DISTRIBUTED VOLUME RENDERING OF VERY LARGE DATA ON HIGH-RESOLUTION SCALABLE DISPLAYS

BY

NICHOLAS SCHWARZ
B.S. University of Illinois at Chicago, 2003

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2007

Chicago, Illinois

To all researchers in the field of volume visualization.

ACKNOWLEDGMENTS

I want to thank Jason Leigh and Andy Johnson of the Electronic Visualization Laboratory for supporting me during the creation of this work. To my committee, Jason, Andy and Luc Renambot, thank you for your valuable guidance and feedback. I would also like to express my appreciation to Lance Long and Pat Hallihan for the tremendous amount of technical support they have provided during the creation of this thesis. Finally, to Lance, Pat, Julieta Aguilera, Atul Nayak, Raj Singh and Allan Spale, thank you for your moral support.

NS

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES (Continued)

FIGURE **PAGE**

LIST OF ABBREVIATIONS

CPU	Central Processing Unit
COTS	Consumer Off-the-Shelf
EVL	Electronic Visualization Laboratory
FPGA	Field-Programmable Gate Array
GPU	Graphics Processing Unit
NCMIR	National Center for Microscopy and Imaging Research
RAM	Random Access Memory
UCSD	University of California, San Diego
UMN	University of Minnesota
VRAM	Video Random Access Memory

SUMMARY

This thesis presents a methodology for rendering very large volume data on scalable high-resolution displays using a distributed-memory commodity cluster. The methodology uses a multi-resolution octree data structure, an image-order data distribution method, a distributed shared-memory data management system, a multi-level cache, and hardware accelerated rendering techniques to produce a solution that is scalable in terms of input data size and output resolution. An analytical cost model validated by experimental results predicts the system's behavior. The methodology's usefulness is demonstrated with a number of domain specific datasets.

The primary contributions of this thesis include:

1. A review of research in the field of volume rendering and parallel volume rendering.

2. A methodology for rendering very large volume data on scalable high-resolution displays using a commodity distributed-memory cluster of computers that scales with the size of input data and the output resolution.

3. An analytical model validated by experimental results that predicts the methodology's behavior.

4. The application of this methodology to domain specific problems in the fields of bioscience, geoscience and medicine.

CHAPTER 1

INTRODUCTION

The development of a direct volume rendering methodology that is scalable in terms of both input data size and output resolution is key to successfully visualizing spatially large volumetric datasets. The desire to increase image resolution is growing as users gain greater access to high-resolution displays, such as large desktop screens and scalable tiled-displays. While there is significant prior work in visualization techniques for large-scale data that can be viewed on single-node clients, and there has been work done in providing interactive visualizations of small data using tiled-display environments, there is still no solution that can handle large-scale input data and produce high-resolution output. This thesis advances the current state of the art by providing a distributed volume rendering methodology that scales in terms of both large input data and high-resolution output.

1.1 Research Merit

Direct volume rendering maps input data values directly to color values in an output image. It differs from other rendering methods, such as isosurface rendering, in that it does not produce an intermediate representation of the data. This technique has proved valuable to researchers in the bioscience, geoscience and medical communities, and is often chosen as a method to visualize spatially large and complex datasets.

Figure 1.1. The LambdaVision scalable high-resoltuion tiled-display.

High-resolution displays provide two main benefits to users. First, they allow users to view spatially large data in high-resolution that is at or near the native resolution of the original dataset. The LambdaVision display, shown in Figure 1.1, developed at the Electronic Visualization Laboratory (EVL) at the University of Illinois at Chicago (UIC) is one such display (1). Second, they allow groups of users to collaborate in the same space at the same time. The LambdaTable interactive display developed at EVL is such a display (2).

Direct volume rendering of large data on high-resolution displays is relevant to researchers in the bioscience and geoscience communities. Two example groups that currently use high-resolution displays to visualize volumetric data are bioscientists from the National Center for

Microscopy Imaging Research (NCMIR) at the University of California, San Diego (UCSD), and geoscientists from the University of Minnesota (UMN). Researchers at NCMIR employ scalable high-resolution displays to visualize microscopy imagery in an effort to discover relationships between biological structures from the anatomical to molecular scales. Researchers at UMN employ large desktop displays to visualize computed tomography (CT) data of geological samples in order to discover how certain processes take place. These two groups have a common need for interaction with spatially large volumetric data in a way that allows users to see the entire dataset.

1.2 Design Challenges and Strategy

An efficient and scalable solution for rendering large volume data on scalable high-resolution displays is dependent on system issues. As most researchers only have access to distributed-memory resources, the interconnect bandwidth between the nodes of a cluster is one of the biggest bottlenecks. Both of the current approaches for rendering large volume data are affected by this bottleneck during different stages of rendering.

This thesis presents a methodology for the distributed rendering large volume data on scalable high-resolution displays. The methodology uses an image-order based data distribution method, coupled with a shared-memory system and multiple cache levels to efficiently manage large volume data. A bricked data structure that stores multiple levels-of-detail and hardware accelerated rendering ensure interactive frames rates. An analytical performance model shows the scalability of the system and is validated by experimental results.

CHAPTER 2

PREVIOUS WORK

Volume renderers attempt to evaluate a common optical model that describes the interaction of light emitted, scattered or absorbed by particles that make up a dataset. A number of serial techniques have been developed to accomplish this task. Attributes of the optical model and associated image compositing theory allow for parallelization of the evaluation process. Parallel techniques that take advantage of these properties have been developed to increase performance and render larger datasets.

Most parallel volume rendering research concentrates on performance and large data. However, some research focuses on the task of producing high-resolution output. Research that specifically deals with high-resoltuion volume rendering has so far used approaches that take advantage of specially designed compositing hardware, or use consumer off-the-shelf (COTS) equipment, but replicate data across all rendering processors.

2.1 Optical Model

Optical models used in volume rendering consider the volume as a group of particles. Often the particles are sampled on a rectangular grid. When particles are sampled this way they are referred to as voxels. Particles can emit light, or alternatively, light from a source can be scattered or absorbed by particles. Realistic models that take all properties of illumination into account are very complicated and require considerable computational resources to solve.

Thus, practical models exist to simplify realistic models. An approximation that considers the emission and absorption of light is given by (3):

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

I_λ is the amount of light of wavelength λ coming from a ray direction r that is received at location x on the image plane. L is the length of ray r. The density of volume particles that receive light from the light sources and reflect it towards the image plane is given by μ. C_λ is the light of wavelength λ reflected at location s in the direction of ray r. In this approximation the reflected light is weighted by the particle's density. The exponential function attenuates the light scattered at s by the densities of the particles between s and the image plane.

Equation (2.1) cannot usually be computed analytically, so most volume rendering algorithms use a numeric solution. A Taylor series approximation of Equation (2.1) that keeps only the first two terms is given by:

$$I_\lambda(x,r) = \sum_{i=0}^{L/\Delta s} C_\lambda(s_i)\,\alpha(s_i) \cdot \prod_{j=0}^{i-1}(1-\alpha(s_j)) \tag{2.2}$$

$C_\lambda(s_i)$ represents the local color values derived from the illumination model. $\alpha(s_i)$ represents the opacity samples along a ray. The functions C and α are also referred to as the color and opacity transfer functions or maps, respectively. These functions are used to assign color and opacity to each scalar intensity value in the data.

2.2 Volume Rendering Techniques

Volume renderers attempt to solve the same basic approximation given in Equation (2.1). Various techniques implementing that approximation have been developed. Techniques are traditionally categorized based on the traversal order of data. The three basic categories are image-order, object-order and hybrid. Image-order techniques begin with pixels on the output image and determine the proper color contributions from voxels for those pixels. Object-order methods first consider voxels in the data and then compute a voxel's color contribution to points on the image plane. Hybrid methods combine various aspects of both image-order and object-order techniques.

Texture mapping techniques utilize texture mapping capabilities common on most commodity graphics hardware. Advanced GPU methods use features found on programmable graphics cards to implement various volume rendering techniques. Additionally, special purpose hardware has been designed specifically for volume rendering.

2.2.1 Image-Order Volume Rendering

Image-order techniques start by considering a point in the output image and then determine which voxels contribute to that point. This process is repeated for each point on the image. Ray casting is an image-order method that casts viewing rays from each point on the image plane through the data (4). Samples are taken at discrete intervals along the ray usually using trilinear interpolation. The samples are used to evaluate the optical model.

When the rays are traversed in front-to-back order an optimization technique, early ray termination, can be applied. As the opacity reaches the maximum value, the rest of the samples

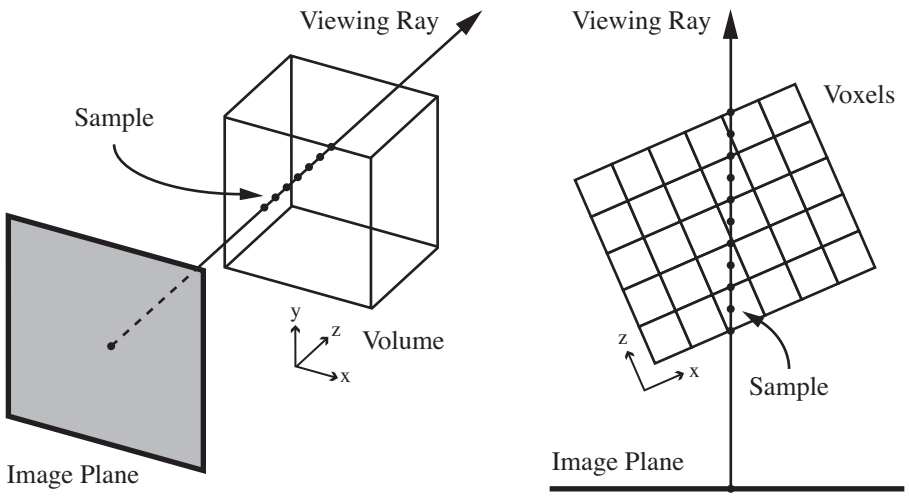


Figure 2.1. Volume rendering using ray casting. A ray starting at a point on the image plane is cast throught the volume to evaluate the optical model. Samples are taken at evenly spaced intervals along the ray. Because the data is discrete trilinear interpolation is used to determine the value of a sample.

along the ray can be ignored because they do not contribute to the final image. Another optimization strategy, empty space skipping, reduces rendering time by increasing the sample distance in sparse sections of the data.

2.2.2 Object-Order Volume Rendering

Object-order methods start by considering voxels in the data and then project their contribution to points on the image plane. Splatting is an object-order method that projects voxel footprints on the image plane (5). Reducing the amount of data traversed by skipping voxels that don't contribute to the output image is an inherent advantage to this approach. A disadvantage is that the color of hidden background objects may bleed into the final image.

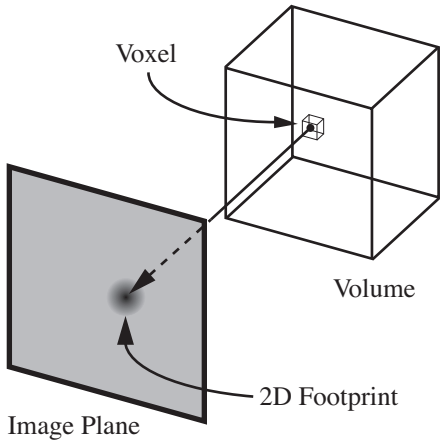


Figure 2.2. Volume rendering using splatting. The optical model is evaluated for each voxel and projected onto the image plane.

A method eliminating bleeding artifacts has been developed that processes voxels within a series a slabs aligned parallel to the image plane (6). Once a single slab has received all voxel contributions it is composited with the current image and the next slab is processed. Another optimization, early splat elimination, allows the algorithm to skip occluded voxels (7). However, this optimization is not as efficient as early ray termination is in ray casting.

2.2.3 Hybrid Volume Rendering

Hybrid volume rendering techniques attempt to combine aspects of both image-order and object-order methods. The shear-warp factorization is recognized as the fastest software renderer to date (8). High performance is achieved by using only 2D image operations instead of 3D image operations. It is based on a factorization of the viewing transformation into a shear

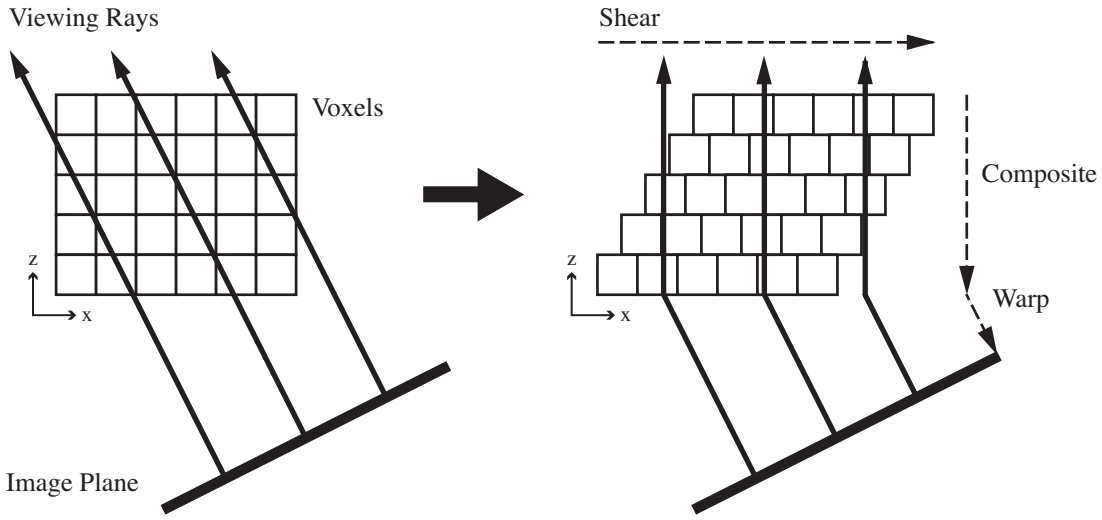


Figure 2.3. Volume rendering using the shear-warp factorization. Volume slices are sheared so that all viewing rays are parallel to the major viewing axis. After compositing, the distorted intermediate image is warped into the final image.

and a warp transformation. Both the volume and image can be traversed simultaneously because the shear transformation makes all the viewing rays parallel to the principal viewing axis. Compositing is then performed, followed by a warp transformation to create the final image. Run length encoding of the intermediate images and of the volume allow for early ray termination and empty space skipping, respectively. Unfortunately, because bilinear interpolation is used within parallel slices, image quality is reduced.

2.2.4 Texture Mapping Techniques

Texture mapping techniques utilize texture mapping capabilities common on most commodity graphics hardware. 2D texture mapping is one method that uses the texture mapping

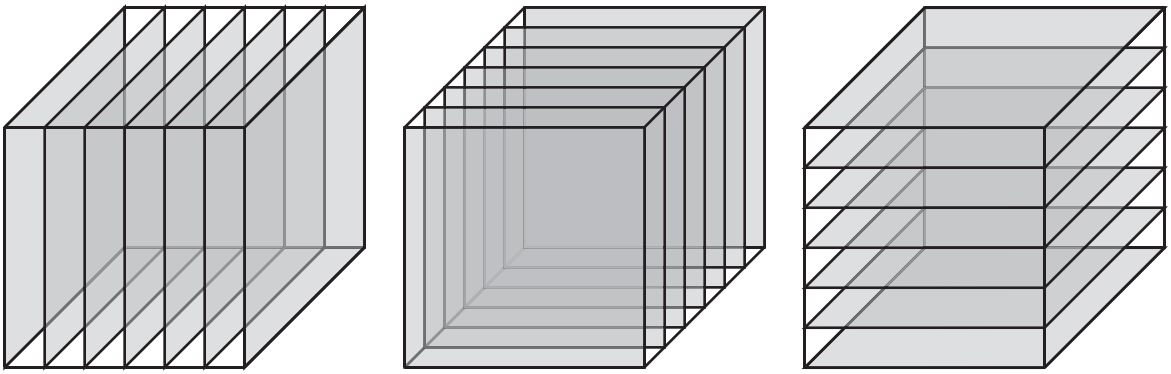


Figure 2.4. Axis-aligned proxy geometry for volume rendering. For each of the three major viewing axes, a set of planes is generated. During rendering the set corresponding to the axis most perpendicular to the viewing direction is chosen and rendered in back-to-front order.

capabilities found on many graphics cards (9). The operations performed are equivalent to the shear-warp factorization. Stacks of slices for each of the three major viewing axes are stored in memory, with the stack most parallel to the view plane used. The textures are mapped to object-aligned proxy geometry that is rendered in back-to-front order using alpha blending. As with the shear-warp factorization this method suffers from poor image quality because only bilinear interpolation is used within slices, storage space is increased to save three stacks of the data, and popping artifacts occur when different sets of stacks are selected.

3D texture mapping methods upload an entire volume dataset into texture memory (10). The hardware then maps the texture to proxy geometry, which may be axis-aligned slices such as in 2D texture mapping, view-aligned slices parallel to the image plane or viewpoint-centered spherical shells, which are concentric spherical shells centered at the viewpoint and culled to

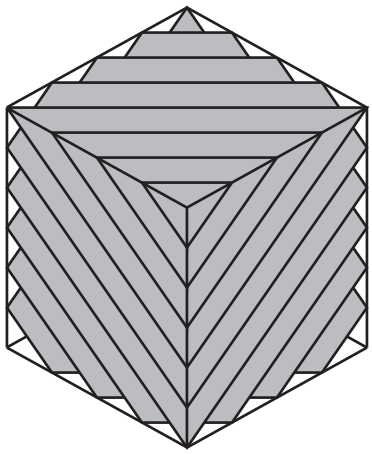


Figure 2.5. View-aligned proxy geometry for volume rendering. A set of planes parallel to the image plane is rendered in back-to-front order. The image plane is aligned to the plane of this page.

the frustum (11). The advantage of using view-aligned slices is the elimination of popping artifacts when the major viewing axis is changed. The advantage of using viewpoint-centered spherical shells as proxy geometry is that the distance from the viewpoint to a proxy shell is identical at all points on the shell. Of the three proxy geometries, viewpoint-centered spherical shells produce the highest quality images, and is the most desirable for perspective and stereo rendering. When using 3D texture mapping methods instead of 2D texture mapping techniques, image quality is improved regardless of the proxy geometry type because trilinear interpolation is performed by the hardware.

