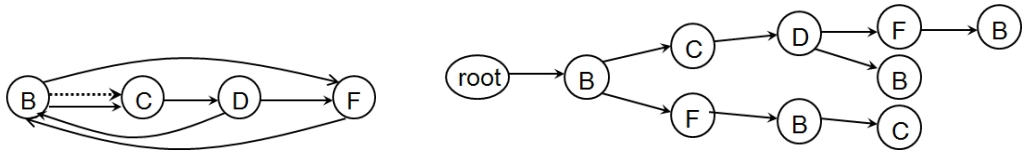
in parallelized ray tracing applications. There is not generally a scheduling order which would allow each cell to be read from memory only once. One heuristic approach is to choose the cell with the most rays pending, or the *max-work* algorithm.

7.2 Cell-tree

The set of ray segments spawned from an initial ray defines a tree of ray-cell dependencies. For a 512×512 image, there are over $250,000$ ray trees, each with its own set of dependencies. If ray r_2 is spawned from ray r_1, it is defined to be a child of r_1. The ray-cell dependencies of ray r_2 are the same as those of r_1, with the addition of any new cell r_2 enters. For each primary, or eye, ray, all of the dependencies spawned from that ray form a ray tree, as shown in Figure 7.1(b). If ray s is spawned from ray r, s is defined to be a child of ray r. The ray dependencies of ray s are the same as ray r with the addition of any new cell ray s enters.

For a 512^2 image, there are over $250,000$ ray-trees, each with its own set of dependencies. The large number of ray-trees, along with the cyclic ray dependencies inherent in distributed ray tracing, has discouraged researchers from investigating ray dependency graph scheduling algorithms. However, if the time required to fetch the cell is significant with respect to the processing time, then the order in which the cells are processed greatly affects the frame rate. Since memory fetches can be orders of magnitude higher than processing speed, we would like to find a cell processing schedule based on the behavior of rays in one frame that can be used in the next frame to reduce the total number of cell fetches.

Our cell-tree [28] gathers these trees into a single, compact description of all ray dependencies. The dependencies of clusters of eye, shadow, reflected and refracted rays are gathered into a compact description. In our experiments, cell-trees were several orders of magnitude smaller than the number of dependencies represented. We use the cell-tree to determine a cache scheduling policy for the subsequent frame.



(a) Cyclic dependencies (b) Ray-cell dependency tree

Figure 7.1: Ray-cell dependencies. (a) Ray tasks traverse between the cells. Ray tracing has pseudo-random ray traversal between data blocks resulting in cyclic dependencies which require some cells to be cached more than once; (b) A ray-cell dependency tree shows these dependencies, but there is a unique tree for each initial ray.

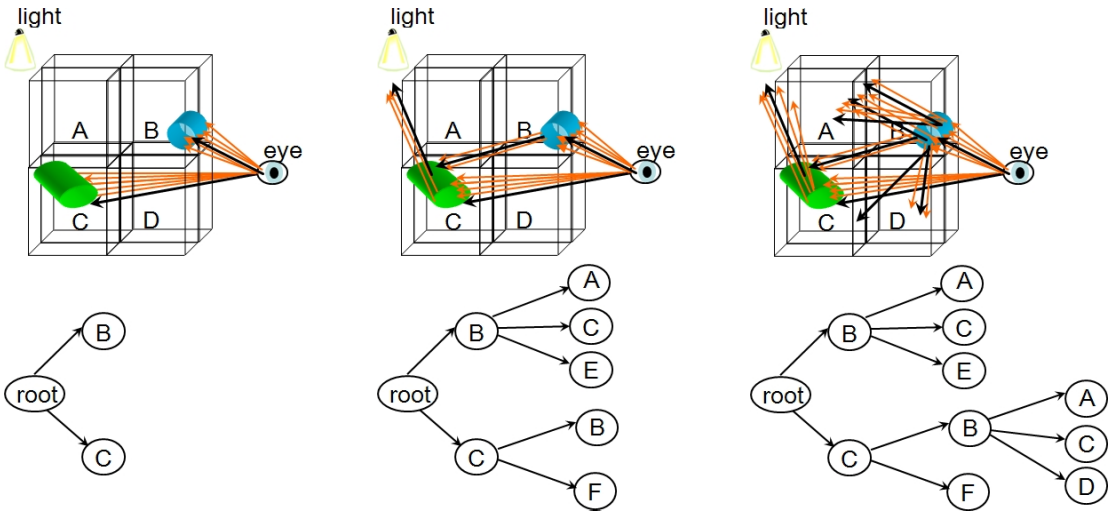
7.2.1 Cell-tree Construction

We gather ray-cell dependencies into coherent groups to create a single, consolidated, cell-tree. All rays which traverse from one cell to another at the same relative time are represented by a single link in the cell-tree, as shown in Figure 7.2. Each node in the cell tree has a $cellID$ and a unique $nodeID$. Several cell-tree nodes may have the same $cellID$. Each ray has a cell-tree $nodeID$ that indicates where the progress of that ray path, is being encoded in the cell-tree.

The cell-tree is initialized with a single node corresponding to each cell that is intersected with an eye ray. Cell-tree construction proceeds by adding new dependencies to the cell-tree as rays are spawned. When ray r traverses from cell i to cell j, the dependency is encoded by adding a node to the cell-tree as a child to the cell-tree node which corresponds to the r, or $r.nodeID$. If this dependency has already been encoded when another ray traversed along the same dependency path, then one of the child nodes of $r.nodeID$ will have a $cellID$ that matches that of .

This process is illustrated in Figure 7.2. The subtree in Figure 7.2(a) has two branches that represent ray dependencies from the viewpoint to cell B and cell C, and the rays are initial eye rays. In Figure 7.2(b), the new rays are reflection and shadow rays that are generated during ray tracing. Figure 7.2(c) illustrates secondary reflections. A separate list of cell-tree nodes is maintained for each cell for later use in the cell-tree peeling algorithm.

In order to have a dependency graph that represents all ray dependency trees, a group



(a) Primary rays (b) Reflection, refraction and shadow rays (c) Secondary reflections

Figure 7.2: Cell-tree construction. A ray dependency occurs when a cluster of rays, shown in orange, traverses from one cell to another. The first ray dependency of each cluster, shown in black, adds a branch to the ray tree. (a) Eye rays intersect volume in cells B and C; (b) Reflection, refraction and shadow rays intersect the volume in cell A, C and F (behind C) from cell B and cells B and G (behind B) from cell C; (c) Secondary reflection rays enter cells A, C and D from cell B.

of rays traversing along the same path is represented by a chain in the tree. This is accomplished by tagging each ray, r, with its corresponding place in the cell-tree, or $r.nodeID$. As long as a spawned ray remains in the same cell, that ray is tagged with the cell-tree node of its parent ray. When a spawned ray traverses from cell i to cell j, the cell-tree node corresponding to its parent p, $p.nodeID$ is examined. If $p.nodeID$ has a child node, c, with $c.cellID = j$, then the spawned ray dependency is represented by c, and the spawned ray is tagged with the $c.nodeID$ of the child node. Otherwise, a new node is added to the cell-tree.

Cell-tree construction consists of a simple piggy-back operation during ray tracing, and is not dependent on any particular scene partition structure. The dependency description of a ray, traversing from cell c_1 to cell c_2, is added to the cell-tree if it intersects the brick in c_2. A ray dependency is represented by two nodes in the cell-tree. For each node n in

the cell-tree, $n.cell$ is the corresponding scene cell. The ray dependency for all rays going from cell c_1 to cell c_2 at the same relative stage along the ray path is represented by parent node n_1 and child node n_2, or $c_1 \rightarrow c_2$, where $n_1.cell$ is c_1 and $n_2.cell$ is c_2. Eye ray dependencies have parent node $root$. Each ray is tagged with a cell-tree $nodeId$, which indicates the current stage in the ray's traversal. Cell-tree construction proceeds by adding new dependencies to the cell-tree as rays are spawned. If a spawned ray follows the same path as another ray, then no new entry is made in the cell-tree. More than one tree node has the same cell value due to the fact that dependent rays may enter the same cell more than once.

A formal description of cell-tree construction follows:

Initialization:

For each eye ray r with first intersection cell c

If $root$ has child node n_1 where $n_1.cell = c$

Tag r with n_1

Else insert new node n_2 as a child to $root$

Tag r with n_2

Ray Traversal:

For each ray r entering cell c

If $r.nodeID$ has child node n_1 where $n_1.cell = c$

Update ray count in n_1 and tag r with c

Else insert new node n_2 as child of $r.nodeID$

Tag r with n_2

7.2.2 Cache Savings Links

In order to take advantage of ray-cell coherence, we introduce the notion of cache savings links. Our cell-tree peeling algorithm attempts to maximize these links. Each ray has an ordered sequence of cells which it depends on. When the same dependency occurs at

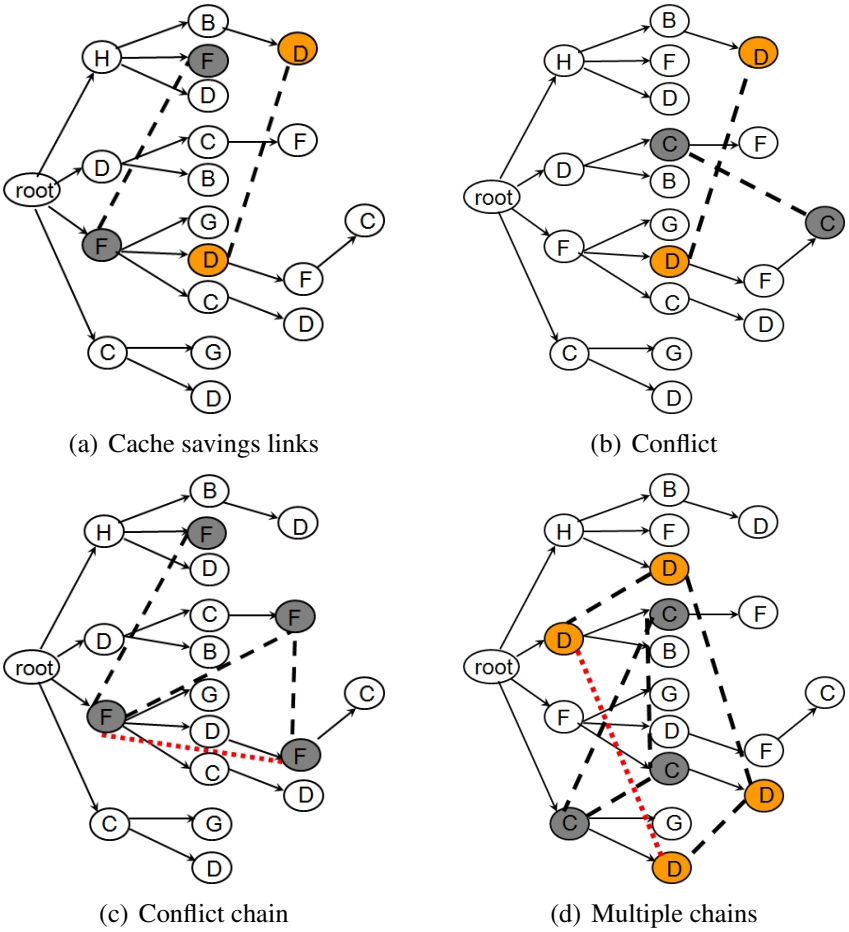


Figure 7.3: Redundant dependencies (a) Cache savings links. When the same dependency occurs at different points in the overall process, the potential cache savings is noted as a cache savings link; (b) Dependencies that cannot be combined in a feasible schedule are called conflicts; (c) Conflict caused by a chain of non-conflicting links is shown in red; (d) Conflict caused by multiple chains, which are each independently valid is shown in red.

different points in the overall process, a potential cache savings is shown in Figure 7.3(a) as a cache savings link. Dependencies that cannot be combined in a feasible schedule are called conflicts, as illustrated in Figures 7.3(b), 7.3(c), and 7.3(d). Only one of the conflicting cache savings links can be used to reduce caching. A schedule with any combination of non-conflicting link groups is a feasible one. The optimal schedule contains a maximal group of non-conflicting links.

7.3 Cell-tree Peeling

The cell-tree peeling algorithm exploits frame-to-frame coherence by using the cell-tree to find potential cache savings links. As with image composition of ray casting algorithms, there is a priority order relationship in ray traversal, which must be preserved in parallelized ray tracing. The number of times each volume block is cached depends on the order in which data cells are cached and rays processed. However, optimizing task scheduling for ray tracing is more difficult than for parallel ray casting because rays travel through the scene in a pseudo-random manner. If there is sufficient inter-frame coherence, the ray-cell dependencies remain the nearly the same from one frame to the next, so we introduce the cell-tree peeling heuristic to take advantage of frame-to-frame coherence. The dependency graph information gathered in one frame allows the cell-tree peeling algorithm to generate the cell processing schedule for the next frame.

The max-work ray-task scheduling algorithm is used by many systems that are based on ray queues, including the GI-Cube architecture, and we use it as the default algorithm for the first frame, as well as for any rays that fall outside of our cell-tree peeling algorithm schedule. In the max-work algorithm, when a render-node becomes available, the cell with the longest queue is assigned to it. As long as a cell of data is in local memory, all rays on the corresponding queue are processed before it is replaced with another cell.

In our cell-tree peeling algorithm [28], an efficient cell-processing schedule for the next frame is determined using a ray dependency graph, the cell-tree. The cell-processing schedule for the next frame is determined in reverse order from the cell-tree. The tree is processed starting with the leaves by peeling the tree nodes with the same $cellID$ and adding it to the reverse schedule. An interim tree is maintained by keeping a list of nodes that have not been included in the schedule.

If there is not sufficient inter-frame coherence, for example when there is a sudden change in the viewing parameters or in the scene, then the solution for one frame is not necessarily feasible for the next frame. In this case, there will be pending rays for one or more cells after the schedule of cells has been followed, and the max-work scheduling

algorithm is used for the rest of the frame.

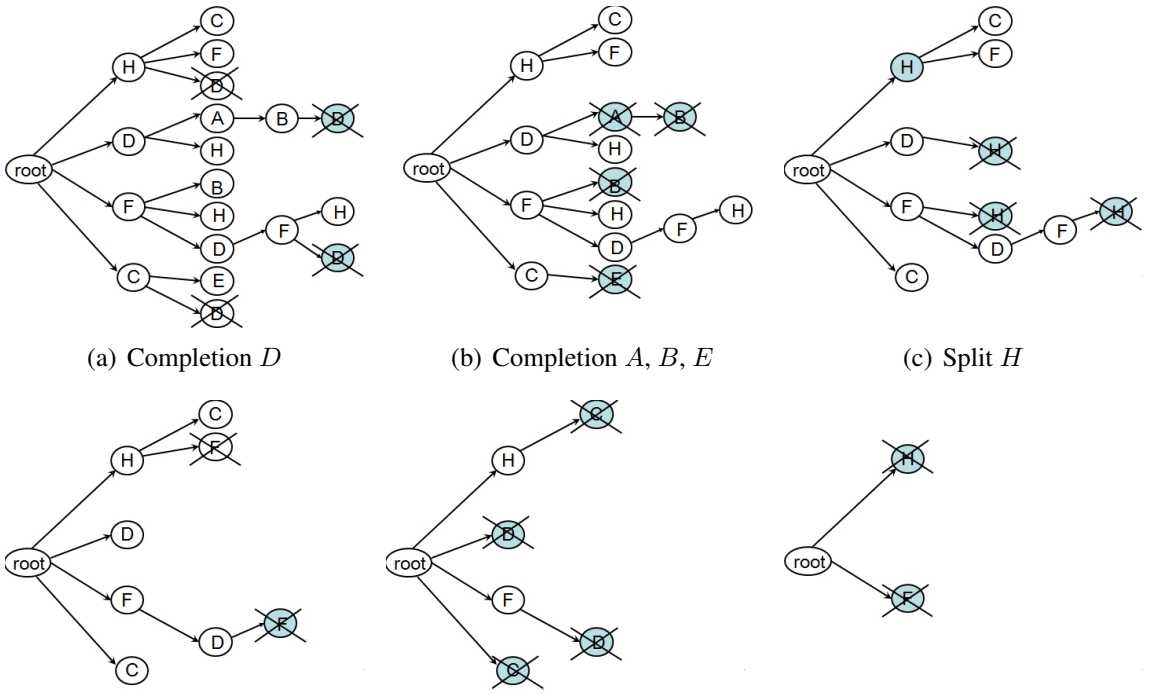
7.3.1 Definitions

Each node in the cell-tree has an associated cell ID, and more than one node may have the same cell ID. The $generation$ of a node is one plus the number of nodes with the same cell that are ancestors to that node. We define $i.maxGen$ to be the maximum generation of cell i on the current tree. If all of the maxGen nodes of a cell are leaf nodes on the current tree, then the cell is in a $ready$ state. In this case, we do a $completion$ peel on that cell by pushing it to the front of the schedule, and peeling all of its leaf nodes from the current tree. If no cell has all of its maximal generation nodes ready, the schedule must include an extra memory fetch for some cell. We call this a $split$ peel; the peel is unable to gather all remaining nodes of a cell's highest generation on the current tree.