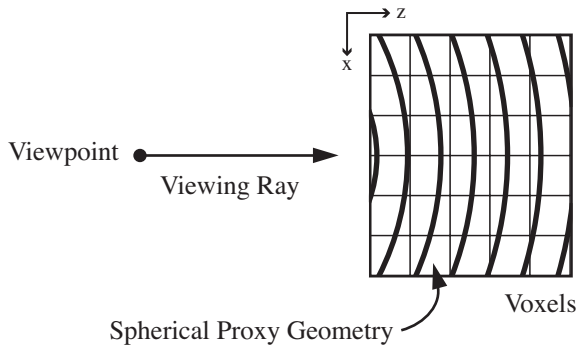


Figure 2.6. Viewpoint-centered spherical shells as proxy geometry. A series of concentric spherical shells centered at the viewpoint and culled to the frustum are used as proxy geometry.

2.2.5 Advanced GPU Methods

Modern GPU pipelines are becoming increasingly programmable. Initially, register combiners added the flexibility of manipulating the graphics pipeline's fragment processing stage. In more recent generations, programs called shaders or kernels can replace parts of the rendering pipeline that were previously fixed. NVIDIA's Cg (12) and the OpenGL Shading Language (GLSL) (13) are two languages currently used with programmable GPUs. Depending on hardware features, these languages allow the direct manipulation of vertex and fragment data.

These hardware features are used to enhance the 3D texture mapping technique (14). A color and opacity transfer function stored in a dependent texture can be applied using a fragment program. The fragment shader performs a texture lookup into the dependent texture retrieving the appropriate color and opacity value for that fragment. The retrieved value is applied to the

fragment. Proxy geometry is generated in hardware using a vertex program. Given the current model-view and projection matrices, and a bounding volume, the vertex shader computes the appropriate vertex and texture coordinates for either axis-aligned or view-aligned slices through the data.

Ray casting with empty space skipping and early ray termination has been implemented using programmable graphics hardware (15). A ray's entry point in the data is determined by rendering the front face of the volume's bounding box into a 2D texture. Rendering the bounding box's back face into another 2D texture and subtracting it from the front face's 2D texture gives the ray's direction. Samples are taken by performing texture lookups according to the parametric ray equation evaluated in a shader. The results are accumulated in another 2D texture. Sampling along a ray can be stopped when the accumulated opacity reaches a maximum value. The sample increment can be altered to skip empty space with the use of a pre-computed min/max octree.

2.2.6 Special Purpose Hardware

Dedicated hardware has been designed specifically for volume rendering. Most special purpose hardware focuses on implementing an image-order ray casting technique. The most commercially successful hardware dedicated to volume rendering is the VolumePro graphics card from Mitsubishi (16). VolumePro has support for parallel projections, gradient estimation, classification, per-sample Phong illumination, and clipping planes. It can render a 256^3 dataset at 30 frames per second. Another more versatile hardware volume rendering solution that supports both parallel and perspective projections is the VIZARD II system (17). VIZARD II

uses a field-programmable gate array (FPGA) that allows the addition of new features without costly hardware redesign. Its versatility limits its performance to about one-third that of the VolumePro.

2.3 Large Data

In their simplest forms, the volume rendering techniques assessed require access to the complete volume dataset. This means the data must be contained entirely in RAM if implemented on the CPU, or the data must fit in texture memory if implemented using graphics hardware. Modified approaches are used when data size exceeds the amount of available memory. The two primary approaches for dealing with large volume data involve data bricking and paging, and the use of compression.

2.3.1 Bricking and Paging

The OpenGL library provides automatic texture paging if texture memory limits are exceeded (18). A library for texture-based rendering called Volumizer uses this method (19). However, such brute-force methods for dealing with volumes whose sizes exceed physical memory severely degrade performance.

LaMar et al. (20) describe an adaptive bricking strategy based on a multi-resolution octree. Each level of the octree is half the resolution of the next level. The leaf nodes are associated with the original resolution, and the root node is associated with the coarsest resolution. Resolution is selected based on a user defined region-of-interest. The selected region is rendered with the finest resolution data. Coarser levels are used as the distance from the selected region increases. Opacity is adjusted as the sample distance changes at different levels. Viewpoint-

centered spherical shells are used as proxy geometry to limit artifacts that occur at brick boundaries. This approach assumes that all data selected for rendering can fit into available texture memory.

Weiler et al. (21) employ a similar approach using view-aligned slices as proxy geometry instead of spherical shells. They determined that brick boundary artifacts are caused by interpolation within bricks more so than by the choice of proxy geometry. Artifacts are eliminated, as long as adjacent blocks do not differ by more than one level, by sharing boundary values between adjacent blocks of the same resolution level, and by sharing a coarser levels boundary vales with an adjacent finer resolution brick. Automatic texture paging provided by most OpenGL implementations is used if the multi-resolution representation does not entirely fit into texture memory.

Octreemizer (22) is a system that uses an octree data structure coupled with a multi-level paging system and predictive cache to roam through large volumetric data. The multi-level cache operates between video memory and main memory, and between main memory and disk. Using a least-recently used replacement strategy, the predictive cache fetches data based on the direction of user movement. This system is limited to rendering 2D slices of the data, and only displays small portions of the data at a time.

2.3.2 Compression

An early attempt at rendering compressed volume data used a vector quantization compression approach combined with ray casting (23). The data is compressed using vector quan-

tization, producing a codebook. The codebook is used by the rendering system to decompress the data as the data is traversed. The final images are of reduced quality.

A compression method that takes advantage of modern graphics hardware uses preprocessed data encoded into a hierarchical structure using a multi-resolution covariance analysis of the original data (24). Using a programmable fragment shader, the compressed data is decoded and rendered on a graphics chip. This method has the advantage that datasets larger than texture memory can be loaded and rendered. Unfortunately, this approach uses nearest neighbor interpolation and suffers from degraded image quality.

Compressed hierarchical wavelet representations of data combined with data bricking produce high-quality images (25). The wavelet representation is decompressed on-the-fly and rendered using hardware texture mapping. Only the levels-of-detail necessary for display are decompressed and sent to the texturing hardware. This approach reclaims disk storage space but is limited by the amount of texture memory available as uncompressed bricks are transfered to the hardware.

2.4 Parallel Volume Rendering

Visualizing large-scale datasets is a very challenging problem. As the size of data increases, so does the rendering cost. In order to gain better performance, parallelizing the volume rendering process by increasing the number of processing units becomes an attractive solution.

2.4.1 Theoretical Model

Parallel volume renderers attempt to produce images equivalent to those produced by serial methods. Thus, parallel methods must evaluate the same approximation used by serial volume

rendering methods. Equation 2.1 can be decomposed into smaller units for parallel processing in two ways.

The first parallelization strategy decomposes work based on the screen position x. Computing the value of each position or groups of positions in the output image is assigned to different processing units. The final results are combined together to form a complete frame. This produces equivalent results because, according to Equation 2.1, the contribution to a position in the output image is independent of other positions in the image.

The second parallelization strategy is based on the image compositing work of Porter and Duff (26). They introduce the over operator which describes the process of compositing one image over another image using an alpha channel. The equation that determines the resulting output of placing image A over image B is given by:

$$c_{A\,over\,B} = c_B + c_A\,\alpha_A(1-\alpha_B)$$
$$\alpha_{A\,over\,B} = \alpha_B + \alpha_A(1-\alpha_B) \tag{2.3}$$

c_A and α_A are the color and opacity contributions, respectively, from image A. The color and opacity contributions from image B are given by c_B and α_B, respectively. Note that the resulting color is dependent on the color contributions from the two images as well as the alpha values.

For a group of images composited in a series the over operator can be written to describe the color and opacity contributions for steps along a ray as:

$$c_{i+1} = c_i + c_s\,\alpha_s(1-\alpha_i)$$
$$\alpha_{i+1} = \alpha_i + \alpha_s(1-\alpha_i) \tag{2.4}$$

c_{i+1} and α_{i+1} give the color and opacity contributions, respectively, to the final image at the next step along the integrated ray r. c_i and α_i represent the acculmulated color and opacity contributions, respectively, at step i along the ray r. The color and opacity contributions from the series of images at sample point s are given by c_s and α_s, respectively.

The over operator is associative; that is, for images A, B and C

$$A\,over\,(B\,over\,C) = (A\,over\,B)\,over\,C$$

The associativity of the over operator allows a renderer to break a ray into segments, process the sampling and compositing of each segment independently, and combine the results from each segment to form a final image.

2.4.2 Parallel Volume Rendering Techniques

Parallel volume renderers aim to solve the same basic approximation given in Equation 2.1 using the two parallelization strategies mentioned in the previous section. Parallelization strategies are commonly categorized as image-order, object-order and hybrid. Image-order parallel techniques take advantage of the fact that a contribution to a position in the output

image is independent of other positions in the image. These techniques break the output image into disjoint regions and assign a processing unit to render everything in that region. Object-order parallel methods are based on the over operator. These methods assign a processing unit to a section of data regardless of where the data appears in the final output image. After each section of data is rendered, a compositing step must construct the final image. Hybrid techniques combine aspects of both image-order and object-order methods.

Any of the methods for rendering volumetric data described earlier can be used to evaluate a task subdivided using these parallelization strategies. Subdivision may occur repeatedly using the same or a combination of parallelization methods. The optimal choice of parallelization technique and rendering method is heavily dependent on the implementation architecture.

The categorization of parallel volume rendering techniques shares many similarities to a common taxonomy of parallel geometry rendering methods (27). This taxonomy classifies algorithms based on where the sort from object coordinates to screen coordinates occurs in a standard graphics pipeline that consists of geometry processing followed by rasterization. In general, sorting can take place during geometry processing, between geometry processing and rasterization, or during rasterization.

All parallelization techniques can be described by a generic rendering pipeline. First, they decompose the data in some fashion. Next, the data is distributed to processors for rendering.

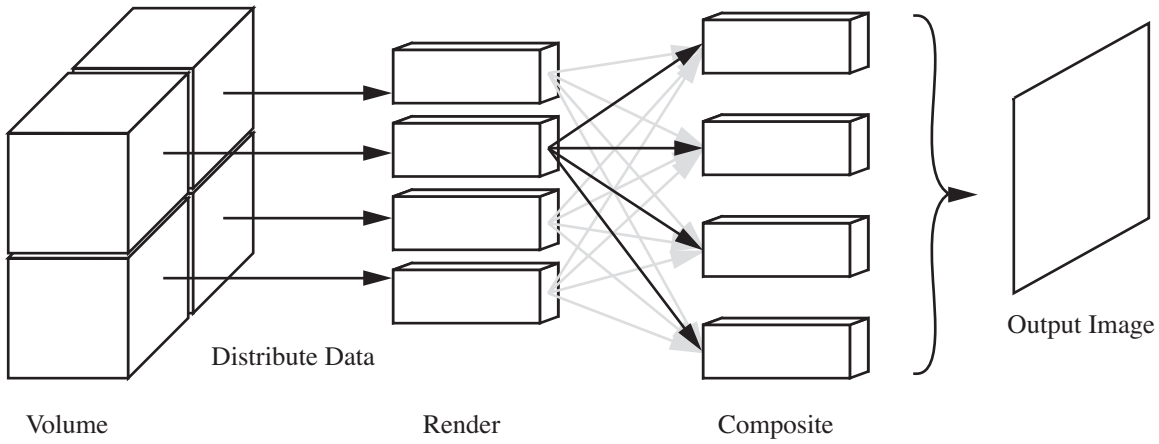


Figure 2.7. Generic parallel volume rendering pipeline.

Then the output of each rendering processor is composited to form the final output image. The total rendering cost can be expressed as:

$$t_{total} = t_{process} + t_{distribute} + t_{render} + t_{composite} + t_{display} \tag{2.5}$$

The cost of preprocessing data is given by $t_{process}$. $t_{distribute}$ gives the cost of distributing data to the appropriate rendering processors. t_{render} is the total rendering time. The cost of compositing, including network communication, is represented by $t_{composite}$. $t_{display}$ represents the time to redistribute the output image among appropriate display computers.

Sort-first methods distribute raw primitives early in the rendering pipeline to processors that can do the rendering. This is done by dividing the screen into disjoint regions and making processors responsible for all rendering calculations that affect their respective screen regions.

Sort-middle algorithms redistribute primitives between geometry processing and rasterization when primitives are in screen coordinates and ready for rasterization. Geometry processors are assigned arbitrary subsets of the primitives and rasterizers are assigned a portion of the display. Sort-last methods redistribute data at the end of the pipeline, after primitives have been rasterized into pixels or fragments. Each processor gets an arbitrary subset of primitives and computes pixel values no matter where they fall in the screen. The rendering processor transmits the pixels to compositing processors that resolve visibility. Sort-last requires an extra compositing step. Hybrid sorting methods combine various aspects of sort-first, sort-middle and sort-last methods.

Parallel image-order volume rendering methods are analogous to parallel sort-first geometry rendering methods. They both distribute raw data at the beginning of the process. Parallel object-order volume rendering techniques are analogous to parallel sort-last geometry rendering techniques. They both render arbitrary data and distribute portions of the final image that require compositing at the end of the process. There is no direct analogy between parallel sort-middle geometry algorithms and parallel volume rendering algorithms. Often, volume rendering literature uses these terms interchangeably.

2.4.2.1 Image-Order Parallel Volume Rendering Techniques

Parallel image-order techniques assign processing units to data or data to processing units based on a disjoint decomposition of the image plane. Each processing unit is responsible for rendering all data that falls within its screen section. Tiling together the results of each processor generates the final output image.

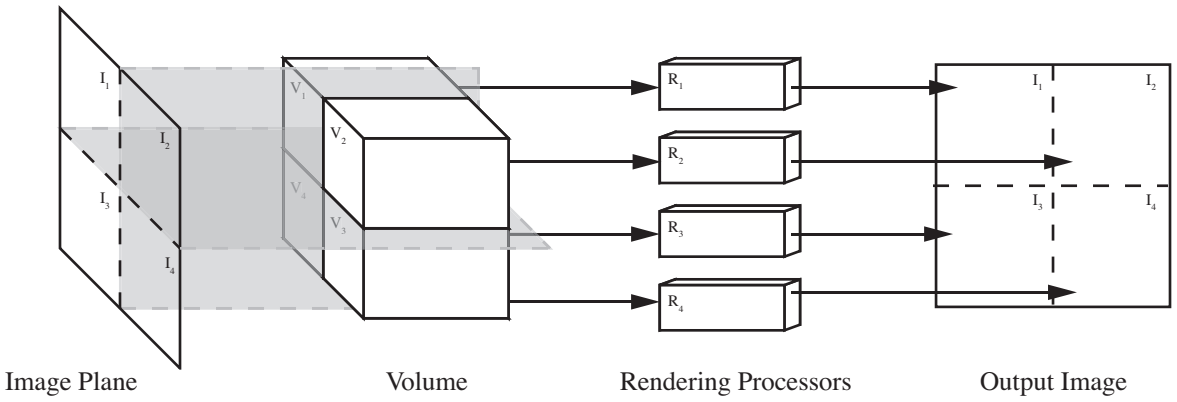


Figure 2.8. Image-order parallel volume rendering. Data is assigned to rendering processors based on a disjoint decomposition of the image plane. Each processor renders all data that falls within its disjoint screen region. The output image is created by tiling together the output of each processor.

The performance of the rendering phase can scale with the number of processors because every processor performs a rendering task equal to the inverse of the number of processors. However, before rendering can begin, data must be redistributed to each processor based on the current view. Therefore, an efficient data distribution mechanism must exist in order to have an efficient image-order parallel rendering technique.

Many parallel ray-casting methods have been designed for shared-memory supercomputing systems (28). These methods use shared-memory to simplify their data access task. Each processor is assigned the task of rendering a disjoint section of the display. The entire dataset is loaded in memory and each processor fetches data as needed. Multi-threaded systems that op-

erate on single processors, such as the Visualization Toolkit (VTK) (29), use a similar approach in which different threads are assigned different sections of the output image.

Amin (30) et al describe a parallel version of the shear-warp algorithm that operates on a distributed-memory CM-5 system. The data is first sheared and then partitioned across nodes based on the view. Sheared data is redistributed when the view changes. The disjoint output images are independently warped to produce the final output image. Run-length-encoding is used to reduce communication cost. Results show that there is very low communication because a small adjacent movement in the view only requires a small amount of data redistribution.

Bajaj (31) uses a compression-based scheme to render large volume data in a distributed-memory environment. Wavelet compression using zerobit encoding that performs well for random access operations is used. Compressed data is replicated locally on each node and decompressed as necessary. A ray-caster optimized with a min/max octree for early ray termination renders very small non-overlapping image portions on each node. The final image is collected on a separate computer.

Coelho implemented a distributed-memory version of the ZSweep algorithm (32). The ZSweep algorithm is a fast method that projects cells on an image plane by sweeping a plane parallel to the viewing plane in order of increasing depth. The distributed version randomly assigns regions of the output image to each processor. Data is distributed to the processors where it is required. Idle processors may receive unprocessed data from nodes that are busy. As with the previous method, the final image is tiled together on another machine.

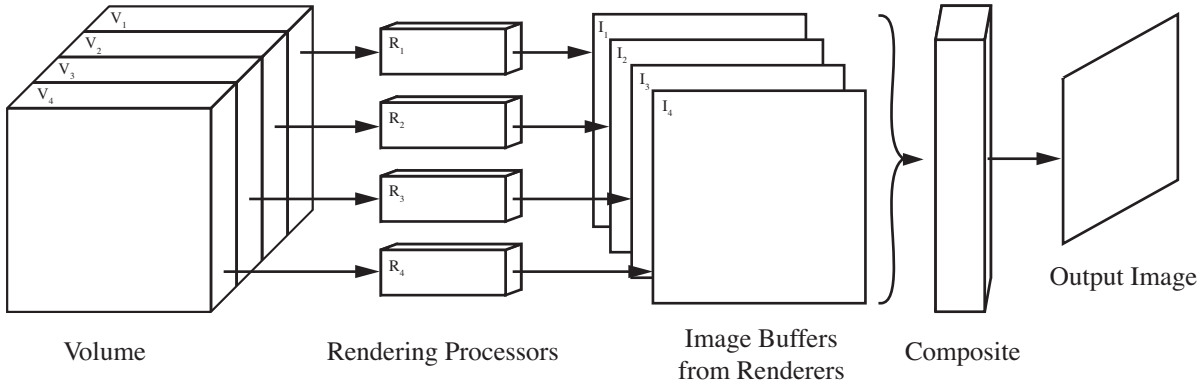


Figure 2.9. Object-order parallel volume rendering. Data is statically assigned to rendering processors without regard to where the data appears in the final output image. Each processor renders the data that it is assigned. Image buffers from each processor are composited together to form the final output image.