7.3.2 Algorithm Description

The tree is processed starting with the leaves by peeling the tree nodes with the same cell ID and adding it to the reverse schedule. An interim tree is maintained by keeping a list of nodes that have not been included in the schedule. Cache savings links are found by gathering leaf nodes which correspond to the same cell. The cell-tree peeling algorithm performs well and has polynomial worst time. The cell-tree algorithm proceeds as follows until all nodes have been peeled from the tree as follows:

Determine if any cell is in the ready state
If i is in the ready state, completion peel i:
  Push i to the front of the schedule
  Peel all leaf nodes with cell=i and decrement $i.maxGen$
Else split peel cell with the most leaf nodes j:
  Push j to the front of the schedule
  Peel all leaf nodes with cell=j



(a) Completion D (b) Completion A, B, E (c) Split H

(d) Completion F (e) Completion C and D (f) Completion H and F

Figure 7.4: Cell-tree peeling. All $maxGen$ nodes for each cell being peeled are highlighted. All iterations except (c) are completion peels. A split peel is shown in (c), where inner node is highlighted because it is a $maxGen$ node for cell H. Every leaf node has some inner $maxGen$ node. H has the most leaf cells, so we add it to the schedule and remove all H leaf nodes.

Figure 7.4 shows an example of cell-tree peeling. The maximum generation nodes for cells that are peeled in each iteration are highlighted. In Figure 7.4(a), $D.maxGen$ is 2, and the only cell D generation 2 nodes are leaf nodes, so cell D is a ready cell. A completion peel of D is made, and the schedule is initialized with cell D. In Figure 7.4(b), cells A, B, and E are each ready cells, so they are added to the schedule with completion peels. The resulting sub-tree after these steps is shown in Figure 7.4(c). There is no ready cell, so Cell H is selected for a split peel since it has the most leaf nodes. As a result, the only $F.maxGen$ node is a leaf nodes in Figure 7.4(d), so it is completion peeled. In Figure 7.4(e), all C nodes are leaf nodes, so cells C and D are now in a ready state and

it is completion peeled. The resulting sub-tree after these steps is shown in Figure 7.4(f). This exposes the remaining nodes for cells H and F, which are completion peeled and the algorithm is finished with a schedule of $FHDCFHAEBD$.

7.3.3 Algorithm Correctness

The cell-tree algorithm always finds a feasible solution. A feasible schedule is one where every ray dependency is included in some subsequence of the schedule. First, we show that the cell-tree contains a link for each ray dependency. After that we show that the construction of the schedule guarantees that the partial order is not violated on the cell-tree, and that every node in the tree is included in the schedule. The cell-tree is constructed by considering every ray dependency. The rays are marked with their current place in the tree, and when a ray (original or spawned) enters a new cell, the dependency is added to the cell-tree if it is not there already. This guarantees that the cell-tree represents all ray dependencies. Hence, all ray paths exist as sub-paths of the cell-tree. Since the reverse schedule is produced by traversal from the cell-tree leaves to the root, no two jobs j and $j-1$ will be placed on the reverse schedule with job $j-1$ preceding job j. This means that the partial order of ray dependencies is never violated. No dependency is ignored since the schedule is not complete until all nodes are peeled.

The cell-tree algorithm will produce an optimal schedule if the best choices of split peels are always made. The expected amount of sub-optimality increases with the number of split peels. Empirical results indicate that the completion peels tend to dominate, thus the algorithm is close to optimal. If there are no completion peels for some sub-tree, then we know that whatever cell we peel at this level must be fetched from memory at least twice. If the current schedule has been obtained strictly by completion peels, then it is optimal.

For comparison, an optimal schedule can be determined recursively as follows. If we compare the optimal schedule of each sub-tree that is obtained by peeling each cells' leaf nodes from the current sub-tree, one cell at a time, the optimal for the original tree is

the current schedule concatenated with the cell whose peel results in the sub-tree with the shortest optimal schedule, together with the optimal schedule of that sub-tree. The recursion can be ended when any sub-tree is found to have all completion peels. If each iteration reveals another split, then a recursive call is made for each cell until the root is reached, and each of these peels may involve only a single node. This means that the recursive algorithm has a worst time bound of O(n!), where n is the number of nodes in the cell-tree. Furthermore, a clone of the current tree must be kept at each stage of the recursion. Both of these problems make the recursive algorithm impractical. We use a simpler version, with polynomial worst time bounds, which is close to optimal. Instead of determining the best choice for the split peel recursively, we simply choose the cell with the most leaf nodes. Alternate criteria include maximum generation or maximal tree level.

7.3.4 Worst Time Bounds

The minimal schedule is equal to the sum of each cell's maximum generation plus the minimum number of split peels. This can be shown as follows: The minimum number of memory fetches for a cell is the largest generation number of that cell. This follows from the definition of generation. If cell i is part of a completion, then peeling cell i would result in no extra memory fetches compared with an optimal schedule. When a completion peel is made, exactly one generation, g, of the cell is completed. Although some nodes that are peeled may be from a smaller generation, f, all of the nodes from f cannot be peeled because each chain leading to the generation g node also contains a generation f node which cannot be peeled at the same time. Thus cell i must be cached in at least once for generation g and a separate time for generation f.

7.4 Architecture Simulation

Ray tracing in our framework is based on the GI-Cube [18] ray tracing coprocessor simulation. It is designed to accelerate volume rendering with Phong shading and local

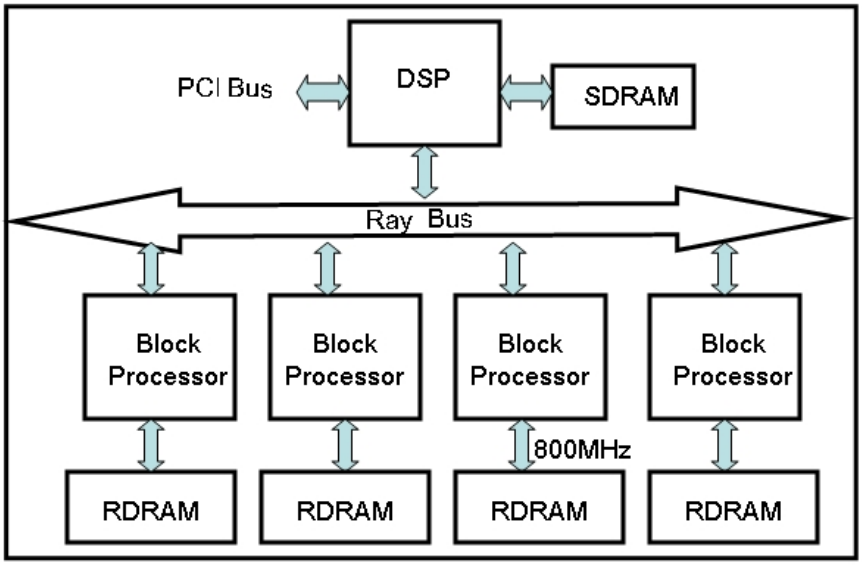


Figure 7.5: GI-Cube ray tracing PCI board.

illumination, as well as with global illumination including shadow casting, reflections, glossy scattering and radiosity. In addition, it provides volumetric ray tracing acceleration support for various algorithms including hyper-texture, photon maps, polygonal global illumination, tomographic reconstruction, bi-directional path tracing, volumetric textures and BRDF evaluation. The volume is subdivided among the processors. We have simulated a ray tracing architecture as an extension of the GI-Cube architecture.

7.4.1 GI-Cube Ray Tracing Architecture

The GI-Cube design has three major components: the Digital Signal Processor (DSP), the processors and the memory. A block diagram of the system is illustrated in Figure 7.5. The DSP is directly connected to the frame buffer and has its own SDRAM. It loads the data set, generates lighting and viewing rays, controls processor I/O and sends the result over the PCI interface. The processors maintain and sort a group of fixed size hardware queues of rays. Each queue is implemented as a pipelined insertion sorter on a separate embedded DRAM (eDRAM) and can hold up to 256 rays. The active queue is processed until it is empty. The queue with the most rays is selected when a processor becomes available.

7.4.2 DSP Cell-tree Scheduling Simulation

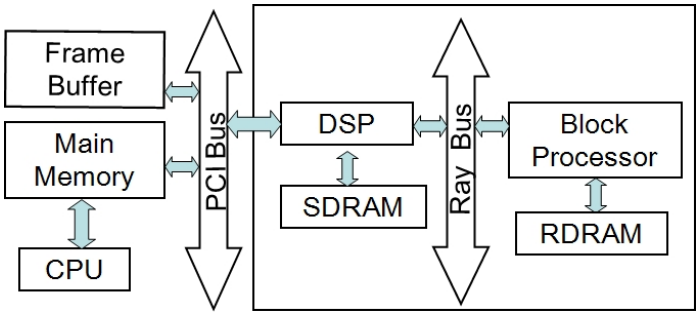


Figure 7.6: Proposed ray tracing architecture PC board.

We simulated a DSP implementation of a ray tracing architecture using cell-tree scheduling to extend the GI-Cube simulations. The block diagram of a ray tracing node for these simulations is shown in Figure 7.6.

The DSP56311 can perform 255 MIPS, and has three megabits of on-chip static RAM [81]. The DSP handles data set loading, generates lighting and viewing rays, controls processor I/O, and sends the final image to the frame buffer. It has a dedicated SDRAM to hold the ray queues of cells not currently in memory. The queues for cells that are currently in on-board RDRAM memory are kept in processor eDRAM, as in GI-Cube. When the volume currently in RDRAM memory no longer has any pending rays, the DSP fetches a different sub-volume from disk memory.

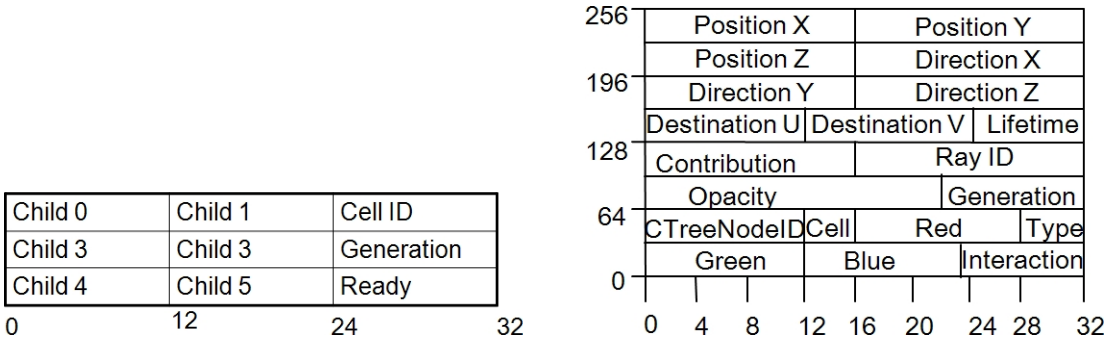
We utilize the idle cycles in the DSP to determine a processing schedule for frame $i+1$ based on ray coherency information of frame i. As the DSP generates initial rays, the cell-tree is initialized with one node for each cell that contains any eye rays. Our ray tracing research is focused on improving memory performance for super volumes using both ray coherence and inter-frame coherence. Distributed ray tracing is used for data which cannot fit into the local memory of the rendering mechanism. As with our distributed ray casting approach, the scene data is divided into axis-aligned cells. Each time a ray exits the cell currently in memory, a ray packet is sent over the ray bus to the DSP. The bus frequency of 100 MHz allows one ray packet to be sent per processing cycle. The processor can generate

either an exit ray or a neighbor ray in a cycle, but not both. This means that in any cycle at most one ray is sent to the DSP. The DSP consolidates the ray dependencies into a cell-tree during idle duty cycles. The DSP computes the schedule for loading the sub-volumes for the next frame. This is done between frames. Alternatively, if a slower DSP is used, the schedule can be created during idle cycles during the next frame. A new schedule can be calculated for each frame in order to maximize the exploitation of inter-frame coherence. The algorithm used by the DSP is further explained in the next section.

Cell-tree creation is done during idle DSP cycles. Each cell-tree node creation requires one write. Each ray packet is used for a single decision and at most one write to the cell-tree. The cell-tree node ID is updated before the ray packet is placed on a queue in the DSP. When a ray needs to be queued by the DSP because it is exiting the cell currently in memory, the ray packet is sent over the ray bus. The cell dependency information, included in this packet, is gathered by the DSP. A prototype layout for the cell-tree node is shown in Figure 7.7(a). The DSP RAM is used for storing the cell-tree; 96 KB are allocated to hold up to $8,000$ 12-Byte cell-tree nodes in our simulations.

In addition to the cell-tree itself, each ray contains a cell-tree node id, a cell id and a ray id. These fit in the GI-Cube ray packet by decreasing the size of some items without any resulting image quality degradation (see Figure 7.7(b)). The on-board RDRAM memory bandwidth of 0.8GB/sec in our design is sufficient to avoid nearly all stalls due to RDRAM fetches, so the only overhead results from retrieving data from disk. If smaller and/or cheaper memory is used, our algorithm could be applied between the on-board and the cache and would have an additional impact on rendering time.

Schedule creation is done in the DSP between frames. In our simulations the average number of peels, including split peels, was 55, which is much less than the worst case, which would be one peel per node. This is because peels usually result in the removal of several nodes, and because in most iterations there is a completion peel. In our tests the average number of nodes examined for each completion peel was 85. Based on our experiments, we estimate the time for schedule creation in a 255 MIPS DSP to be between



(a) Cell-tree node (b) Bit widths in the 32 byte ray packet

Figure 7.7: Proposed ray tracing architecture data structures.

5 and 10 microseconds. If, instead, the schedule for frame $i + 1$ is created while the processor is rendering i, a copy of the cell-tree from frame $i - 1$ must remain on the DSP, thus doubling the DSP memory requirement.

7.5 Results

We simulated our system in C++ using scenes which include a mix of volume and polygonal data. The images rendered for testing the algorithm are shown in Figure 7.8. Three scenes are shown, each rendered with a 256×256 image size. The first scene is an MRI brain reflected in multiple mirrors for a complete look. It includes one $128 \times 128 \times 84$ by eight bits/voxel volume and three planes. The second scene is two clouds and a moon reflected in a lake. It includes two $120 \times 120 \times 120$ volumes, a geometric sphere and three planes. The third is a lobster reflected several times to give a tunnel appearance. It includes one $256 \times 254 \times 57$ volume and five planes. All of the volumes had eight bit voxels.

In our simulations, the cell-trees had an average of 1797 nodes (see Figure 7.9), which is more than 100 times smaller than the average number of rays, which is $199,919$. Figure 7.10 illustrates the scalability of tree sizes as image size increases. It shows the number of ray dependencies for the brain scene at resolutions of 100×100, 256×256 and 512×512 pixels. We use a logarithmic scale because the cell-tree sizes grow much more slowly than the number of ray dependencies. The sizes of the cell-trees did not grow nearly as quickly

(a) CT brain with mirrors

(b) Clouds and moon reflecting in a lake

(c) CT lobster reflected in several mirrors give a tunnel appearance

Figure 7.8: Ray traced images for algorithm testing.

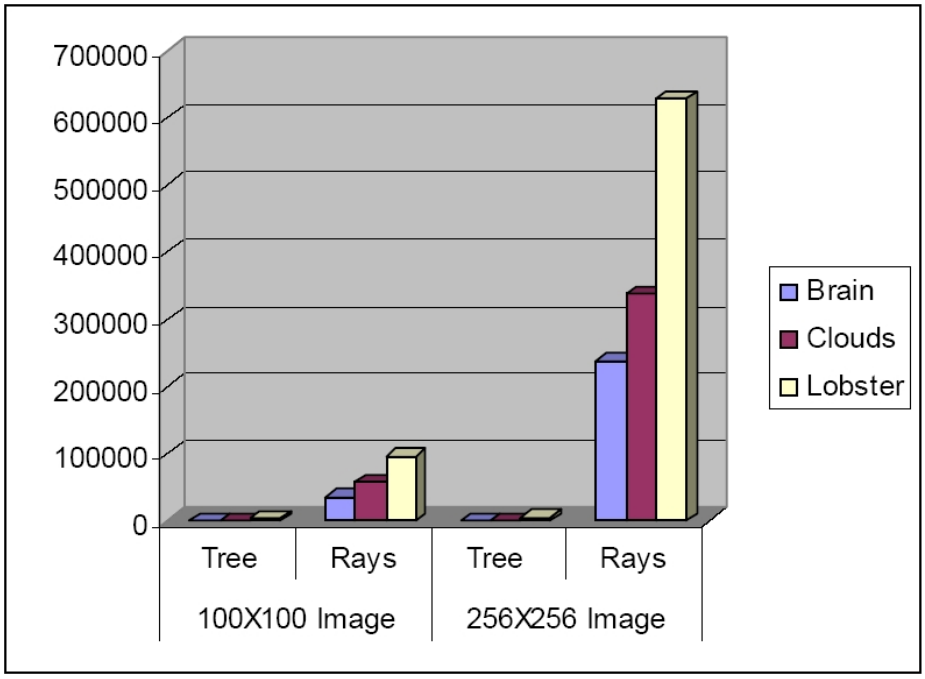


Figure 7.9: Nodes in cell-tree compared with number of ray dependencies represented. Cell-tree size is orders of smaller than number of ray dependencies.

as the number of rays. Image and volume sizes each had a relatively small influence on cell-tree sizes. The biggest factor in the cell-tree sizes was the number of reflections.