2.4.2.2 Object-Order Parallel Volume Rendering Techniques

Parallel object-order techniques assign a section of data to a processor based on an efficient or convenient data distribution method without regard for where the data appears in the output image. Each processing unit renders data it is assigned. Data is usually distributed once before rendering occurs and is never reassigned. Data must be composited together not only along the axes that lie on the image plane, but also in depth as explained by Porter and Duffs over operator.

The performance of the rendering phase can scale with the number of processors because every processor performs a rendering task equal to the inverse of the number of processors. However, the compositing task grows with the number of processors. Therefore, an efficient

compositing method must exist in order to have an efficient object-order parallel rendering technique. This final compositing phase is the focus of much research that has led to the development of four compositing methods: direct send, projection, binary-tree and binary-swap compositing.

The direct send compositing method sends partially composited results to a single computer (33). The computer accumulates all the color and opacity values for each pixel and computes the final pixel color once all processors have reported values. This method transmits about $n^{1/3}\,p\,(1-1/n)$ pixels, where n is the number of processors and p is the number of pixels in the output image. Direct send takes $O(n-1)$ stages, where n is the number of processors.

The projection compositing method propagates completed image segments in back-to-front order along the viewing direction (34). Because composition is performed sequentially, the entire rendering process can be pipelined. This method transmits $O(n^{1/3}\,p)$ pixels, where n is the number of processors and p is the number of pixels in the output image. The projection method takes $O(n^{1/3})$ stages, where n is the number of processors.

The binary-tree compositing method pairs all processors (35). One processor in every pair sends its subimage to the other processor in the pair. The processors that receive a subimage composite the subimage and proceed to the next stage. The number of idle processors increases at each stage. The total number of stages is $\log n$, where n is the number of processors.

The binary-swap compositing method is an improvement over the binary-tree method (36). In the binary-swap method, each pair of processors splits its image into two pieces, and exchanges one of the two pieces with the other in the pair. No processors become idle. This

method transmits $O(2.43\, n^{1/3}\, p)$ pixels, where n is the number of processors and p is the number of pixels in the output image. The total number of stages is $\log n$, where n is the number of processors.

A number of researchers have developed parallel volume rendering systems that use an object-order approach. Software systems that run on platforms ranging from a SGI Onyx4 to a 128 processor super-computer have been written. Two research groups have implemented hardware compositors based on the binary-swap algorithm.

Elvins developed a volume renderer for an nCUBE2 environment with 128 processors (37). Slices through the data along the major viewing axis are evenly distributed among all the processors. The processors then use splatting to produce images for their respective data. The master node is responsible for compositing the final image.

Muller et al. describes an object-order system that utilizes the direct send compositing algorithm (38). A ray-caster is implemented in the graphic cards fragment shader. Data is split among the processors based on a k-d tree. A load-balancing mechanism moves data between nodes based on the render time of the previous frame. Using an eight node cluster, the system can render a 512^3 dataset to a 1024x768 output window at sixteen frames per second.

Gribble et al. (39) present an object-order renderer, Rhesus, which uses binary-swap compositing. Implemented on an eight pipe SGI Onyx4, the system reports rendering a 512^3 section of the Visible Human dataset to a 256x256 output window at about ten frames per second. The authors results show that as the number of pipes increase, the compositing time begins to approach the time each pipe takes to render its section of data.

The Sephia (40) and Sephia-2 (41) architectures are hardware implementations of the binary-swap object-order compositing algorithm. Data is first distributed to a cluster of computers where each data chunk is rendered using a VolumePro graphics card. The output is read from the graphics card via the system bus and uploaded to FPGA-based blending hardware. Compositing hardware on each node is interconnected over Ethernet.

Frank and Kaufman used an eight-node cluster equipped with Sephia compositing hardware to visualize medical data (42). They implemented empty-space skipping by using a blocking scheme to divide data chunks based on a seed-growing algorithm. The system is capable of rendering the Visible Human dataset at three frames per second to a 1280x1024 output window.

Muraki et al present an object-order rendering system that operates on a cluster of computers (43). Data is distributed to each node where it is rendered using 3D textures. A hardware compositing system developed by Mitsubishi implements the binary-swap algorithm. A load-balancing scheme attempts to place equal numbers of visible voxels on each node based on the opacity transfer function. Their results show that this works well for a few nodes, but that the benefit decreases as the number of nodes increase. Using sixteen nodes, the system can render the Visible Human dataset to an output image of 768x768 at twenty frames per second.

2.4.2.3 Hybrid Parallel Volume Rendering Techniques

Hybrid parallel volume rendering techniques attempt to increase performance by combining aspects of both parallel image-order and parallel object-order rendering methods. Garcia and Shen describe a hybrid parallel rendering method that interleaves imge-order and object-order decompositions (44). Processors are first divided into groups. Each group is responsible for

rendering a portion of the volume data. Inside a group, the screen is divided into a set of pixels where each set is assigned to a processor. The pixels are assigned by alternating rows and columns. The subvolume assigned to a processor group is interleaved among processors in the group. Image compositing is done between pixels from different groups, and is not required for pixels within groups because processors within a group are assigned disjoint screen regions.

The amount of interleaving that occurs is determined by an interleaving factor. Compositing equivalent to the binary-swap algorithm occurs when there is no interleaving. As interleaving increases, the number of compositing stages decreases, and image artifacts increase due to interleaving pixel rows and columns. The extreme case in which interleaving is maximum is equivalent to a pure image-order partitioning scheme. A low interleaving factor can reduce the cost of compositing with only a moderate loss of image quality.

2.5 Volume Rendering on High-Resolution Displays

Most research in the field of parallel volume rendering has focused on performance and large data. Much less effort has been devoted to producing high-resolution output. The systems that do focus on parallel high-resolution volume rendering either use specialized compositing hardware, or replicate data on all rendering processors. The Sephia-2 architecture uses an object-order approach that is theoretically scalable to high-resolution displays. The FlowVR, Volume Visualizer and Vol-a-Tile systems all use image-order approaches to visualize data entirely resident in texture memory.

Lombeyda et al. state that the Sephia-2 object-order hardware compositing architecture theoretically scales in image space to support high-resolution displays. A new routability proof

that accounts for image sources contributing to more than one logical pipeline, and a dynamic frame reassembly property is required. However, no known implementations exist.

The TeraVoxel project (45), aimed at rendering a 1024^3 dataset at interactive frame rates, produced a solution for high-resolution displays called Volume Visualizer. Using a cluster equipped with four VolumePro cards, the system is able to render a 256x256x1024 volume to the 3840x2400 output window of a ten megapixel IBM T221 LCD. This is accomplished by replicating the entire dataset in all four nodes.

Vol-a-Tile (46) is a distributed direct volume rendering application designed for scalable, high-resolution displays. Users roam through large volume datasets that may be stored remotely, viewing small regions-of-interest in full resolution. The data within a region-of-interest is uploaded to the graphics card in every node where it is culled based on view and rendered using 3D textures. When the region changes, each node uploads identical copies of new data for the region. Vol-a-Tile is limited to viewing regions of data that fit entirely in the texture memory of a single node.

The FlowVR framework is a sort-first parallel rendering framework used to visualize volumetric data on high-resolution display walls (47). It uses a shader based protocol to implement hardware rendering algorithms. The system uses VTK and a shader based volume rendering class for generating graphics. This system is not scalable because data is replicated on all nodes of the cluster.

A specialized hardware system based on the binary-swap object-order compositing method scales in regards to large data. It may scale in terms of resolution, but a solid proof and

an implementation do not exist. Current approaches that scale in terms of resolution use image-order data decomposition. However, they do not scale with respect to data because they replicate data at each rendering processor. This thesis uses an image-order approach that scales with respect to both data and resolution using COTS hardware. Table I shows a comparison of current parallel high-resolution volume rendering systems to the system presented in this thesis.

TABLE I

COMPARISON OF PARALLEL HIGH-RESOLTUION VOLUME RENDERING SYSTEMS

System	Attribute			
	Large Data	High-Resolution	COTS	Approach
Sephia-2	Yes	No	No	Object-Order
Volume Visualizer	No	Yes	Yes	Image-Order
Vol-a-Tile	No	Yes	Yes	Image-Order
FlowVR	No	Yes	Yes	Image-Order
This Thesis	Yes	Yes	Yes	Image-Order

CHAPTER 3

METHODOLOGY

This chapter describes the fundamental methodology used by the volume rendering system. First, a set of requirements and their implications are defined and examined. Then a suitable design overview is presented. Next, the data structure used by the system is described. After that, the calculation used to determine the view-frustums for each tile of a scalable high-resolution tiled-display is derived. The volume rendering pipeline is then described in detail followed by the data management system. Finally, the methods used to maintain interactivity are presented.

3.1 System Design

The design of this system is based around a set of requirements that set its function and operating environment. These requirements lead to a number of implications about the most suitable approaches. Based on the requirements and their implications, a design overview is presented for rendering large volume data on scalable high-resolution displays.

3.1.1 Requirements

The development of a volume rendering solution warrants the consideration of specific application requirements. These requirements influence the methodology used to produce a solution. This system should render large data collected a priori, produce high-resolution output, allow

interactive manipulation of the view position and transfer functions, and operate on COTS systems.

Volume data is collected and processed before rendering begins. It may be collected from a scanning device such as a CT scanner, or a numerical simulation may generate it. Whatever the case, the data may be manipulated and transformed into a format more suitable for rendering.

The system must scale with respect to the size of data. The data is so large that it can neither fit entirely in the RAM of a single processor, nor can it fit entirely in the texture memory of a single graphics card. This requirement rules out the possibility of replicating data on multiple processors.

The system must generate output on scalable high-resolution displays. A high-resolution display is one capable of displaying at least one million pixels. Scalable high-resolution displays are tiled LCD or projector walls that are capable of displaying much more than one million pixels.

Users must be able to interact with the volume data they are rendering. The system must allow users to interactively manipulate the viewing transformation to see the data from different viewpoints. A mechanism must be provided to allow interaction with the color and opacity transfer functions.

In order to be useful to domain scientists, the system should run on a COTS platform. Distributed-memory clusters interconnected with a high-bandwidth network are the most common of such platforms. These systems often include graphics cards on each node.

3.1.2 Implications

The requirements explained above have a number of implications on the choice of techniques that are suitable. Preprocessing a priori data allows freedom in choosing a data format that compliments the interactivity requirement. Rendering large data that cannot be replicated across nodes, and producing high-resolution output on display tiles distributed among a cluster of computers influences the data distribution scheme used.

A priori data collection allows the system to use data in a format other than that of the output produced by the original collection or generation process. The data may be preprocessed into another format during an offline step. This format may be used to help reach other system goals, such as interactivity. A specific data format should also lend itself to easier rendering or data distribution given the implementation platform.

A distributed-memory cluster of computers fits the requirement of using a COTS system. In such a system, processors have their own local memory, not a globally accessible shared-memory pool. Individual processing nodes are interconnected via a high-speed network, such as Ethernet, not over a dedicated system bus. Any data distribution and composting algorithms must take this into account as it affects their performance and scalability.

Graphics cards connected to each processing node enable the system to take advantage of hardware accelerated rendering techniques. The amount of texture memory on most commodity graphics cards is significantly less than the amount of RAM available to each processor. Depending on the size of the data, rendering algorithms may have to swap data between RAM

and texture memory. Frame buffers on graphics cards may be restricted in size to due hardware limitations.

Supporting large data that is not replicated due to its size is a major challenge for many systems. For volume rendering systems in particular, the task of redistributing large becomes a bottleneck in the rendering pipeline. For this reason, many parallel volume rendering systems that operate on distributed-memory clusters utilize object-order data distribution techniques and generate the output image during a final compositing stage. This is in contrast to image-order techniques that redistribute the raw input data based on the view transformation. However, it can be shown that as the number of nodes increases, the cost of the final compositing stage always increases, while the cost of redistributing data using an image-order approach decreases. Because the data may exceed the combined RAM of all the processing nodes, a paging system between disk and memory is desirable. Image-order approaches also require a mechanism to redistribute data when view transformations change.

A cluster of computers usually drives a scalable high-resolution display. Each computer has a graphics card that is connected to one or two LCDs or projectors. Increasing the resolution of such displays involves adding more computers and LCDs or projectors. Very high-resolution scalable displays often consist of many computers. In order to display an image, an image must either be redistributed across the cluster, or it must be rendered directly to a tile.

Ten frames per second is considered an interactive frame rate. Since the data is large, it is conceivable that rendering a frame may take many seconds or even minutes. Thus, a level-of-detail technique should be used to allow fast interaction with a representation of the data. In

order to provide interactive manipulation of transfer functions the system should classify data during rendering, not during a pre-classification stage.

3.1.3 Overview

An appropriate solution to the problem of rendering large data on scalable high-resolution displays fulfills the requirements and considers the implications mentioned above. This design uses a parallel image-order approach and a distributed shared-memory system with paging and caching to distribute data. A pre-generated octree representation of the data complements the parallel approach used and aids with system interactivity. The system's renderer uses hardware accelerated 3D texture mapping techniques.

A parallel image-order volume rendering approach that scales in terms of both large data and high-resolution displays is appropriate. In this approach, each node renders and displays the data that falls within its view-frustum. This approach is scalable in terms of rendering large data on clusters that drive high-resolution displays.

In an image-order approach, the amount of data that each node requires when the view is updated decreases as the number of nodes increases, which is the case when more tiles are added to a display. Thus, the amount of time spent transmitting data based on the view transformation over an interconnection network decreases. In an object-order approach, in contrast, the amount of time spent during the compositing stage increases as the number of nodes increases.

Data redistribution time also decreases because image-order approaches inherently take advantage of image coherence. Often, interaction with a particular dataset is adjacent, that is,

it consists of small translations, scales, and rotations that do not greatly alter the view. Since the view does not change very much, only a small amount of data on each node needs updating, decreasing the amount of data redistributed.

This approach scales for high-resolution displays driven by large clusters for another reason. Since the output image is displayed in place on the node that renders it the final image does not have to be redistributed among the computers to which individual tiles are attached. In contrast, because object-order techniques use a compositing stage that collects the final image at one location they must spend time redistributing that image to the proper nodes for display.

A distributed shared-memory system serves as a mechanism to allow nodes access to the data that falls within their respective view-frustums. A paging system and multi-level cache ensure that large data that exceeds the amount of combined RAM on a cluster, as well as the amount of combined texture memory on all of a cluster's graphics cards can be rendered. The paging and cache system exist locally between texture memory and RAM and between RAM and disk.

Preprocessing data and storing it as an octree across the nodes of a cluster enables level-of-detail rendering for better interactivity, and provides a convenient and efficient data structure for view-frustum culling. Each successive level of the octree represents a higher-resolution version of the data. The system can reach interactive frame rates by rendering coarser levels while a user interacts and switching to more detailed levels when user interaction stops. Traversing an octree in a top-down manner discarding nodes that do not fall within the view-frustum has a logarithmic asymptotic time complexity.

Hardware rendering and classification using texture mapping techniques are widely supported by current graphics cards, produce high-quality images and aid interactivity. Hardware rendering using 3D textures mapped to view-aligned proxy geometry provides good image quality, and takes advantage of the capabilities of most current graphics hardware. The application of color and opacity transfer functions with a fragment shader allows for the interactive classification of data without reloading data textures every time a user reclassifies a data value.

Figure 3.1 shows an overview of the system's design. A commodity distributed-memory cluster equipped with graphics cards and a high-speed interconnect is the base platform. Subsequent sections describe the system in more detail.

Before the system starts, data is preprocessed into an octree. Data for each level is distributed evenly among the nodes of the cluster and stored on local disks. The first few lowest levels may be replicated across all nodes of the cluster, as they may not consume much storage space. The metadata associated with the octree is replicated on each processing node.

Next, the view-frustum for each node is computed. The frustum for each node is based on the view-frustum of the entire display. This is usually calculated only once when the system starts or may be calculated once for a display and stored.