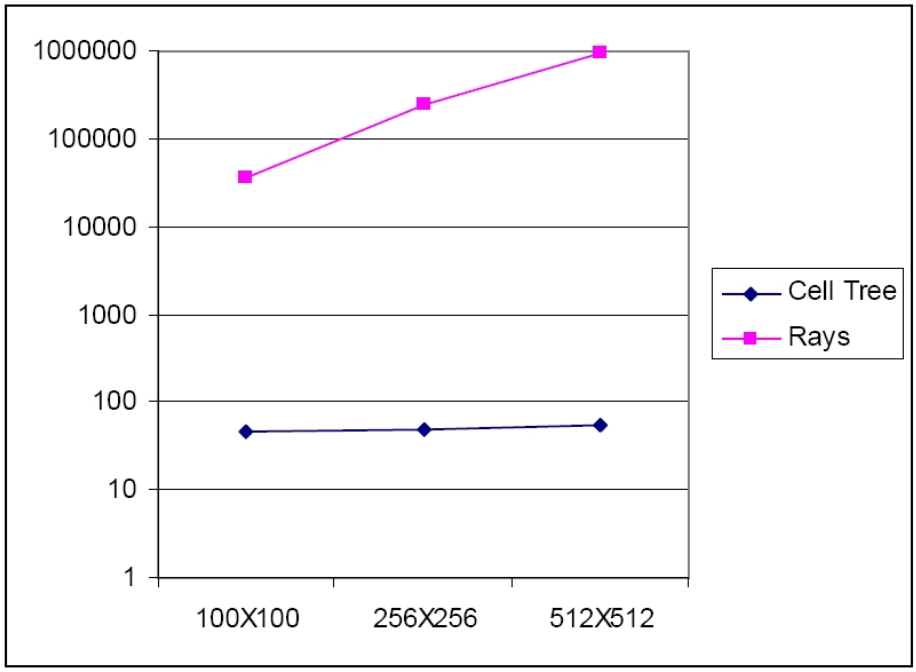


Figure 7.10: Cell-tree Scalability. Cell-tree sizes for ray-traced brain using different image sizes.

In order to test our cell-tree construction and schedule creation algorithms, we constrained the on-board memory to be $1/8$ and $1/27$ of the volume for each scene. The relative number of misses with the cell-tree schedule compared with the max-work algorithm schedule, and with the lower bounds of the optimal schedule, are shown in Figure 7.11.

The max-work algorithm solves the *on-line* version of the scheduling problem described in Subsection 7.1.2. This means it solves the problem without knowing a priori what the ray dependencies are. The cell-tree algorithm solves the *off-line* version, where the dependencies are known. For the first frame, there is no dependency information at all, so the max-work algorithm is used. The queue with the most rays is selected when a processor has completed all rays in the queue of its current cell. As rays are generated, a cell-tree is constructed. This is used to determine a better cache schedule for the next frame.

The memory fetches decreased an average of 30% by using dependency graph based scheduling, with a range of 20% to 37%. Also shown are the lower bound sizes of the optimal schedule. This is a very conservative lower bound. It assumes that the only necessary

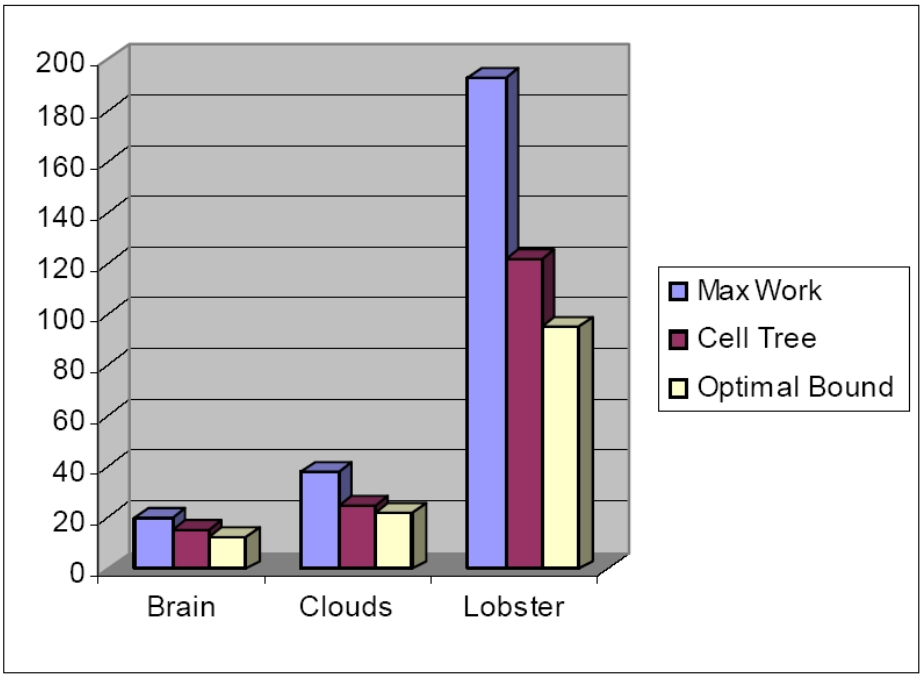


Figure 7.11: Performance comparison. Memory fetches with the max-work and the cell-tree peeling algorithms, and the lower bounds of the optimal schedule.

split peel is the first one, which is not likely to be the case. The cell-tree schedule was always within 33% of this lower bound. The performance improvement is greatest when there are more inter-reflections, which is also when it is needed most because the cells must be fetched from memory more frequently. The resulting frame rate increase depends on the relationship between disk fetch time and on board memory cache-in time.

By comparing the number of split peels to the number of completion peels we can get a feeling of how close to optimal the schedule is. The algorithm performs best when there are relatively few split peels. Each iteration of cell-tree peeling removes at least one node from the tree. Each node removal requires checking each node in the tree at most once. Thus, the worst time case for cell-tree peeling is O(n^2) where n is the number of tree nodes. The upper bound of the cell-tree size is the total number of rays created. However this is an unrealistic bound in that it would mean that each ray is independent of every other ray.

7.6 Summary

In this chapter we provided an overview of our ray tracing acceleration framework. We have demonstrated the effectiveness of using ray dependency information in reducing the total number of memory fetches for a ray tracing system where the volume is subdivided into cells. The cell-tree can be constructed easily in hardware with virtually no cost. Since the cell-tree takes very little time and memory to build, it can easily be rebuilt for each frame, which increases the chances of having sufficient inter-frame coherence. Depending on the implementation platform this can be done either between frames at very high processing speeds, or during frame i to be used in frame $i + 1$. A data-structure containing a compact description of cell-dependency information gives us a scientific means to compare several configurations.

Our cell-tree peeling algorithm performs well and has polynomial worst time. Ray trace rendering efficiency improves using ray dependency encryption. In our experiments, ray-task schedules created by our cell-tree peeling algorithm result in an average cache miss reduction of 30% during ray trace mode compared with the max-work algorithm. The cell-tree algorithm could be extended for dynamic load balancing in both distributed and shared memory systems. Simulations have indicated that volume sub-division could be improved using ray dependencies. A future work would be to take advantage of geometric clusterings [12] to increase the scalability of our ray tracer.

The cell-tree allows us to study the best way to split up a volume. For example, when a ray is being reflected between two cells it is likely that the load balance would be improved by a new subdivision which places the parts of the volume being reflected into the same subdivision. Allowing multiple cells per memory unit allows greater flexibility in the schedule at the cost of further complexity. This trade-off should be studied further.

An extension of the algorithm should prove useful in multiprocessor scheduling. If p cells are to be processed at the same time, and cell i is repeated in the schedule within p processes, it may be handled by the same processor, but it will stall while waiting for other cells which the second occurrence of cell i is dependent on (and the first one is not), to

complete processing. The tradeoff of waiting for other processors and fetching the cell an extra time should be evaluated. This promises to be a richly theoretical as well as useful endeavor.

Chapter 8

Conclusions

In this dissertation, we have presented a framework for distributed volume visualization. We have developed several techniques to improve the efficiency and scalability of massive data management. Parallel ray casting is accomplished by independently rendering portions of the scene and compositing the images together. The correctness of the compositing operation depends on a deterministic image composition order. We have introduced the problem of finding an optimal partition of data for distribution under deterministic viewing conditions as the LBND problem. The use of dynamic programming for volume distribution is also introduced.

In order to achieve sufficient resolution for scientific and medical studies, and to utilize all available information, we do not use any down-sampling. Volumetric data sets frequently contain large portions of empty space, or regions that will never be rendered. We take advantage of this empty space. We reduce data sizes early in the pipeline and retain data extent and brick dependency information as it becomes available. By removing empty space early in the pipeline, both network bandwidth and rendering workloads are reduced.