During rendering, the user interactively determines the view-transformation. Culling the octree against each tile's view-frustum creates a list of all bricks to be rendered. The list is sorted level-by-level, and the in back-to-front order within each level. The renderer then renders each brick on the list, swapping buffers after each level is completed. Rendering is aborted and restarted at the lowest level if user interaction occurs before a level is completed. In order to

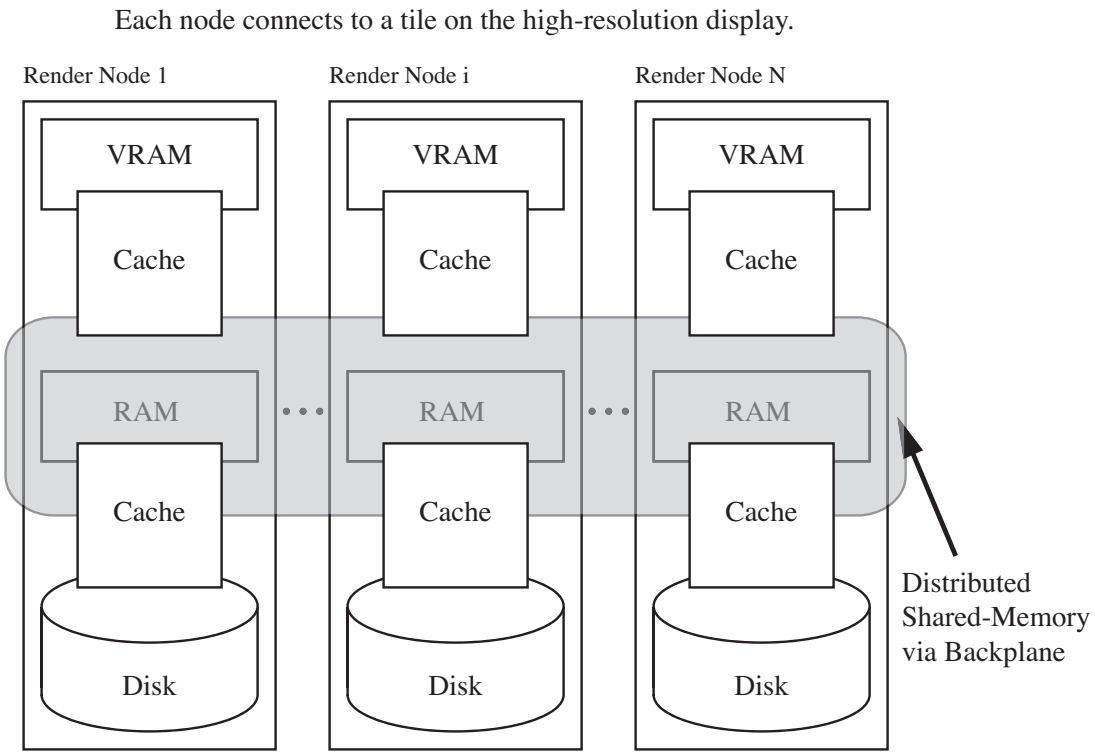


Figure 3.1. System overview diagram. The base platform consists of a commodity distributed-memory cluster equipped with graphics cards and connected via a high-speed network. The area shaded in grey depicts the distributed shared-memory system connected over the cluster's high-speed interconnect.

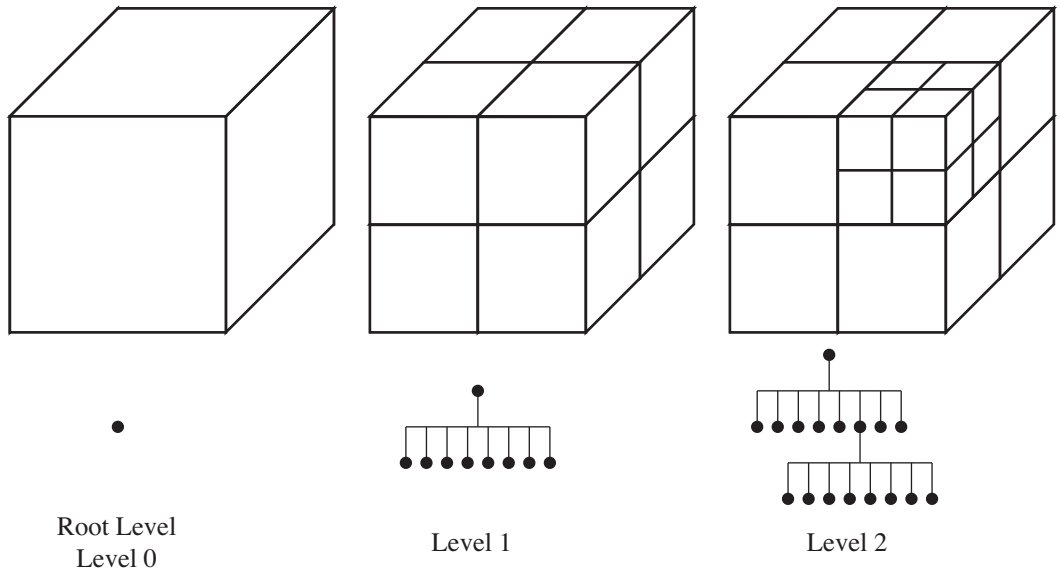


Figure 3.2. Illustration of an octtee. An octree is a tree data structure based on a node with eight children. Each node of an octree represents a region in space. Each child represents an octant of its parent.

access a brick for rendering, the renderer first searches in local texture memory cache. If not present, the distributed shared-memeory system is queried. The brick may be loaded from a node's local disk if not present in either of the two caches.

3.2 Octree Data Structure

An octree is a tree data structure based on a node with eight children. Each node of an octree represents a region in space. Each child represents an octant of its parent. The total number of nodes in an octree is $\sum_{i=0}^{L} 8^i$ where L is the number of levels in the octree starting at 0. See Figure 3.2.

Each level of the octree is associated with a different level-of-detail representation of the original data. The leaf nodes are associated with the original, highest resolution data. The root node is associated with the coarsest resolution data. Data associated with each level of the octree can be computed in any appropriate way. This system uses linear interpolation based on the highest resolution data to create coarser levels. The voxel dimensions of data represented by a node must be a power of two, and all nodes must have identical voxel dimensions. It follows that the voxel dimensions of each successive level are a power of two greater than those of the previous level.

Nodes in the octree represent distinct regions in space. Each of the eight vertices of a node have a world-space coordinate associated with it. A child node's vertices have coordinates appropriate to their position within their respective parent node in world-space coordinates. See Figure 3.3

In order to avoid interpolation artifacts at brick boundaries when rendering, the boundary voxels of nodes within a level are repeated between adjacent nodes. A node's world-space coordinates are adjusted to consider this overlap. See Figure 3.4 for an illustration.

3.3 Frustum Calculation

The view-frustum for each tile on a scalable high-resolution display is calculated as a fraction of the entire display's frustum. If the display has mullions that are to be considered they must be accounted for in the calculation. The view-frustum for a tiled-display is determined by:

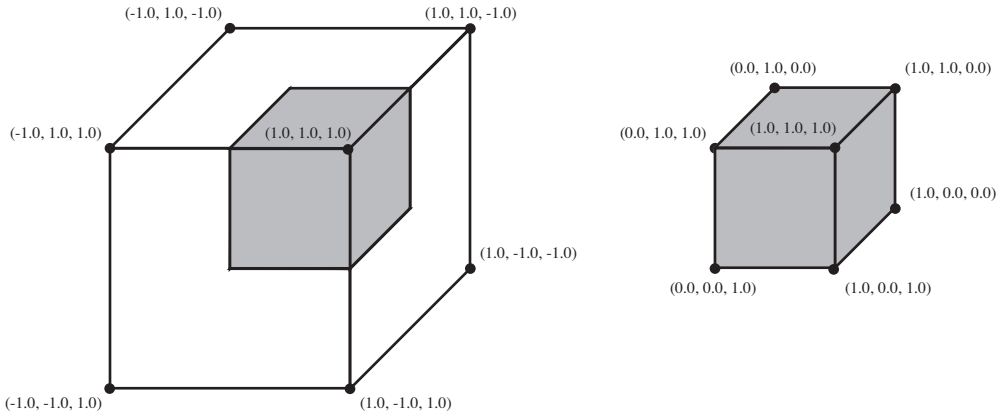


Figure 3.3. Assignment of world-space coordinates. Each of the eight vertices of the parent node on the left have a world-space coordinate associated with it. The vertices of the child node on the right have coordinates appropriate to their position within the parent node in world-space coordinates.

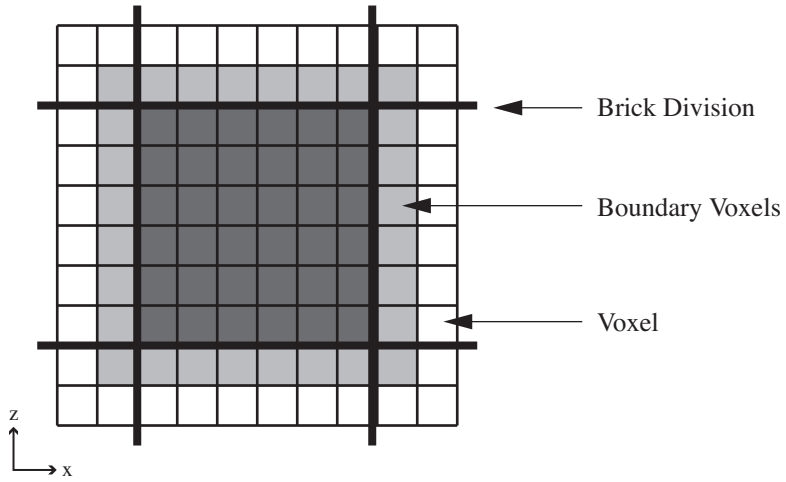


Figure 3.4. Sharing boundary voxels. Boundary voxels are shared among adjacent nodes within a level in order to avoid interpolation artifacts.

$$
\begin{aligned}
A &= \frac{Width_{Display}}{Height_{Display}} \\
T &= Clip_{Near} \cdot \tan\left(\frac{FieldOfView_{Vertcal}}{2}\right) \\
left &= -A \cdot T \\
right &= A \cdot T \\
top &= T \\
bottom &= -T
\end{aligned} \tag{3.1}
$$

The total width and height of the tiled-display is given by $Width_{Display}$ and $Height_{Display}$, respectively. The vertical field-of-view from the viewpoint is given by $FieldOfView_{Vertical}$. $Clip_{Near}$ is the distance from the display to the near clipping plane. A is the computed aspect ratio of the display. The view-frustum is determined by $left$, $right$, top and $bottom$. See Figure 3.5.

3.4 Rendering

Rendering is the process of generating an image from a model. In order to render data, each node must first determine which data from the octree falls within its frustum. The bricks that are entirely within or intersect the frustum are placed in a list and sorted in an appropriate order. Each brick is then rendered in that order. The framebuffer is swapped after every level-of-detail is fully rendered.

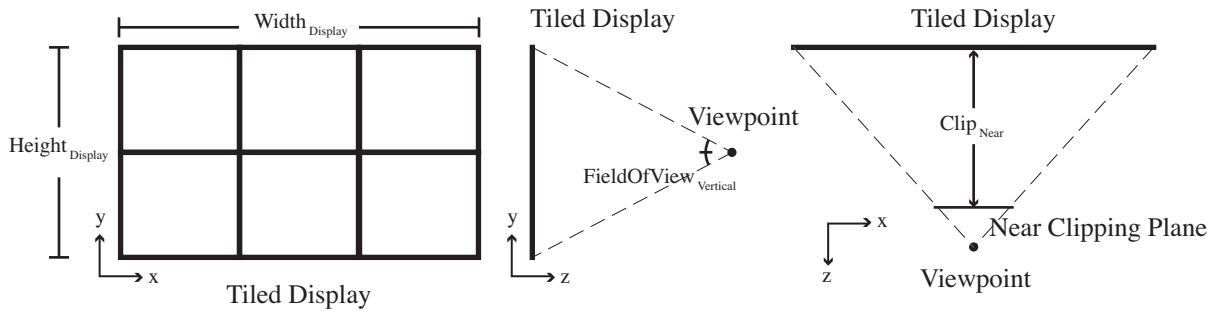


Figure 3.5. Calculating the frustum of a scalable high-resolution tiled-display.

Each processor implements a complete rendering pipeline because each node is responsible for fully generating a portion of the final output image. The three main stages in a volume rendering pipeline are reconstruction, classification and compositing. Reconstruction is the process of constructing a continuous function from the dataset. Classification is the process of assigning a color and an opacity value to a data value. Compositing determines the contribution of a classified sample to the final image. Figure 3.6 shows the local volume rendering pipeline.

3.4.1 Visibility Culling

Visibility culling identifies what is totally or partially inside the view frustum and culls everything that is not inside. This can improve the performance of the application since only those objects that are visible are processed. In the case of rendering on a scalable high-resolution display, each node separately culls the scene against its frustum to determine the objects it must render.

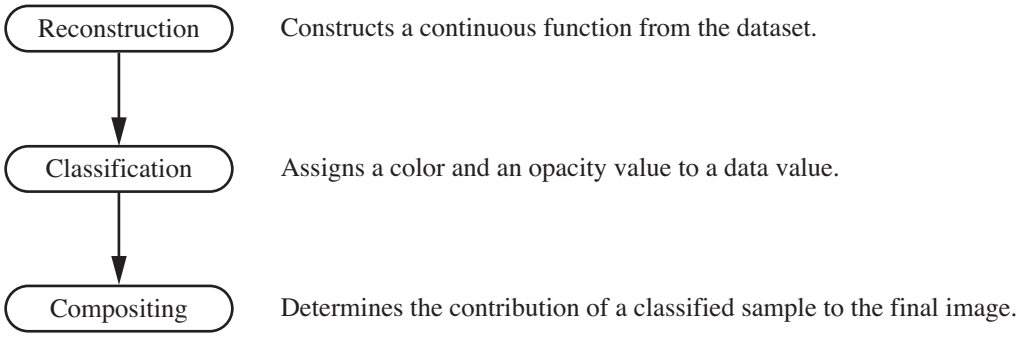


Figure 3.6. Local volume rendering pipeline.

In this case, the objects in the scene are the nodes of an octree. Each time the view is updated the system traverses all levels of the octree and creates a list of nodes that are potentially visible for rendering. Figure Figure 3.7 illustrates this concept.

The octree traversal algorithm uses a breath-first search (BFS). Figure Figure 3.8 contains pseudo code for this algorithm. First, the root node is checked against the frustum. If it's visible it is added to a queue. Next, the node at the front of the queue is removed and placed in the list of visible nodes. Each of the node's children is checked against the frustum. If a child node is visible it is added to the queue. The process repeats by removing the next node from the queue and examining it, and continues until the queue is empty.

Testing whether a brick falls within a frustum is done geometrically. Each of the six planes that bound the frustum are calculated. Then, each of the node's eight vertices is tested against

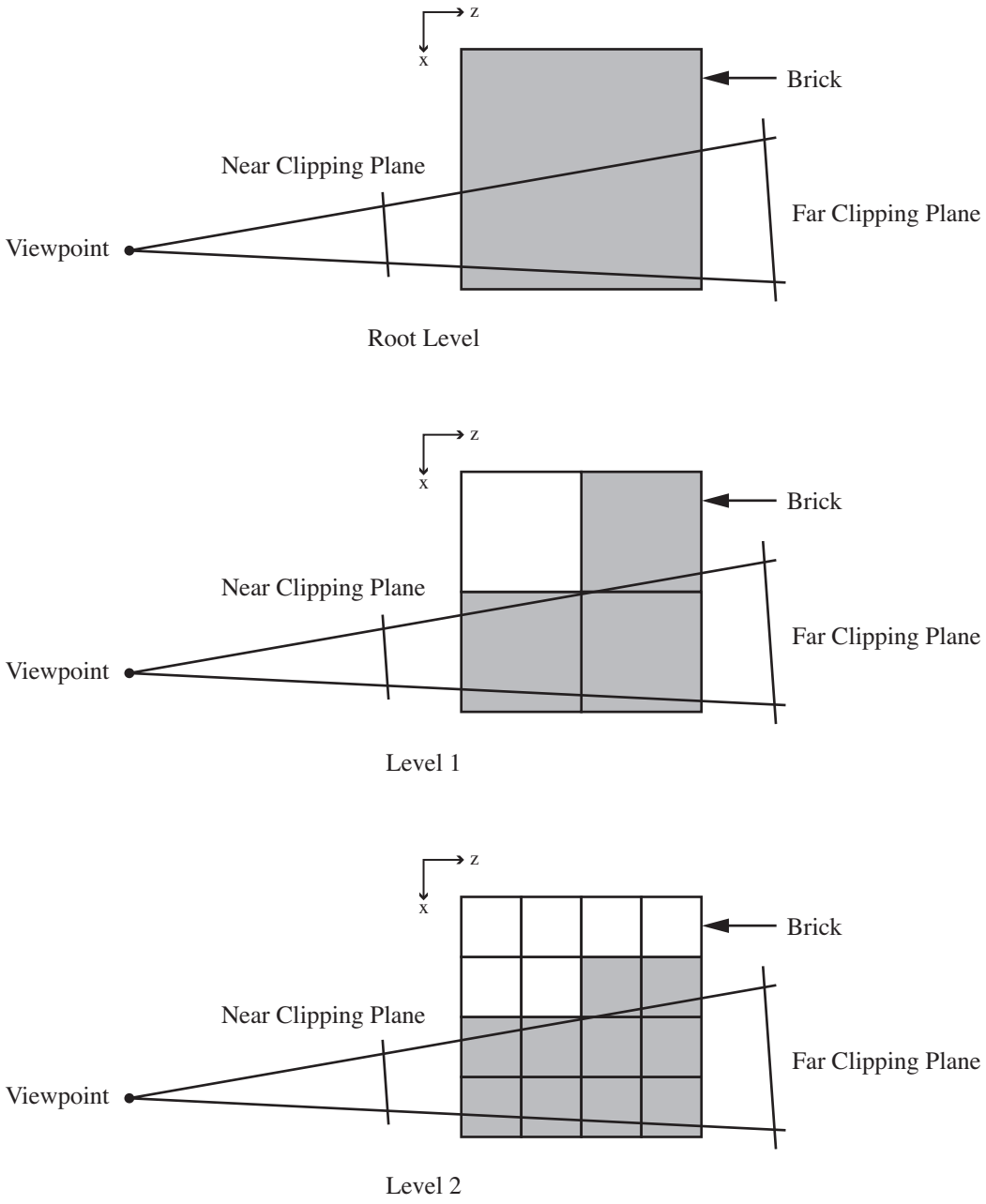


Figure 3.7. Illustration of visibility culling. Three levels of an octree are shown. Grey bricks are selected for rendering because they fall within the frustum. White bricks are culled.

OctreeTraversal

1. Initialize a queue, Q, and a list, L.
2. If the root node is inside the frustum, add it to Q.
3. Remove a node, N, from the beginning of Q and add it to L.
4. For each child, C_i, of N:
   If C_i is inside the frustum, add it to Q.
5. If Q is empty, all visible nodes are in L. Stop.
6. Repeat from step 2.

Figure 3.8. Octree traversal algorithm for visibility culling.

each of the frustum's six planes. Based on these tests a node is determined to be either inside or outside a frustum.

3.4.2 Brick Sorting and Placement

The list of bricks produced by the frustum-culling step cannot be rendered in an arbitrary order. In order to composite correctly, they must be rendered in series constrained by the associative property shown in Equation (2.5). Furthermore, bricks from all levels of the octree are present on the list. Thus, bricks are sorted level-by-level starting at the lowest level, and in back-to-front order within each level.

Each brick represents the contents of a different spatial region within the volume. The location of the region is given by the brick's boundary vertices. These vertices along with the current model-view matrix and level within the octree are used to appropriately place the brick in the scene in relation to the other bricks.

GenerateView-AlignedProxyGeometry

1. Using the model-view matrix, transform the volume bounding box vertices into view coordinates.
2. Compute the number of sampling planes used between the minimum and maximum z coordinates of the transformed vertices. This is based on the number of voxels and the desired sample rate.
3. For each plane in back-to-front order:
   Add each intersection of the plane with the edges of the bounding box to a list.
   Sort the plane vertices clockwise.

Figure 3.9. Generating view-aligned proxy geometry.

3.4.3 Reconstruction Using View-Aligned 3D Texture Mapping

Volume rendering using texture-mapping techniques performs the sampling and composting steps necessary to generate an image by rendering a set of 2D geometric primitives inside the volume. Each 2D geometric primitive is assigned texture coordinates for sampling the 3D volume texture. This proxy geometry is rasterized and blended into the frame buffer, usually in back-to-front order.

In order to render a data brick, it is first uploaded to the graphics card and stored as a 3D texture. Next, view-aligned proxy geometry is generated. This is computed on the CPU from the computed vertex positions and the transformed volume bounding box. Appropriate texture coordinates are also computed on the CPU and interleaved with the vertex coordinates. Figure 3.9 gives an algorithm for generating view-aligned proxy geometry for volume rendering.

3.4.4 Opacity Correction

The assigned opacity depends on the number of samples taken, and the level of the octree being rendered. When taking fewer samples, the opacity has to be scaled up so that the overall image intensity remains unchanged, instead of being reduced. Conversely, when more samples are taken, the opacity has to be scaled down so that the image buffer does not become oversaturated.

This occurs when the user changes the sampling rate via a control in the user interface, or whenever the renderer switches to a different level-of-detail. Because each brick in the octree has the same voxel dimensions, the same number of sample planes are generated for each brick rendered. Thus, the number of samples orthogonal to the view-plane, the sample rate, doubles at each successive level. Image intensity will vary as each progressive level is rendered, unless the opacity is corrected at each level.

The corrected opacity value based on the sampling rate is given by:

$$\alpha = 1 - (1 - \alpha_o)^{\frac{s_o}{s}} \tag{3.2}$$

α gives the new opacity value, and α_o represents the original reference opacity value. s is the new sample rate, and s_o is the original reference sample rate.

3.4.5 Classification

Classification is the process of assigning a color and opacity to a reconstructed function value. Transfer functions, usually implemented as lookup tables, are used for this purpose. A

```
void Fragment(in float3 inTex : TEXCOORD0,
              out float4 outColor : COLOR0,
              const uniform sampler3D DataTexture,
              const uniform sampler2D MapTexture) {
   float2 val = tex3D(DataTexture, inTex).ar;
   outColor = tex2D(MapTexture, val);
}
```

Figure 3.10. Fragment program for one-dimensional lookup tables. This fragment program written in Cg supports a one-dimensional color and opacity lookup table for 8-bit and 16-bit data using a 2D dependent texture.

reconstructed value determined during rendering is used as an index for the lookup tables that contain corresponding color and opacity values.

This system supports a one-dimensional lookup table for 8-bit and 16-bit integer voxels.. It is implemented using a 2D dependent texture to hold a one-dimensional lookup table and the fragment shader shown in Figure 3.10. The texture coordinate of the sample point to be reconstructed, along with the 3D texture representing the data and the 2D texture representing the lookup table are provided to the fragment shader. The shader computes the reconstructed data value, and then calculates the appropriate color and opacity value based on the 2D dependent texture.

3.4.6 Compositing

Compositing between nodes is not necessary because the system uses an image-order data decomposition approach where each node is responsible for rendering all the data falls within it

view frustum. However, compositing does occur locally on each node to accumulate the color and opacity results of the view-aligned proxy geometry sorted in back-to-front order.

Compositing takes place entirely in the graphics hardware. In OpenGL, the way the colors of the incoming source fragment is combined with those already stored in the framebuffer is specified by a blending function. For consistency with the compositing function given in Equation 2.3 and with rendering in back-to-front order, this system uses `GL_SRC_ALPHA`, and `GL_ONE_MINUS_SRC_ALPHA` as the source and destination functions, respectively.

3.5 Data Management

The data management system is critical to the system's ability to render large data on a distributed-memory cluster. The system uses a memory system similar to distributed shared-memory architectures. It keeps track of data bricks loaded in RAM across all nodes of the cluster. The system may request data to be transferred between nodes.

Image coherence argues that changes to the view-transformations are often slight, resulting in only small amounts of new data being loaded by each node per frame. A multilevel cache system is used to increase performance by keeping the most recently used data bricks as close to the graphics hardware as possible. The first level of the cache exists locally between the texture memory on each node and the memory system. The second level exists locally between the memory system and the local storage system on each node. Both cache levels employ the least-recently-used (LRU) replacement strategy.

3.5.1 Texture Memory Cache

The texture memory cache keeps as many of the most recently used bricks resident in texture memory as possible. When the rendering component requests a brick from the data management system, this is the component that first gets the request. The number of bricks in the cache is determined by the size of texture memory allotted to the system divided by the brick size. The user sets the amount of texture memory allotted to the system.

When the texture cache system receives a request for a brick it first checks the bricks currently in the cache. If the brick is present, it is moved to the top of the cache and the renderer is allowed to proceed. If the brick is not present in the cache a request for the brick is sent to the memory system. When the memory system fetches the brick it is loaded into texture memory, placed a the top of the cache and the renderer is allowed to continue. This process may push the least-recently-used brick out of the cache. Figure 3.11 illustrates this process with an example.

3.5.2 Memory System

The memory system keeps track of all bricks loaded in the local RAM on every node. Any node can query the memory system for any brick using the bricks ID value stored along with the octrees metadata. If a brick is present on another node, the memory system transfers it to the requesting node without changing the cache on the source node. In the event that a brick is not present in the memory system, it is temporarily loaded onto the node where it resides on disk and transferred to the requesting node. The caches are not altered on the node that provided the brick.

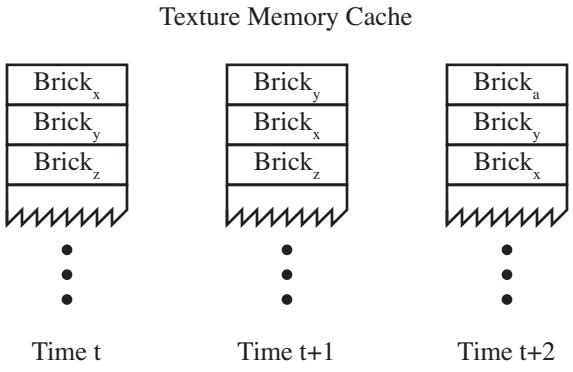


Figure 3.11. Illustration of texture memory cache operations. The initial state of the cache at time t is shown on the left. The cache contains bricks Brick_x, Brick_y and Brick_z. The rendering system requests Brick_y at time $t + 1$. It is present in the cache, so it is moved to the top. At time $t + 2$ the system requests Brick_a. It is not present in the cache, so the system requests it from the memory system. Once it is received it is placed at the top of the cache.

3.5.3 Memory System Cache

The memory system keeps as many of the most recently used bricks requested by an individual node in the local RAM of the requesting node as possible. When the rendering component fails to locate a brick in its texture cache, the request for the brick goes to the memory cache system. The number of bricks in the cache is determined by the size of RAM allotted to the system divided by the brick size. The user sets the amount of RAM allotted to the system.

When the memory cache system receives a request for a brick, it first checks the bricks currently in the cache on the local node. If the brick is present, it is moved to the top of the cache and a pointer is given to the requestor, that is, to the texture cache system. If it is not present locally, a query is made to the memory system, which looks for the brick on other nodes

and the disk system. The brick is then transferred to the local node where it is placed at the top of the cache. This cache like the texture cache system may push the least-recently-used brick out of the cache.

3.6 Maintaining Interactivity

Interaction is very important in a volume visualization system. The user must be able to freely navigate the data by panning, scaling and rotating. In addition, the user must be able to interactively change the color and opacity transfer functions. The system enables this with the use of three mechanisms: adaptive level-of-detail rendering, user determined sampling frequency, and interactive classification during rendering.

The adaptive level-of-detail mechanism automatically switches to using the coarsest resolution data from the octree whenever the user interacts. This allows the user instant feedback based on navigation without waiting for a dataset to be fully rendered. When the user stops navigation, the system progressively renders higher resolution data from successive levels of the octree giving the user an image of continuously increasing detail.

The user can change the number of samples taken by modifying the sampling frequency. Reducing the sampling frequency decreases the number of proxy geometry slices though the data. Decreasing the number of slices reduces the number of texture lookups that are performed, thus decreasing the total rendering time. This increases performance at the cost of reducing image quality. Conversely, the sampling frequency can be increased to produce higher resolution images.

Classification is performed interactively during the rendering stage. One-dimensional color and opacity transfer functions are applies during the rasterization stage using a fragment shader. Applying the lookup table at this stage allows reclassification of the data without modifying the original data values and reloading the data textures every time the transfer functions are modified, which can take considerably more time than uploading a new lookup table.

CHAPTER 4

PERFORMANCE ANALYSIS AND RESULTS

Testing the system determines the scalability of the methodology in terms of data size and output resolution. An analytical cost model shows the system is scalable for a wide range of input data sizes, output resolutions and cluster sizes. Performance tests run on a subset of the scenarios used in the analytical model's evaluations validate the model's predications.

4.1 Theoretical Performance Analysis

A theoretical analysis of the methodology provides a mechanism to evaluate the performance of the system under ideal conditions and to predict the system's behavior. An analysis of this methodology with some assumptions about the implementation platform produces results that show the system's scalability. The analytical cost model scales to produce larger output images from larger data sizes as the number of processing nodes increases.

4.1.1 Analytical Cost Model

The performance of the methodology presented in this thesis can be described at a high level by (Equation 2.5). As a reminder, (Equation 2.5) states:

$$t_{total} = t_{process} + t_{distribute} + t_{render} + t_{composite} + t_{display}$$

The cost of preprocessing data is given by $t_{process}$. $t_{distribute}$ gives the cost of distributing data to the appropriate rendering processors. t_{render} is the total rendering time. The cost of

compositing, including network communication, is represented by $t_{composite}$. $t_{display}$ represents the time to redistribute the output image among appropriate display computers.

In this case, the cost of preprocessing data, $t_{process}$, is not considered because it is done only once for a given dataset. Because this is an image-order based method without a compositing step, $t_{composite}$ is zero. $t_{display}$ is also zero as each node in the cluster is assumed to display the data it renders. This leaves the equation:

$$t_{total} = t_{distribute} + t_{render}$$

The time to distribute data and the time to render are considered together. The worst-case occurs when all of the data must be redistributed among all the rendering nodes. In this case, any benefit that may be gained by using a multi-level cache, or that may inherently exist due to image-coherence does not improve performance. The total time to render a frame in the worst-case is given by:

$$\begin{aligned} t_{total} = & \frac{D/N}{S_{VRAM}} \cdot \Big((S_{VRAM} \cdot B_{RAM\,to\,VRAM}) + (S_{VRAM} \cdot B_{RAM\,to\,RAM}) + \\ & (S_{VRAM} \cdot B_{disk\,to\,RAM}) + t_{render}\left(S_{VRAM}, \frac{p}{N}\right) \Big) \end{aligned} \tag{4.1}$$

The total number of nodes in the cluster is given by N. The total size of the data is given by D. p is the total number of pixels in the output image. S_{VRAM} is the amount of texture memory on each graphics card. $B_{RAM\,to\,VRAM}$ is the bandwidth between RAM and texture memory. The bandwidth between disk and RAM is given by $B_{disk\,to\,RAM}$. The bandwidth

between the memory of two nodes of the cluster is given by $B_{RAM\,to\,RAM}$. $t_{render}\left(S_{VRAM}, \frac{p}{N}\right)$ is a function that gives the cost of rendering data of size S_{VRAM} to p/N pixels.

(Equation 4.1) assumes that each node must fetch its data from the local disk of a remote node. It also assumes that this is performed in blocks equal to the size of texture memory. The total time to render each block is given by the time to load it from disk to memory, transfer it to the local node from a remote node, upload it to the graphics card, and render it in the graphics card. The product of this time by the number of blocks that each node must render gives the total rendering time. The number of blocks that each node must render is given by $(D/N)/S_{VRAM}$.

The term $(S_{VRAM} \cdot B_{disk\,to\,RAM})$ can be omitted when the data is large enough to fit entirely in the combined memory of each computer. This occurs when $D \leq S_{RAM} \cdot N$. If the data can be replicated in the memory of each computer, the term $(S_{VRAM} \cdot B_{RAM\,to\,RAM})$ becomes zero. This happens when $D \leq S_{RAM}$. The term $(S_{VRAM} \cdot B_{RAM\,to\,VRAM})$ becomes zero when the entire dataset is small enough to be replicated in the texture memory of each node, that is, when $D \leq S_{VRAM}$.

4.1.2 Analytical Results

Many of the variables in the cost model reflect system properties, such as the amount of RAM on each node. These variables must have values associated with them in order to use the

cost model. Values are chosen that closely resemble the testbed system described later. The values used in this analysis are:

$$B_{disk\,to\,RAM} = 1.5\,Gbps$$

$$B_{RAM\,to\,RAM} = 1.0\,Gbps$$

$$B_{RAM\,to\,VRAM} = 2.1\,GBps$$

$$S_{RAM} = 2\,GB$$

$$S_{VRAM} = 128\,MB$$

Figure 4.1 shows the result of evaluating the cost model for a number of output resolutions, data sizes and processor numbers. Six different output resolutions are shown, ranging from one megapixel to 128 megapixels. Seven datasets are evaluated, ranging from two gigabytes to 128 gigabytes. Eight different cluster sizes with between eight and 1024 processors are examined.

The graphs in Figure 4.1 show that for any output image size, as the number of nodes increases, the time to render a given dataset decreases. This means that the methodology is scalable in terms of output resolution and the number of processors. Further, even for the largest dataset and output resolution, the total time to render in the worst-case is around five seconds.

The plots for each dataset appears to follow the same pattern of behavior. As the data size increases, the shape of the plot remains the same. The only difference is a shift that indicates

an increase in rendering time. This shows that the method is scalable in terms of data size and the number of processing nodes.

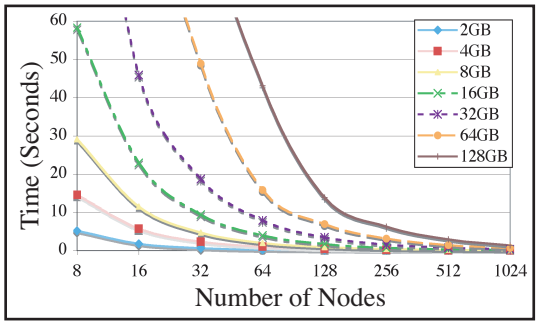
Figure 4.2 and Figure 4.3 show the same results of evaluating the cost model on different number of nodes for a given dataset as the output resolution increases. For a given dataset size, and for a fixed number of nodes, the time to render increases, as expected. As the number of nodes increases, the time to render decreases.

4.2 Experimental Results

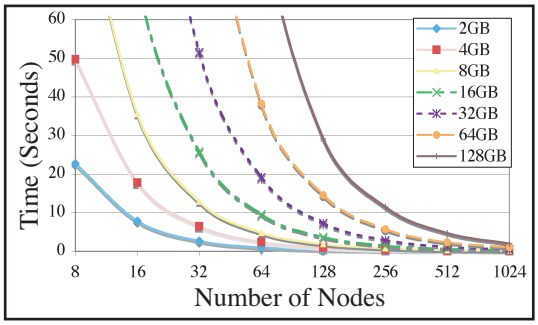
In order to validate the theoretical cost model described in the previous section, a performance evaluation is performed on an implementation of the methodology. A commodity cluster attached to a scalable high-resoltuion tiled-display serves as a testbed. Data of various sized is generated for testing. Results show that a smaller octree brick size produces a performance speed-up over a larger octree brick size. Experimental tests also show that the system scales to produce larger output images form larger data sizes as the number of processing nodes increases.

4.2.1 Test Platform

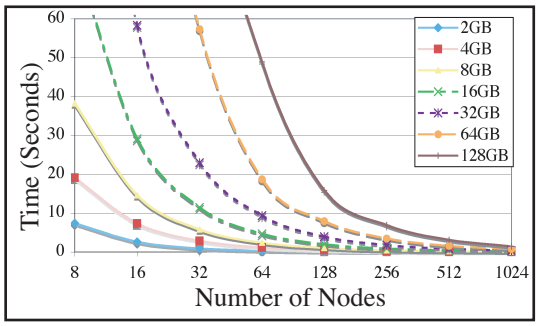
A twenty-nine node cluster serves as the test platform for running experimental tests. Each node has two AMD Opteron 246 processors running at 2GHz, 2 GB of RAM, and 250 GB of local storage. The cluster is interconnected by a 1 Gbps Ethernet backplane. Each node mounts a large network file system (NFS) using the cluster's backplane. In addition, each node is equipped with an nVidia FX3000 graphics processor with 128MB of VRAM over an AGP 8x bus.