8.1 Summary of Contributions

We have introduced several out-of-core techniques for processing massive data sets. Segmentation is used to find a region of interest, or portion of data to be rendered. Region growing is a segmentation algorithm that expands on an a priori seed voxel in the region of interest. It is limited to small data sets, so we have introduced out-of-core region growing, which processes a slab of consecutive data slices at a time. The valid seeds for each slab is output from the previous slab. We have also introduced the slab-projection slice to encrypt empty space information so that preprocessing steps used for data distribution can proceed without moving data in and out of main memory. These steps include out-of-core bricking and kd-tree partitioning.

We have demonstrated a mapping of the LBND problem to the NP-complete job-shop scheduling problem. The use of dynamic programming for volume distribution is also introduced. Brick grouping DP minimizes a cost function on geometrically coherent groups of bricks to partition a directed acyclic graph of bricks into a view-dependent load balanced data distribution. Moving walls DP finds a view-independent flex-block partition using slab-projection slices directly. Our flex-block contains a combination of empty space and a cropped subvolume. Our flex-block tree, which represents a flex-block partition, is similar to a kd-tree partition, except that the cut planes do not alternate, and partition cells, the tree leafs, are flex-blocks.

Ray tracing is used to enhance rendering realism with global illumination. It is similar to ray casting, except that additional rays are spawned at ray-object intersection points to approximate reflected, refracted and shadow rays. With parallel ray tracing, scene cells cannot be rendered independently. A ray segment is the intersection of a ray with a scene cell, and the set of these defines a tree of ray-cell dependencies. The order in which ray segments are processed impacts the end-to-end rendering time, and the problem of finding the optimum processing order is NP-complete. We have introduced the cell-tree, which is a concise representation of ray-traversal dependencies, and cell-tree peeling, a dependency graph ray-task scheduling for ray tracing. The cell-tree peeling algorithm is designed to be

run in parallel with ray tracing process cycles.

8.2 Summary of Results

This research has provided several algorithms which automate the volume distribution and task scheduling processes for volume visualization. Our algorithms improve the efficiency and scalability of global illumination, segmentation, and volume rendering in a PC cluster. Although new graphics, networking and image compositing hardware have emerged continuously throughout this research, these changes in technology have not affected the applicability of these algorithms, which have been designed to be hardware independent.

Our out-of-core region growing enables segmentation of massive volumetric data. We use a heuristic to choose volume boundaries to crop away empty space and create subvolumes which may be set to a particular maximum size. For data sets with a large portion of empty space, cropping reduced memory consumption by an average of 68%. Our slab-projection slice encodes data extent information for out-of-core preprocessing Dynamic programming distributes data using a cost function to reduce load imbalance by an average of 100

Our out-of-core bricking algorithm creates bricks of data which are connected by a DAG which represents the relative priority order of these from the view direction defined to follow the input slice data, the z-axis in our experiments. These bricks are sized primarily to fit GPU hardware, but may be larger to prevent the recursion from becoming intractable. Our brick grouping DP algorithm finds a partition that is optimal with respect to these bricks and DAG. However, the average time of the algorithm depends on the brick sizes.

We have also introduced the moving walls DP algorithm, which finds a near-optimal solution to the LBND problem. A flex-block partition is produced from a sweep through a series of slab-projection slices while shrink-wrapping volume blocks to remove empty space. These algorithm have the advantage of using a cost function, which facilitates the

study of various cluster configurations by using algorithm parameter changes.

Our dependency graph approach uses neighbor block information from the flex-block tree to determine which neighbors a ray may enter, and for image composition order for parallel ray casting. We achieve more control over allocation of volumetric data between render-nodes on a visualization cluster, and ray-task scheduling for ray tracing. Our other dependency graph data structure, the cell-tree, represents dependencies between ray segments. A single link in the cell-tree encodes hundreds to thousands of ray-cell traversal dependencies. The cell-tree is constructed as a simple piggy-back operation to the ray traversal process during ray tracing.Our cell-tree peeling algorithm, which is a ray-task scheduling for distributed volumetric ray tracing based on our automatic ray dependency encoding. Simulations have shown that the cell-tree peeling algorithm can be used to improve cache coherency by an average of 30%.

Several massive volumetric data sets have been rendered on the Stony Brook Visual Computing Cluster as part of this research. These volumes have sizes that would not allow region-growing or direct volume rendering without using out-of-core and/or parallel methods. Our techniques reduce the average segmentation and rendering time significantly for all data-sets segmented and rendered.

8.3 Near-Term Future Work

An immediate research goal is to incorporate algorithms developed in this research into the HP open source Parallel Compositing Library software package.Scalable priority-constrained data distribution is an open area of research. We have introduced a solution, using dynamic programming, which allows a cost function to drive the distribution process. As demonstrated by our test results, our solution allows the scene partition to be controlled so that data is distributed more evenly between render-nodes. The portion of the scene assigned to each render-node is dictated by the distribution of empty space in the scene, resulting in better load balancing than with traditional partitioning methods such as octrees.

Our framework for managing of massive volumetric data sets has opened new avenues for future research. Algorithms have been designed to be independent of the underlying hardware and operating system. They are equally appropriate for implementation on a cluster, on a special purpose hardware, or on programmable FPGA or DSPs hardware. Our dependency graph data structures can also be used in conjunction with other data primitives such as geometry data. Current and near-future work includes running additional tests on several data sets to determine the impact our algorithms have on rendering in different cluster environments, as well as other hardware settings.

In our framework, we use early data reduction to reduce problems associated with distribution and rendering of massive volumetric data sets. The slab-projection slice kd-tree has some advantages over the dynamic programming model. Recent developments in dynamic scanner technologies [109], produce time varying data scans which are very well suited for these render clusters due to their tremendous size. Further research is needed for adapting the partition found during our dynamic programming preprocess. A likely solution is to use an incremental update approach to move partitions as regions of non-empty voxels move throughout the scene. The transfer function driven dynamic load balancing method proposed by Li et al. [69] marks value ranges in each cell, and is applicable to the cells in our data distribution.

Future work includes adjusting our algorithms to manage dynamic scenes. We plan to explore the impact of dynamic volumes on our algorithms. Changes in a particular volume block may result in localized changes to the scene. An incremental version of our moving walls algorithm would be an interesting avenue to explore for these changes. The cell-tree building and peeling algorithms do not need additional updates to handle dynamic volumes, as long as the underlying data structure has a well-defined neighbor relationship between scene cells. The inter-frame coherency is affected by dynamic scenes, and this would be an interesting area for further exploration.

The cell-tree algorithm is designed to be run in parallel with ray tracing at run-time, so no change should be required for dynamic scenes. The dependency graph approach could

also be adapted for data primitives other than volumes. In this case the brick would be a bounding volume of the primitives. Our dependency graph approach is extendible to any scene partition which has an unambiguous distance order from any viewpoint, including more general BSP partitions.

8.4 Extended Vision

Our long-range research goal is to develop data management techniques which support other large-scale, distributed systems to solve real-world problems using a solid theoretical basis. For example, visual analytics applications in cyber security extract information from an vast volume of data to create a intuitive picture of the vulnerabilities of critical infrastructure. This is a rich area of research with many of the same underlying characteristics as distributed volume visualization.

A long-range research goal is to develop data management techniques which support other large-scale, distributed systems and computationally intense science and engineering simulations. Complex computer systems require computer scientists to work closely with experts in application areas to insure that stringent mathematical models are accurately implemented. The dependency graph and dynamic programming approaches could be extended to data management in other contexts such as large-scale simulations.

In addition, visualization is becoming increasingly important in education contexts. Volume rendering allows real-time manipulation of 3D objects in space. Visualization adds an artistic aspect to the process to illustrate complex concepts to enhance understanding. Today's students are exposed to interactive games from an early age, making a multi-media format appropriate for teaching. Familiarity with games is likely to make them a successful platform for instilling a positive attitude toward subject areas which are difficult to teach in a traditional setting. This is especially important for abstract concepts in science, technology, engineering and math, or the STEM subject areas.

Bibliography

[1] R. Adams and L. Bischof. Seeded region growing. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16:641–647, 1994.

[2] S. Adelson and L. Hodges. Stereoscopic ray-tracing. *The Visual Computer*, 10(3):127–144, Dec. 1993.

[3] K. Akeley. Reality Engine graphics. *ACM SIGGRAPH Computer Graphics*, pages 109–116, 1993.

[4] J. Amanatides and A. Woo. A fast voxel traversal algorithm for ray tracing. *ACM SIGGRAPH Eurographics*, pages 3–10, 1987.

[5] J. Arvo and D. Kirk. Particle transport and image synthesis. *ACM SIGGRAPH Computer Graphics*, 24(4):63–66, August 1990.

[6] M. Ashikhmin, S. Premoze, and P. Shirley. A microfacet-based BRDF generator. *ACM SIGGRAPH Computer Graphics*, pages 65–74, July 2000.

[7] C. Aykanat, V. Isler, and B. Ozguc. An efficient parallel spatial subdivision algorithm for parallel ray tracing complex scenes. *First Bilkent Computer Graphics Conference, ATARV-93*, 26:883–890, 1994.