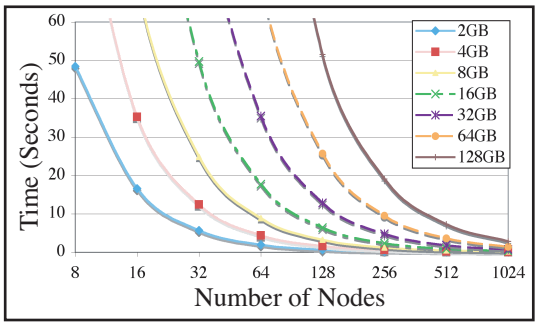
a) 1 Mega-Pixel Output Image



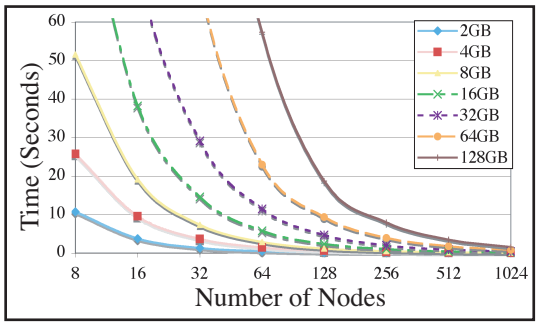
d) 16 Mega-Pixel Output Image



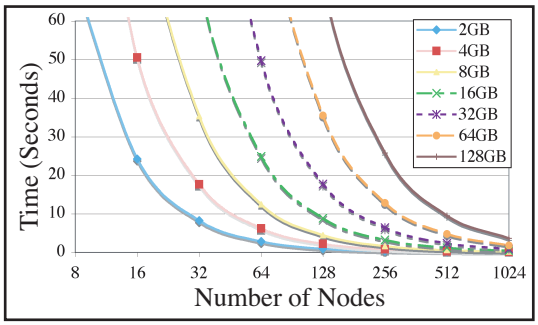
b) 2 Mega-Pixel Output Image



e) 64 Mega-Pixel Output Image

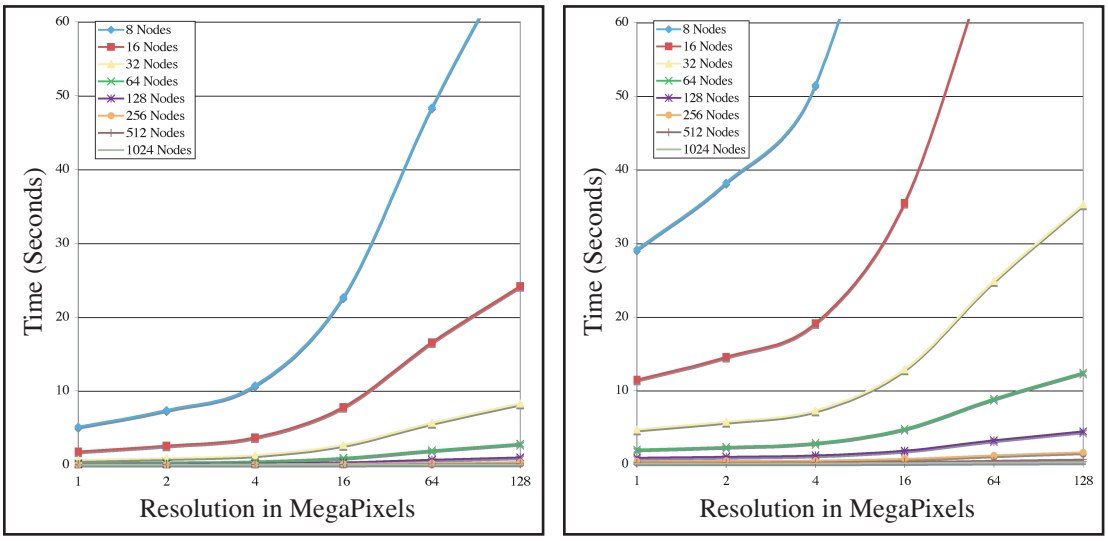


c) 4 Mega-Pixel Output Image



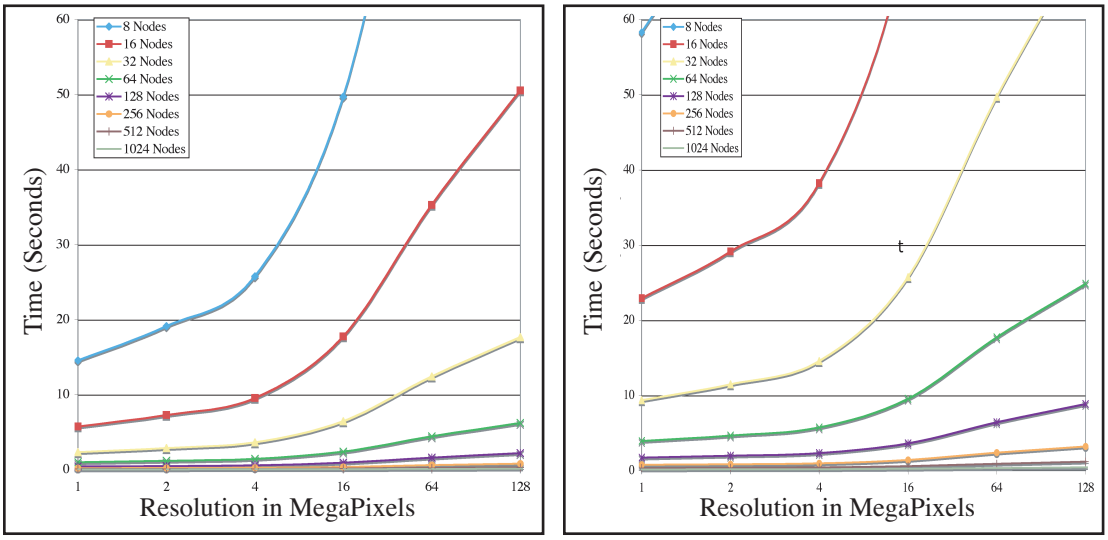
f) 128 Mega-Pixel Output Image

Figure 4.1. Cost model evaluation for different data sizes and output resolutions showing performance as the number of processors increase.



a) 2 GigaBytes of Data

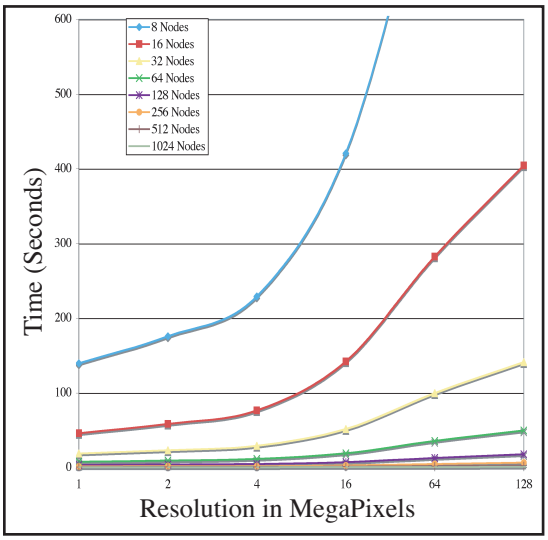
c) 8 GigaBytes of Data



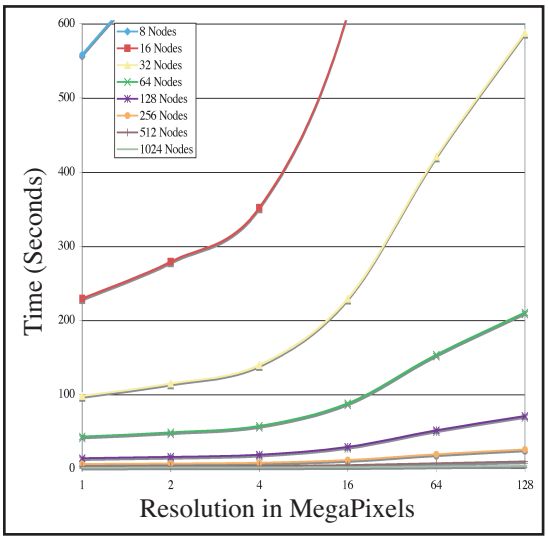
b) 4 GigaBytes of Data

d) 16 GigaBytes of Data

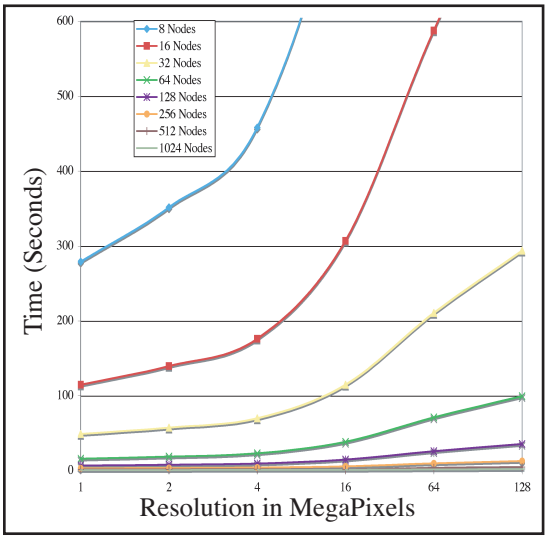
Figure 4.2. Cost model evaluation for increasing output resolution on datasets from 2 giga-bytes to 16 giga-bytes.



a) 32 GigaBytes of Data



c) 128 GigaBytes of Data



b) 64 GigaBytes of Data

Figure 4.3. Cost model evaluation for increasing output resolution on datasets from 32 giga-bytes to 128 giga-bytes.

The cluster is configured such that one node serves as a master node, and the remaining nodes serve as slave nodes. In this configuration, it powers EVL's 105 megapixel LambdaVision display. Each of the slave nodes connects to two of the fifty-five commodity monitors that makes the display, and generates up to a 3.75 megapixel output image.

4.2.2 Test Data

Test data is generated in order to run experimental performance tests on the implementation system. Each test dataset contains 16-bit unsigned integer voxels. The data values are a gradient in the z-direction over the extent of the 16-bit space, that is, they range from zero to 65,535.

The color transfer function used graduates through a rainbow like spectrum. An opacity transfer function is chosen that ranges evenly from completely transparent to completely opaque. Figure 4.4 shows the test dataset rendered at various orientations.

The actual contents of the test data as well as the transfer functions chosen will not alter the performance of this system. The hardware accelerated 3D texture mapping technique performs texture lookups for all voxels within view regardless of the visibility determined by occlusion or transparency due to classification. This is in contrast with other software and GPU based rendering methods that take advantage of classification dependent optimization techniques, such as early ray termination and empty space skipping.

Three datasets of sizes 2 GB, 16 GB and 128 GB are generated. Each dataset is preprocessed into three octrees of varying levels. For a given dataset, increasing the number of levels in the octree increases the number of bricks in the octree while reducing each brick's size. The process

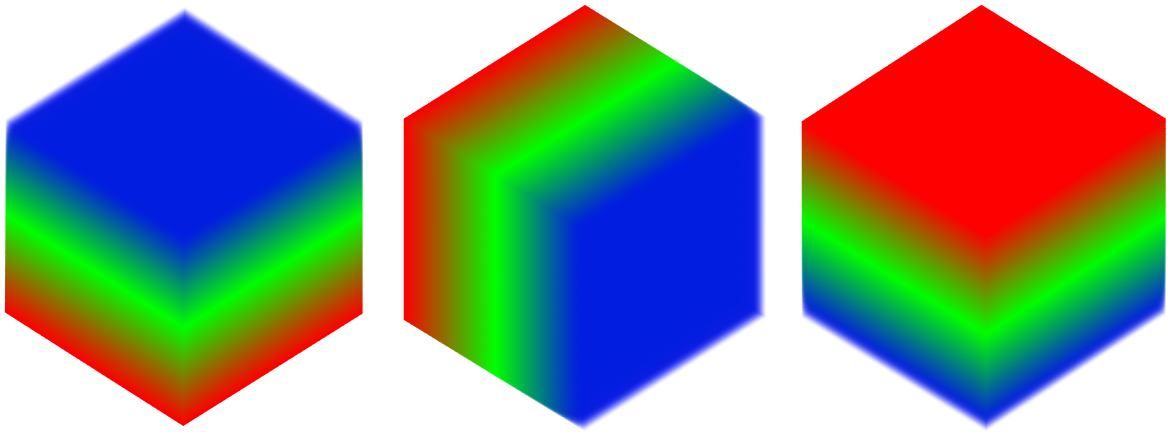


Figure 4.4. Test data rendered at various orientations.

of generating an octree increases the size of the dataset because each octree contains data for coarser resolution versions of the original data in addition to the original data. Table II lists each test dataset and its original size, along with the number of levels in each octree, the number of bricks in each octree, each brick's dimensions and size, and the total size of the dataset after processing.

4.2.3 Test Results

Performance tests measure the best brick size to use, the time to render the highest level-of-detail, and the rate at which users can interact with the system with the lowest level-of-detail. Preliminary tests are performed to determine the brick size that produces the best performance. Thorough tests are performed on the highest level-of-detail for each of the three datasets with the optimum brick configuration, and vary the number of nodes and output resolution. Tests

are run with groups of eight, sixteen, twenty-four and twenty-eight nodes, and produce output images of one, four, sixteen, sixty-four, ninety-six and 105 megapixels. Each dataset is tested on each of the four groups of nodes, However, because a single node is limited to producing a maximum output resolution, tests with higher resolution output images are restricted to using larger numbers of nodes.

Preliminary test are performed on each dataset to determine which brick size produces the best result. Exhaustive tests are then performed based on the findings. The results were reasonably similar regardless of the number of nodes, the size of the dataset, or the output resolution. The largest brick size is the baseline for comparing the other brick sizes against. Figure 4.5 shows the results of testing the three various brick sizes. The smallest brick size, 0.5 MB, increases performance the most. The moderate brick size, 4 MB, increased performance by less than 10 percent. Because of these results, the smallest brick size is used in all further tests.

Figure 4.6 shows test results for rendering the highest level-of-detail of each dataset at different output resolutions using an increasing number of nodes. The datasets are rotated a few degrees about their y-axis every frame, making one complete rotation. The charts for tests up to and including the test producing a sixty-four megapixel output image show that as the number of nodes increases for a given dataset and output resolution, the rendering time decreases. This holds true as the output resolution increases, indicating that the system scales in terms of the number of nodes and output resolution. The graphs for the tests producing

TABLE II

PROPERTIES OF TEST DATASETS

Dimensions	Original Size	Number of Levels	Number of Bricks	Brick Dimensions	Brick Size	Total Size
1024^3	2GB	3	73	256^3	32 MB	2.3 GB
		4	585	128^3	4 MB	2.3 GB
		5	4681	64^3	0.5MB	2.3 GB
2048^3	16GB	4	585	256^3	32 MB	18.4 GB
		5	4681	128^3	4 MB	18.4 GB
		6	37449	64^3	0.5 MB	18.4 GB
4096^3	128GB	5	4681	256^3	32 MB	146 GB
		6	37449	128^3	4 MB	146 GB
		7	299593	64^3	0.5 MB	146 GB

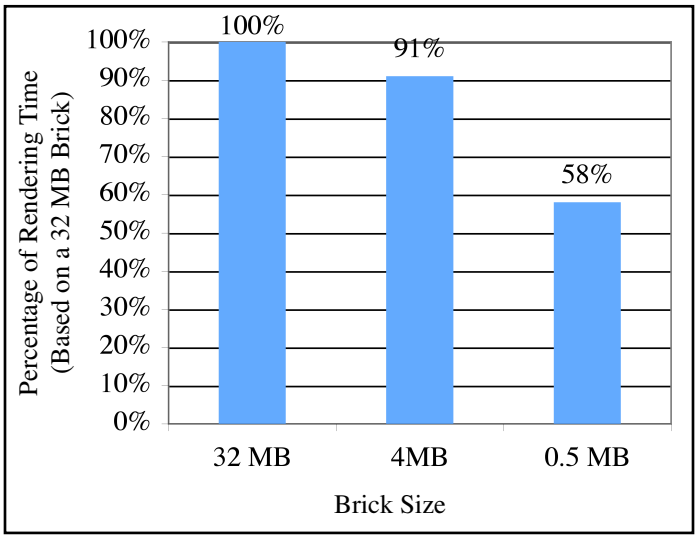


Figure 4.5. Impact of brick size on rendering performance.

ninety-six and 105 megapixel output images do not have enough data to draw a conclusion, but act as a reference to the system's performance.

The charts in Figure 4.7 show the same results of evaluating the cost model on different number of nodes for a given dataset as the output resolution increases. For a given dataset size, and for a fixed number of nodes, the time to render increases, as expected. As the number of nodes increases, the time to render decreases. Not all combinations of nodes and output resolutions are represented for the reason mentioned above.

When a user interacts with the system the lowest level-of-detail is loaded allowing for much faster interaction. The data for this level is stored entirely in one brick, which becomes replicated on each node's graphics card when in use. Thus, maintaining interactivity in the system is dependent on having a small enough brick size such that the system can render a single brick at an interactive rate.

Each of the three bricks sizes tested allows the user to interact with the system at at least one or two frames per second. The performance of the interactive rate increases for smaller brick sizes. The maximum achieved during these tests was twelve frames per second using a 64^3 brick size.

4.3 Comparison

A comparison of the experimental test results and the analytic results from the cost model validate the models correctness. Plots of the analytic results and experimental results together show that the implementation system performs better than the worst-case model during the test cases. Although the experimental results out perform the cost model, the experimental

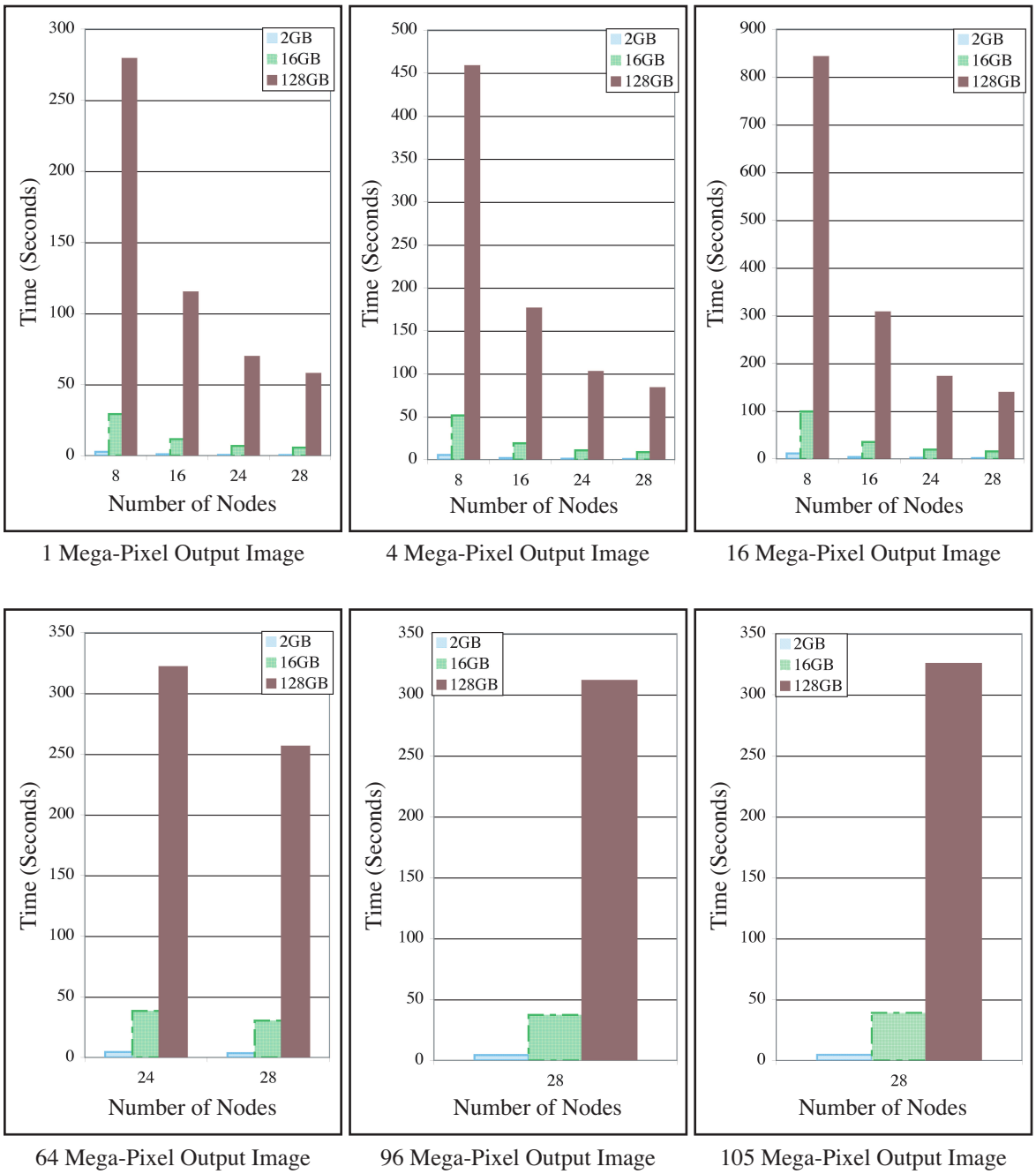
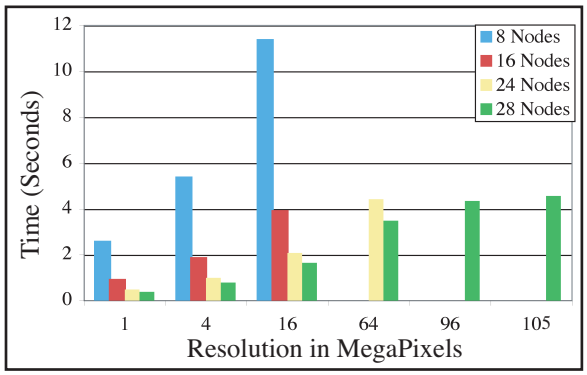
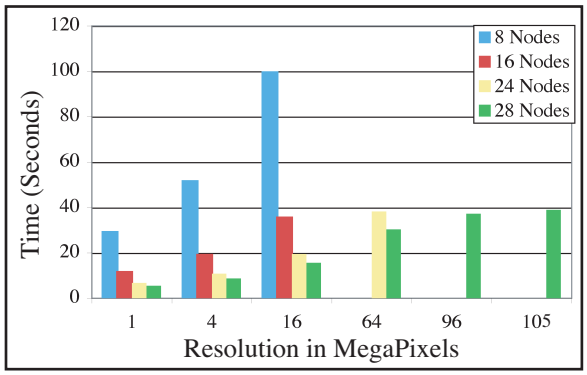


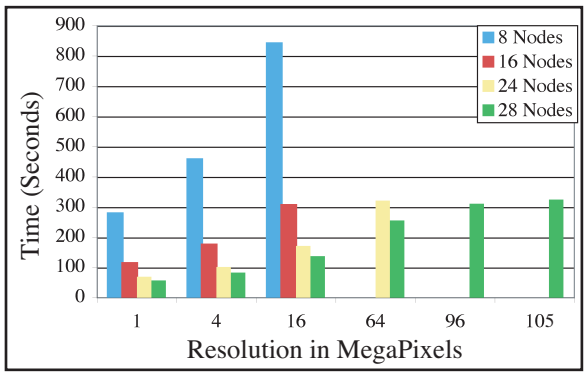
Figure 4.6. Performance test results for different data sizes and output resolutions showing rendering time as the number of processors increase.



2 Giga-bytes of Data



16 Giga-bytes of Data

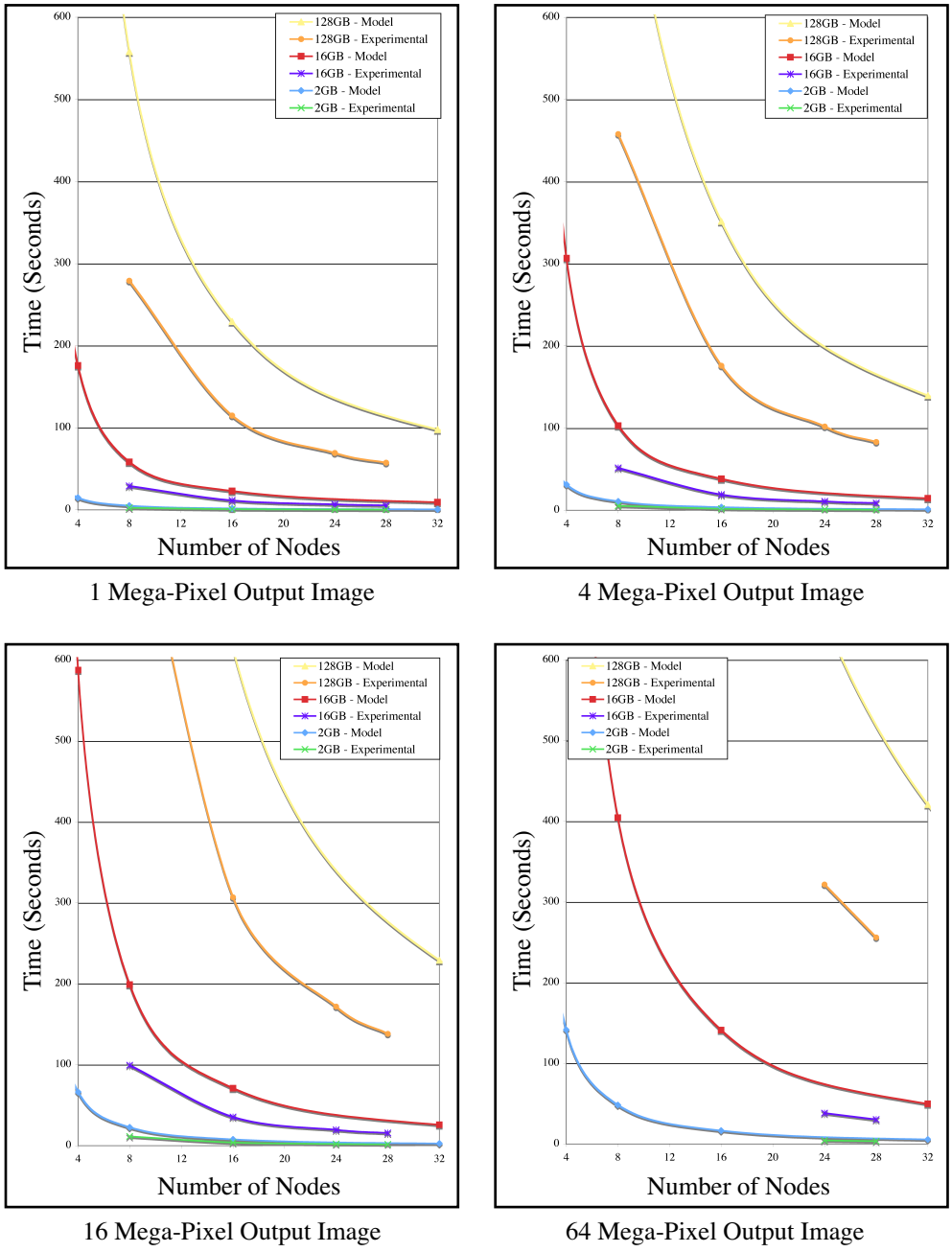


128 Giga-bytes of Data

Figure 4.7. Performance test results as output resolution increases.

results and the analytic results follow the same pattern of behavior and scalability. This shows that the cost model provides a valid upper-bound analysis of the methodology.

Results of the cost model evaluation and of the experimental tests are shown together in Figure 4.8 for four different output resolutions. The experimental results take about half the time as the analytic model predicted. This discrepancy can be attributed to the brick size and to image-coherency. Results from the brick size tests shown in Figure 4.5 indicate that the rendering time using the smallest brick size is 58 percent of the time when rendering using the largest brick size. This accounts for much of the discrepancy indicated in the graphs. The rest of the discrepancy may be attributed to image-coherancy, that is, that because the dataset is rotated by small amounts, only a small amount of data needs to be redistributed among the nodes, lowering the overall data distribution time. Regardless, the discrepancy is a constant shift in rendering time, indicating that the experimental system scales by the relationship as does the analytic model.



1 Mega-Pixel Output Image

4 Mega-Pixel Output Image

16 Mega-Pixel Output Image

64 Mega-Pixel Output Image

Figure 4.8. Performance results from the analytic cost model and experimental tests.

CHAPTER 5

APPLICATIONS

In order to demonstrate the applicability of the methodology presented, the system is applied to a number of domain specific datasets. The datasets range in size from a few hundred megabytes to fifty gigabytes, and are visualized on two high-resoltuion displays; the LambdaVision, a 105 megapixel displays, and the LambdaTable, a twenty-four megapixel interactive tabletop display. The Purkinje neuron and the rat kidney datasets are examples from the bioscience domain. The rock dataset comes from the geoscience community and is used to help understand the reactions that occur during rock formation. The Visible Female data comes from the medical domain and is often used by the volume rendering community.

5.1 Purkinje Neuron

Purkinje cells are some of the largest neurons in the human brain. They have an intricately elaborate network of dendritic spines. Purkinje cells are arranged in a domino like fashion in the portion of the brain responsible for fine motor control, the cerebellar cortex. Each cell is a tiny supercomputing structure with up to 200,000 inputs from various sensory units all over the body. They are triggered at rates of ten to 150 times a second and have only one output, which is the result of their internal computation.

Depending on the species, a cerebeller cortex can contain billions of these cells. In humans, Purkinje cells are affected in a variety of diseases ranging from toxic exposure from alcohol and

lithium, to genetic mutations, such as spinocerebellar ataxias and autism, and neurodegenerative diseases that are not thought to have a known genetic basis, such as sporadic ataxias.

The Purkinje neuron shown in Figure 5.1, Figure 5.2 and Figure 5.3 is sampled from a rat brain and imaged by multi-photon microscope. Each layer in the volume is a mosaic of smaller, high-resolution images that are stitched together. The layers are stacked to create a volume. The original slice data creates a 1664 x 2080 x 109 volume of 8-bit samples. The original size of the data is 359 MB. The data is provided by Hiroyuki Hakozaki, Diana Price, Masako Terada and Mark Ellisman of the National Center for Microscopy and Imaging Research (NCMIR) at the University of California, San Diego (UCSD).

5.2 Rat Kidney

The rat kidney shown in Figure 5.4 and Figure 5.5 is imaged by a high-power microscope. Each layer in the volume is a mosaic of smaller, high-resolution images that are stitched together. The layers are stacked to create a volume. The original slice data creates a 34664 x 22043 x 23 volume of 24-bit color samples. The original size of the data is 50 GB. The data is provided by Hiroyuki Hakozaki, Diana Price, Masako Terada and Mark Ellisman of the National Center for Microscopy and Imaging Research (NCMIR) at the University of California, San Diego (UCSD).

5.3 Rock Sample

Geoscientists are interested in the reactions that occur between different areas within rocks, and in the pathways that connect those different areas. Previously, the internal structure of rocks was determined by manually polishing rocks at set intervals, a very labor intensive task.

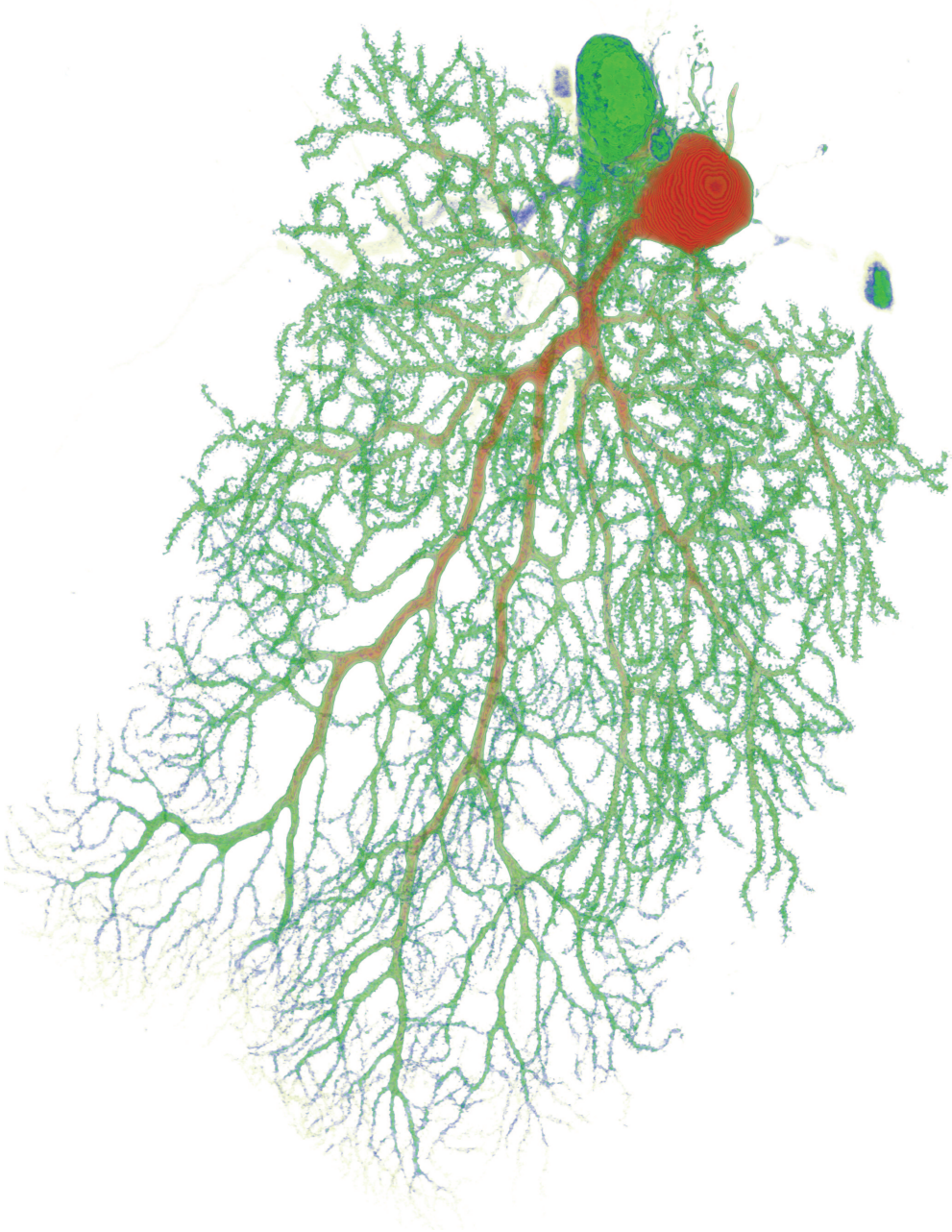


Figure 5.1. Volume visualization of the Purkinje neuron.

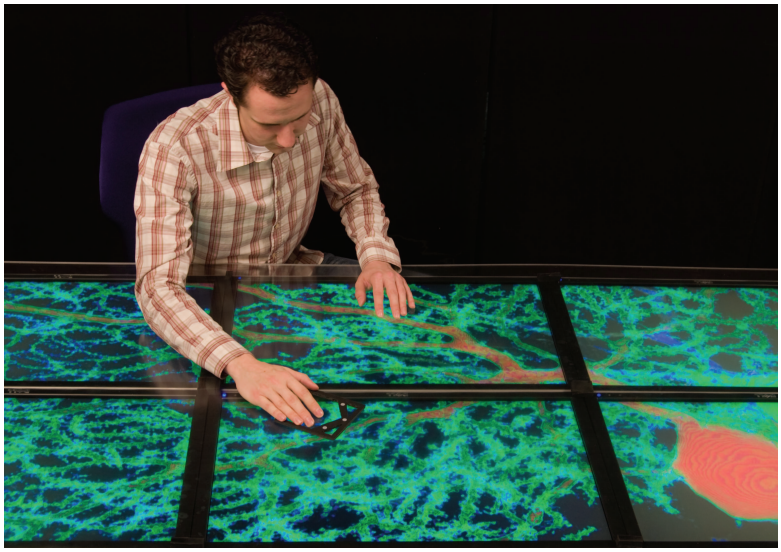


Figure 5.2. Researcher analyzing the Purkinje neuron on the LambdaTable.