[8] R. E. Bellman. *Dynamic Programming*. Dover, 1957.

[9] J. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[10] I. Bitter and A. Kaufman. A ray-slice-sweep volume rendering engine. *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 121–130, 1997.

[11] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. *ACM Symposium on Volume Visualization*, pages 91–98, 1994.

[12] V. Capoyleas, G. Rote, and G. Woeginger. Geometric clusterings. *Canadian Conference on Computer Geometry*, pages 28–31, 1990.

[13] A. Chalmers. *Practical Parallel Rendering*. A K Peters, 2002.

[14] E. Chan, R. Ng, P. Sen, K. Proudfoot, and P. Hanrahan. Efficient partitioning of fragment shaders for multipass rendering on programmable graphics hardware. *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 69–78, Sept. 2002.

[15] S. Chen, H. Rushmeier, G. Miller, and D. Turner. A progressive multi-pass method for global illumination. *ACM SIGGRAPH Computer Graphics*, 25:165–174, July 1991.

[16] D. Cohen. Voxel traversal along a 3D line. pages 366–369. 1994.

[17] R. Cook, T. Porter, and L. Carpenter. Distributed ray tracing. *ACM SIGGRAPH Computer Graphics*, 18:137–145, July 1984.

[18] F. Dachille and A. Kaufman. GI-Cube: An architecture for volumetric global illumination and rendering. *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 119–128, August 2000.

[19] F. Dachille, K. Kreeger, B. Chen, I. Bitter, and A. Kaufman. High-quality volume rendering using texture mapping hardware. *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 69–76, 1998.

[20] M. de Boer, A. Gröpl, J. Hesser, and R. Manner. Latency and Hazard-Free Volume Memory Architecture for Direct Volume Rendering. *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 109–118, 1996.

[21] D. E. DeMarle, S. G. Parker, M. Hartner, C. Gribble, and C. D. Hansen. Distributed interactive ray tracing for large volume visualization. *IEEE Symposium on Parallel Visualization and Graphics*, pages 87–94, 2003.

[22] M. Doggett, M. Meissner, and U. Kanus. A low-cost memory architecture for pci-based interactive ray casting. *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 7–13, 1999.

[23] K. Engel, M. Kraus, and T. Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 9–16, Sept. 2001.

[24] R. Fajardo, T. Ryan, and J. Kappelman. Assessing the accuracy of high-resolution X-ray computed tomography of primate trabecular bone by comparisons with histological sections. *American Journal of Physical Anthropology*, 118(1):1–10, 2001.

[25] J. Falby, M. Zyda, D.Pratt, and R. Mackey. NPSNET: Hierarchical data structures for real-time three-dimensional visual simulation. *IEEE Computer Graphics and Applications*, 17(1):65–69, 1993.

[26] J. Flaherty, R. Loy, M. Shephard, B. Szymanski, J. Teresco, and L. Ziantz. Parallel structures and dynamic load balancing for adaptive finite element computation. *Applied Numerical Math*, 26:241–263, 1998.

[27] J. Foley, A. van Dam, S. Feiner, and J. Hughes. Computer graphics: Principles and practice in C. Addison-Wesley Professional, 1995.

[28] S. Frank and A. Kaufman. Dependency graph scheduling in a volumetric ray tracing architecture. *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 127–135, Sept. 2002.

[29] S. Frank and A. Kaufman. Distributed volume rendering on a visualization cluster. *CAD/Graphics*, pages 371–376, 2005.

[30] S. Frank and A. Kaufman. Dependency graph approach to load balancing distributed volume visualization. The Visual Computer, 2009.

[31] S. Frank and A. Kaufman. Out-of-core and dynamic programming for data distribution on a volume visualization cluster. Computer Graphics Forum, 2009.

[32] G. Frieder, D. Gordon, and R. Reynolds. Back-to-front display of voxel-based objects. *IEEE Computer Graphics and Applications*, 5(1):52–60, 1985.

[33] A. Ghosh, P. Prabhu, A. Kaufman, and K. Mueller. Hardware assisted multichannel volume rendering. *Computer Graphics International*, pages 2–7, July 2003.

[34] A. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, 1984.

[35] A. S. Glassner. *Principles of Digital Image Synthesis, Volume Two*. Morgan Kaufmann Series in Computer Graphics and Geometric Modeling, 1995.

[36] A. Glassner, ed. *An Introductin to Ray Tracing*. Morgan Kaufmann, 1989.

[37] S. M. Goldwasser, R. A. Reynolds, T. Bapty, D. Baraff, J. Summers, D. A. Talton, and E. Walsh. Physician's workstation with real-time performance. *IEEE Computer Graphics & Applications*, 5(12):44–57, Dec. 1985.

[38] C. Goral, K. Torrance, D. Greenberg, and B. Battaile. Modelling the interaction of light between diffuse surfaces. *ACM SIGGRAPH Computer Graphics*, 18:213–222, July 1984.

[39] D. Gordon and R. A. Reynolds. Image space shading of 3-dimensional objects. *Computer Vision, Graphics, and Image Processing, 29*, pages 361–376, 1985.

[40] E. A. Haines and D. P. Greenberg. The light buffer: A ray tracer shadow testing accelerator. *IEEE Computer Graphics and Applications*, 6(9):6–16, 1986.

[41] A. Heirich. Optimal automatic multi-pass shader partitioning by dynamic programming. *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 91–98, 2005.

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

[43] B. Hendrickson and T. Kolda. Graph partitioning models for parallel computing. *Parallel Computing*, 26:1519–1534, 2000.

[44] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. *ACM/IEEE conference on Supercomputing*, pages 435–446, 1995.

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

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

[47] W. Hunt, W. Mark, and G. Stoll. Fast kd-tree construction with an adaptive error-bounded heuristic. *IEEE Symposium on Interactive Ray Tracing*, pages 81–88, 2006.

[48] H. Igehy, M. Eldridge, and P. Hanrahan. Parallel texture caching. pages 95–106, 1999.

[49] D. Jackel and W. Strasser. Reconstructing solids from tomographic scans - the PARCUM II system. *Advances in Computer Graphics Hardware II*, pages 209–227, 1988.

[50] H. W. Jensen. Global illumination using photon maps. *Eurographics Rendering Workshop*, pages 21–30, 1996.

[51] H. W. Jensen. *Realistic Image Synthesis using Photon Mapping*. A K Peters, 2001.

[52] G. Karypis. Multi-constraint mesh partitioning for contact/impact computations. In *ACM/IEEE Conference on Supercomputing*, page 56, 2003.

[53] G. Karypis and V. Kumar. Metis: unstructured graph partitioning and sparse matrix ordering system. Technical Report, 1995.

[54] A. Kaufman and R. Bakalash. Memory and processing architecture for 3d voxel-based imagery. *Computer Graphics and Applications*, 8(6):10–23, November 1988.

[55] A. Keller. Instant radiosity. *ACM SIGGRAPH Computer Graphics*, pages 49–56, 1997.

[56] G. Kindlmann, R. Whitaker, T. Tasdizen, and T. Moller. Curvature-based transfer functions for direct volume rendering: Methods and applications. *IEEE Visualization*, pages 513–520, 2003.

[57] J. Kniss, G. Kindlmann, and C. Hansen. Interactive Volume Rendering Using Multi-Dimensional Transfer Functions and Direct Manipulation Widgets. *IEEE Visualization*, pages 255–262, 2001.

[58] G. Knittel. VERVE - voxel engine for real-time visualization and examination. *Computer Graphics Forum*, 12(3):37–48, 1993.

[59] G. Knittel and W. Strasser. VIZARD: Visualization accelerator for realtime display. *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 139–147, August 1997.

[60] A. Knoll, I. Wald, S. Parker, and C. Hansen. Interactive isosurface ray tracing of large octree volumes. pages 115–124, 2006.

[61] K. Kreeger and A. Kaufman. Interactive volume segmentation with the PAVLOV architecture. *Symposium on Parallel Visualization and Graphics*, pages 61–68, October 1999.

[62] J. Krüger and R. Westermann. Acceleration Techniques for GPU-Based Volume Rendering. *IEEE Visualization*, pages 287–292, 2003.

[63] P. Lacroute and M. Levoy. Fast volume rendering using a shearwarp factorization of the viewing transform. *ACM SIGGRAPH Computer Graphics*, pages 451–457, July 1994.

[64] S. Lakare. *Ray Based Exploration of Volumetric Data*. PhD thesis, Stony Brook University, 2004.

[65] S. Lakare and A. Kaufman. OpenVL: The open volume library. *International Workshop on Volume Graphics*, pages 69–78, July 2003.

[66] S. Lakare and A. Kaufman. Light weight space leaping using ray coherence. *IEEE Visualization*, pages 19–26, 2004.

[67] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(5):29–37, May 1988.

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

[69] W. Li, K. Mueller, and A. Kaufman. Empty space skipping and occlusion clipping for texture-based volume rendering. *IEEE Visualization*, pages 317–324, Oct. 2003.

[70] B. Lichtenbelt. Design of a high performance volume visualization system. *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 111–119, 1997.

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

[72] K. Ma and J. Painter. Parallel volume visualization on workstations. *IEEE Computer Graphics and Applications*, 17:31–37, 1993.

[73] K. Ma and S. Parker. Massively parallel software rendering for visualizing large-scale data sets. *IEEE Computer Graphics and Applications*, 21(4):72–83, 2001.

[74] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. Smart memories: a modular reconfigurable architecture. *ACM International Symposium on Computer Architecture*, pages 161–171, 2000.

[75] W. Mark, R. S. Glanville, K. Akeley, and M. Kilgard. Cg: A system for programming graphics hardware in a C-like language. *ACM Transactions on Graphics*, 22(3):896–907, July 2003.

[76] M. Meissner, U. Kanus, and W. Strasser. VIZARD II, a PCI-card for real-time volume rendering. *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 61–68, 1998.

[77] L. Moll, A. Heirich, and M. Shand. Sepia: scalable 3D compositing using PCI pamette. *IEEE Symposium on Field Programmable Custom Computing Machines*, pages 146–155, April 1999.

[78] S. Molnar, J. Eyles, and J. Poulton. PixelFlow: High-speed rendering using image composition. *Computer Graphics*, 26(2):231–240, 1992.

[79] B. Moloney, D. Weiskopf, T. Mller, and M. Strengert. Scalable sort-first parallel direct volume rendering with dynamic load balancing. *Eurographics Symposium on Parallel Graphics and Visualization*, pages 45–52, May 2007.

[80] C. Morris and D. S. Ebert. Direct photographic volume rendering using multi-dimensional color-based transfer functions. pages 115–124, 2002.

[81] Motorola Semiconductor Products Sector. Digital signal processors: Product specification. $http://ewww.motorola.com/webapp/$, 2002.

[82] S. Muraki, E. Lum, K. Ma, M. Ogata, and X. Liu. A PC cluster system for simultateous interactive volume modeling and visualization. *IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 95–102, October 2003.

[83] H. Nishimura, T. Endo, T. Maruyama, J. Saito, and P. H. Christensen. Parallel rendering and the quest for realism: The KILAUEA massively parallel ray tracer. *SIGGRAPH Course Notes*, pages 1–59, Aug. 2001.

[84] N.Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, pages 99–108, 1995.

[85] A. Olejniczak and F. Grine. Assessment of the accuracy of dental enamel thickness measurements using micro-focal X-Ray computed tomography . *The Anatomical Record Part A: Discoveries in Molecular, Cellular, and Evolutionary Biology*, 288A(3):263–275, 2006.

[86] M. Ozdal and C. Aykanat. Hypergraph models and algorithms for data-pattern-based clustering. *Data Mining and Knowledge Discovery*, 9(1):29–57, 2004.

[87] J. Park, J. Chung, S. Hwang, B. Shin, and H. Park. Visible Korean Human: Its techniques and applications. *Clinical Anatomy*, 19:216–224, 2006.

[88] S. Parker, W. Martin, P.-P. J. Sloan, P. S. Shirley, B. Smits, and C. Hansen. Interactive ray tracing. *ACM Symposium on Interactive 3D Graphics*, pages 119–126, 1999.

[89] S. Parker, M. Parker, Y. Livnat, P.-P. Sloan, C. Hansen, and P. Shirley. Interactive ray tracing for volume visualization. *IEEE Transactions on Visualization and Computer Graphics*, 5(3):238–250, July 1999.

[90] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P.-P. Sloan. Interactive ray tracing for isosurface rendering. *IEEE Visualization*, pages 233–238, 1998.

[91] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler. The VolumePro real-time ray-casting system. *ACM SIGGRAPH Computer Graphics*, pages 251–260, August 1999.

[92] H. Pfister and A. Kaufman. Cube-4 - a scalable architecture for real-time volume rendering. *ACM Symposium on Volume Visualization*, pages 47–54, 1996.

[93] H. Pfister, A. Kaufman, and T.-C. Chiueh. Cube-3: a real-time architecture for high-resolution volume visualization. *ACM Symposium on Volume Visualization*, pages 75–82, 1994.

[94] M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan. Rendering complex scenes with memory-coherent ray tracing. *ACM SIGGRAPH Computer Graphics*, 31:101–108, August 1997.

[95] T. Purcell, I. Buck, W. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. *ACM SIGGRAPH Computer Graphics*, 21(3):703–712, July 2002.

[96] T. Purcell, C. Donner, M. Cammarano, H. Jensen, and P. Hanrahan. Photon mapping on programmable graphics hardware. *SIGGRAPH/Eurographics Conference on Graphics Hardware*, pages 41–50, 2003.

[97] D. C. R. Yagel and A. Kaufman. Discrete ray tracing. *IEEE Computer Graphics and Applications*, 12(5):19–28, 1992.

[98] E. Reinhard, A. Chalmers, and F. Jansen. Hybrid scheduling for parallel rendering using coherent ray tasks. *IEEE Parallel Visualization and Graphics Symposium*, pages 21–28, 1999.

[99] E. Reinhard, B. Smits, and C. Hansen. Dynamic acceleration structures for interactive ray tracing. *Eurographics Workshop on Rendering*, pages 299–306, 2000.

[100] A. Riffel, A. E. Lefohn, K. Vidimce, M. Leone, and J. D. Owens. Mio: fast multipass partitioning via priority-based instruction scheduling. *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 35–44, 2004.

[101] S. M. Rubin and J. T. Whitted. A 3-dimensional representation for fast rendering of complex scenes. *Computer Graphics*, 14:110–116, July 1980.

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

[103] J. Schmittler, I. Wald, and P. Slusallek. Saarcor - a hardware architecture for ray tracing. *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 27–36, Sept 2002.

[104] K. B. Sebastian Fernandez and D. Greenberg. Local illumination environments for direct lighting acceleration. *Eurographics Workshop on Rendering*, pages 7–14, 2002.

[105] L. Sobierajski and A. Kaufman. Volumetric ray tracing. *Volume Visualization Symposium*, pages 11–19, October 1994.

[106] V. M. Spitzer, M. Ackerman, A. Sherzinger, and D. Whitlock. The Visible Human Male: A technical report. *Journal of the American Medical Informatics Association*, pages 118–130, 1996.

[107] T. Stark. Relative geologic time (age) volumes - relating every seismic sample to a geologically reasonable horizon. *The Leading Edge*, pages 928–932, September 2004.

[108] G. Stoll, M. Eldridge, D. Patterson, A. Webb, S. Berman, R. Levy, C. Caywood, M. Taveira, S. Hunt, and P. Hanrahan. Lightning-2: A high-performance display

subsystem for PC clusters. *ACM SIGGRAPH Computer Graphics*, pages 141–148, August 2001.

[109] Toshiba Medical Systems. Aquilion one: Worlds first dynamic volume CT scanner. $http://www.medical.toshiba.com/products/ct/aquilion\_one/$, 2007.

[110] T. Ullmann, A. Schmidt, D. Beier, and B. Brüderlin. Adaptive progressive vertex tracing in distributed environments. *Eurographics Workshop on Rendering*, pages 285–294, 2001.

[111] I. Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Universitat des Saarlandes, January 2004.

[112] I. Wald, T. Ize, A. Kensler, A. Knoll, and S. Parker. Ray tracing animated scenes using coherent grid traversal. *ACM Transactions on Graphics*, 25(3):485–493, 2006.

[113] I. Wald, T. Kollig, C. Benthin, A. Keller, and P. Slusallek. Interactive global illumination using fast ray tracing. *Eurographics Workshop on Rendering*, pages 15–24, 2002.

[114] I. Wald, T. J. Purcell, J. Schmittler, C. Benthin, and P. Slusallek. Realtime Ray Tracing and its use for Interactive Global Illumination. *Eurographics State of the Art Reports*, 2003.

[115] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive rendering with coherent ray tracing. *Computer Graphics Forum*, 20(3):153–164, 2001.

[116] L. A. Westover. *Splatting: a parallel, feed-forward volume rendering algorithm*. PhD thesis, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, 1991.

[117] T. Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, 1980.

[118] S. Woop, G. Marmitt, and P. Slusallek. B-kd trees for hardware accelerated ray tracing of dynamic scenes. *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 67–77, 2006.

[119] T. Yoo, U. Neumann, H. Fuchs, S. Pizer, T. Cullip, J. Rhoades, and R. Whitaker. Direct visualization of volume data. *IEEE Computer Graphics and Applications*, 12(4):63–71, 1992.