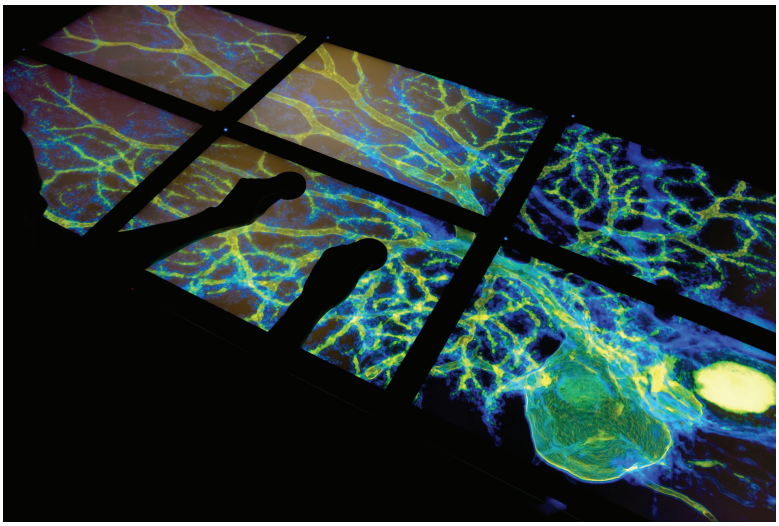


Figure 5.3. Scientist interacting with the Purkinje neuron on the LambdaTable.

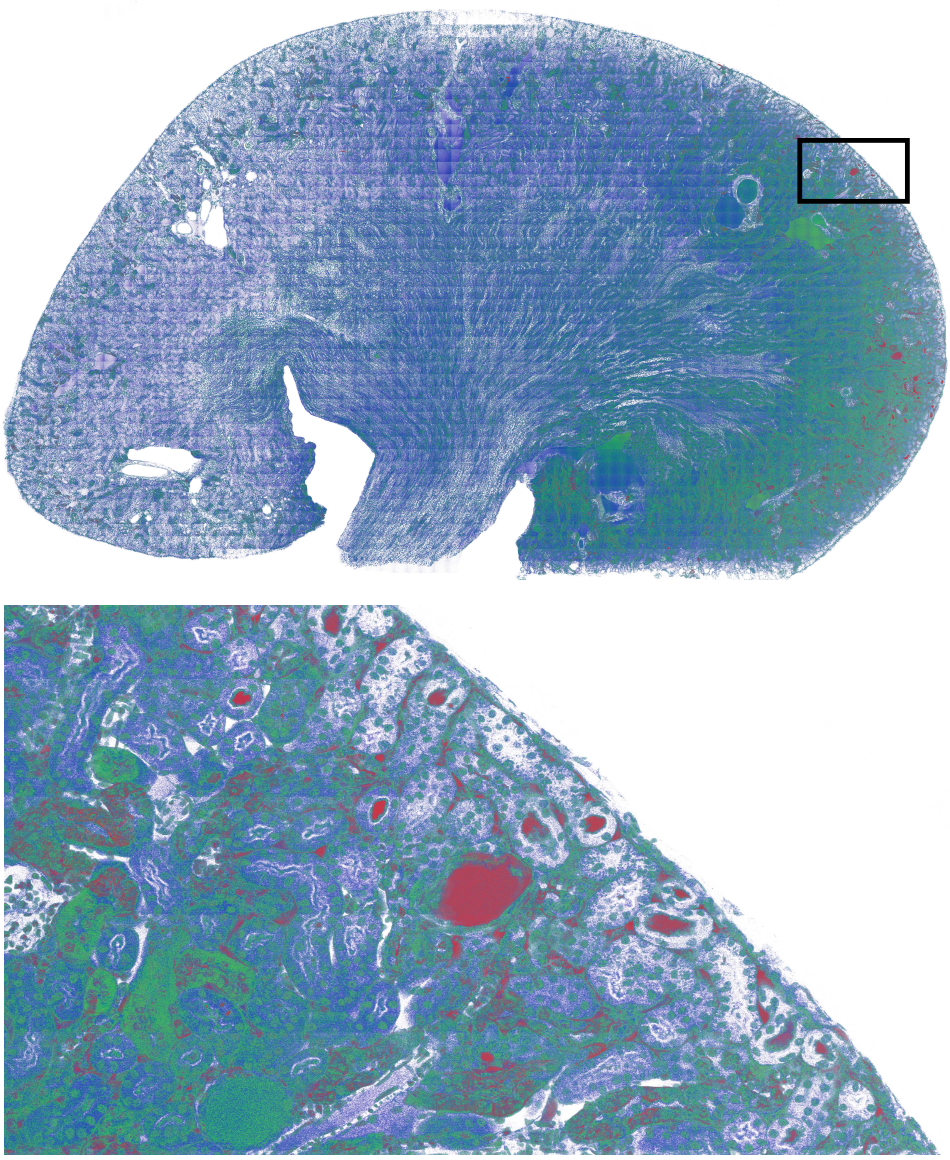


Figure 5.4. Volume visualization of the rat kidney. The image at the top shows the entire kidney. The bottom image shows the section highlighted in the top image.

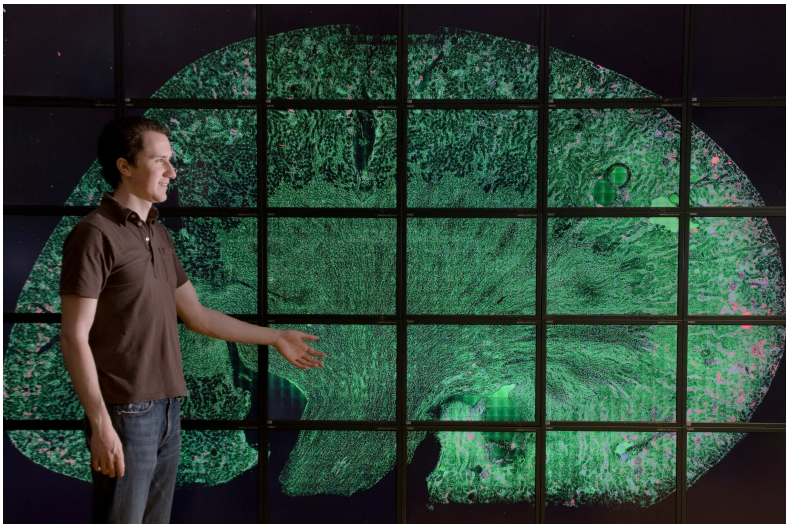


Figure 5.5. Volume visualization of the rat kidney on the LambdaVision.

Now, using computed tomography (CT) scanners, geoscientists can get faster and more precise samples. In this sample, geoscientists are interested in minerals in a pathway between garnet and sillimanite. The minerals in this location can provide clues as to how the rock was formed.

The rock sample shown in Figure 5.6 is imaged by a Computed Tomography (CT) scanner. The data consists of CT scans of a rock sample that measures about 25 mm x 25 mm x 27 mm. Each cross-section is taken at a resolution of 1024 pixels by 1024 pixels where each pixel is made up of 16-bits of grey tone. There are a total of 2,450 cross-sections making the entire dataset 4.8 GB. The data was provided by Eric Goergen of the University of Minnesota (UMN).

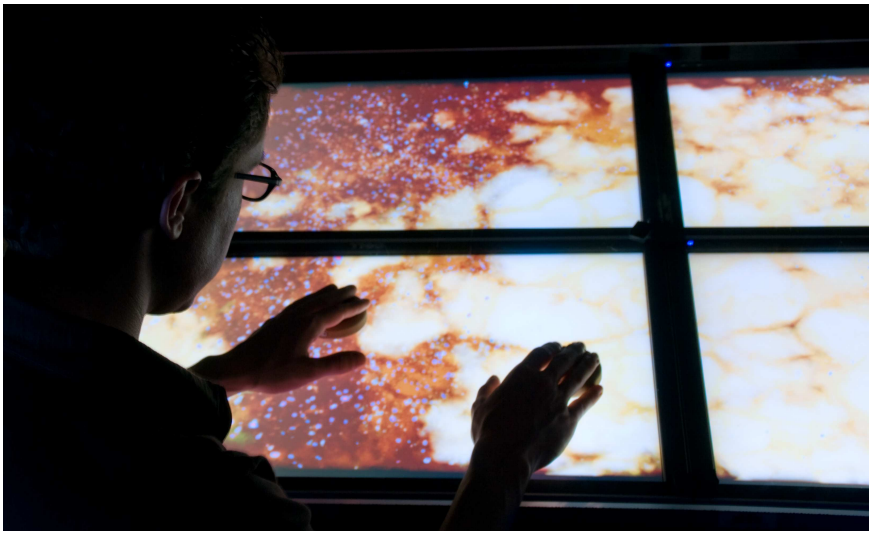


Figure 5.6. Volume visualization of the rock sample on the LambdaTable highlighting high-intensity garnet regions.

5.4 Visible Female

The Visible Female data was collected as part of the the National Library of Medicine's (NLM) Visible Human Project. It serves as a reference for the study of human anatomy by providing a complete, anatomically detailed, three-dimensional representation of the normal female human body. The Visible Female data consists of axial CT scans of an entire female body taken at 0.33 mm intervals at a resolution of 512 pixels by 512 pixels where each pixel is made up of 12-bits of grey tone. There are a total of 1,871 cross-sections making the entire dataset 702 MB. Figure 5.7 shows a volume visualization of the Visible Female on the LambdaTable. Figure 5.8 shows a volume visualization of the Visible Female skull.

Figure 5.7. Volume visualization of the Visible Female on the LambdaTable.

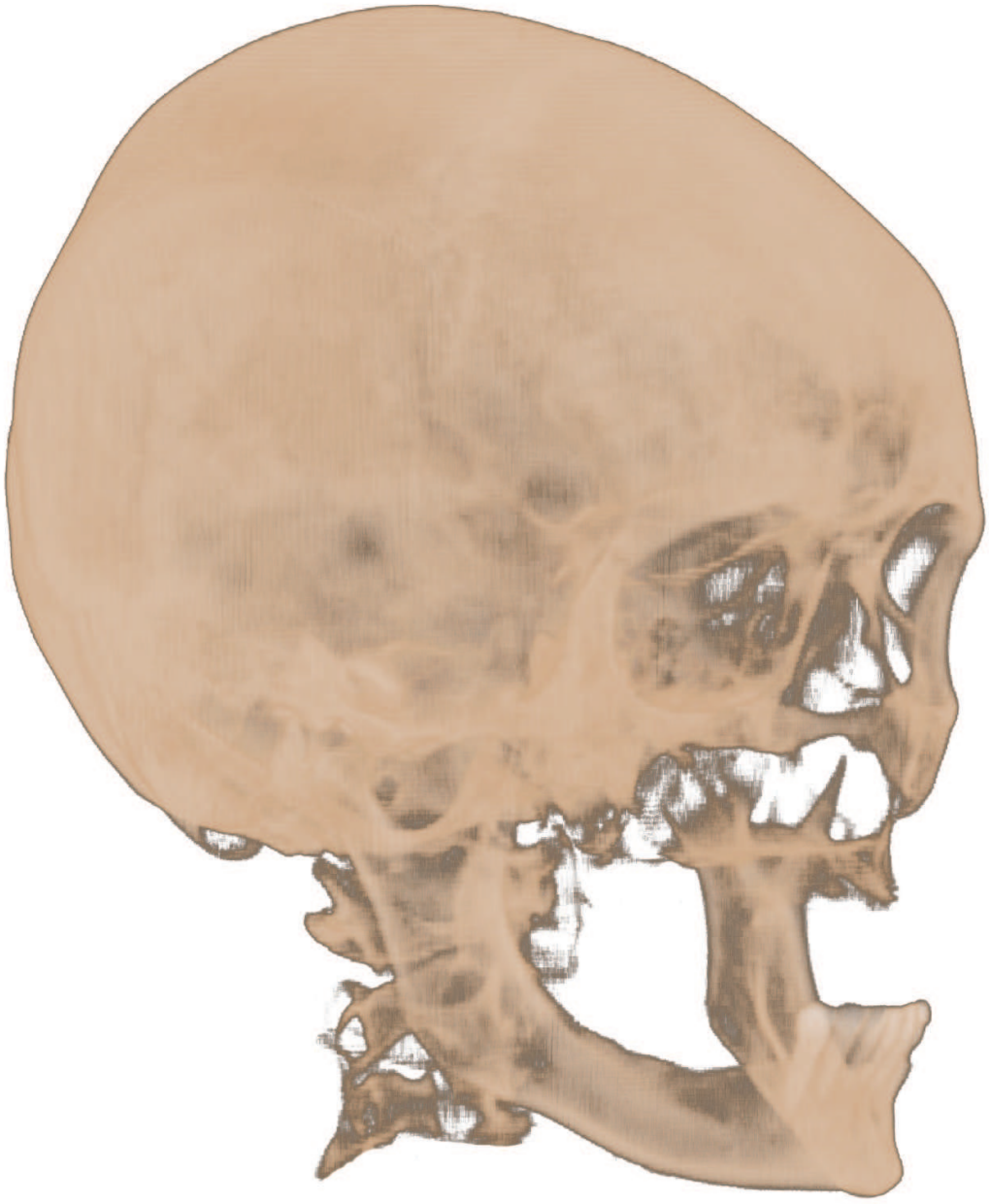


Figure 5.8. Volume visualization of the Visible Female skull.

CHAPTER 6

CONCLUSION

This thesis presents a methodology for rendering very large volume data on scalable high-resolution displays using a distributed-memory commodity cluster. The methodology uses a number of techniques which when combined produce a system that is scalable in terms of input data size and output resolution. An analytical cost model validated by experimental results predicts the system's behavior. Finally, the system is demonstrated using a number of domain specific datasets.

A thorough review of research in the volume rendering field and especially in the sub-field of parallel volume rendering was performed. The review found that although much research has been performed in the area of visualizing large volume data on small displays, and some research has been performed concerning rendering small data on large displays, not enough exists that addresses rendering large data on scalable high-resolution displays.

A methodology that addresses rendering large volume data on scalable high-resolution displays is presented that employs a number of techniques. Data is preprocessed into a multi-resolution octree that provided interactivity and an efficient structure for data management. An image-order data distribution approach is used that assigns each rendering node driving a high-resolution display a disjoint portion of the output image to render. A data management system that includes a distributed shared-memory system and a multi-level cache is presented. One level of cache exists between the local texture memory of each node and the distributed shared-

memory system, and the other level of cache resides between each node's local disk and the memory system. Local rendering is performed using hardware accelerated 3D texture-mapping techniques and view-aligned proxy geometry. Interactive post-classification is performed with the use of a fragment program.

A worst-cast analytical cost model describing the methodology's run-time behavior is presented. Experimental results validate the model for a number of test scenarios. The model and results show that the system is scalable in terms of increasing input data size, increasing output resolution, and an increasing number of processors. An optimal brick size for nodes in the octree is also determined.

The applicability of the methodology presented is demonstrated with a number of domain specific datasets. The datasets range in size from a few hundred megabytes to fifty gigabytes, and are visualized on two high-resoltuion displays. The Purkinje neuron and the rat kidney datasets are examples from the bioscience domain. The rock dataset comes from the geoscience community and is used to help understand the reactions that occur during rock formation. The Visible Female data comes from the medical domain and is often used by the volume rendering community.

CITED LITERATURE

1. Renambot, L., Jeong, B., Jagodic, R., Johnson, A., Leigh, J., and Aguilera, J.: Collaborative visualization using high-resolution tiled displays. In CHI 06 Workshop on Information Visualization and Interaction Techniques for Collaboration Across Multiple Displays, 2006.

2. Krumbholz, C., Leigh, J., Johnson, A., Renambot, L., and Kooima, R.: Lambda table: High resolution tiled display table for interacting with large visualizations. In Workshop for Advanced Collaborative Environments (WACE) 2005, 2005.

3. Meißner, M., Huang, J., Bartz, D., Mueller, K., and Crawfis, R.: A practical evaluation of popular volume rendering algorithms. In Proceedings of the 2000 IEEE Symposium on Volume Visualization, pages 81–90, New York, NY, USA, 2000. ACM Press.

4. Levoy, M.: Display of surfaces from volume data. IEEE Computer Graphics and Applications, 8(3):29–37, 1988.

5. Westover, L.: Footprint evaluation for volume rendering. In SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques, pages 367–376, New York, NY, USA, 1990. ACM Press.

6. Mueller, K. and Crawfis, R.: Eliminating popping artifacts in sheet buffer-based splatting. In VIS '98: Proceedings of the conference on Visualization '98, pages 239–245, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

7. Mueller, K., Shareef, N., Huang, J., and Crawfis, R.: High-quality splatting on rectilinear grids with efficient culling of occluded voxels. IEEE Transactions on Visualization and Computer Graphics, 5(2):116–134, 1999.

8. Lacroute, P. and Levoy, M.: Fast volume rendering using a shear-warp factorization of the viewing transformation. In SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques, pages 451–458, New York, NY, USA, 1994. ACM Press.

9. Rezk-Salama, C., Engel, K., Bauer, M., Greiner, G., and Ertl, T.: Interactive volume rendering on standard pc graphics hardware using multi-textures and multi-stage rasterization. In HWWS '00: Proceedings of the ACM

SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware, pages 109–118, New York, NY, USA, 2000. ACM Press.

10. Cabral, B., Cam, N., and Foran, J.: Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In VVS '94: Proceedings of the 1994 symposium on Volume visualization, pages 91–98, New York, NY, USA, 1994. ACM Press.

11. Grzeszczuk, R., Henn, C., and Yagel, R.: Advanced geometric techniques for ray casting volumes: Siggraph 1998 course notes. In SIGGRAPH '98: ACM SIGGRAPH 1998 courses, New York, NY, USA, 1998. ACM Press.

12. Fernando, R. and Kilgard, M. J.: The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics. Boston, MA, USA, Addison-Wesley Longman Publishing Co., Inc., 2003.

13. Rost, R. J.: OpenGL(R) Shading Language (2nd Edition). Addison-Wesley Professional, 2005.

14. Ikits, M., Kniss, J., Lefohn, A., and Hansen, C.: Volume Rendering Techniques, chapter 39, pages 667–691. Addison Welsey, 2004.

15. Kruger, J. and Westermann, R.: Acceleration techniques for gpu-based volume rendering. In VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03), page 38, Washington, DC, USA, 2003. IEEE Computer Society.

16. Pfister, H., Hardenbergh, J., Knittel, J., Lauer, H., and Seiler, L.: The volumepro real-time ray-casting system. In SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques, pages 251–260, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

17. Meißner, M., Kanus, U., Wetekam, G., Hirche, J., Ehlert, A., Straßer, W., Doggett, M., Forthmann, P., and Proksa, R.: Vizard ii: a reconfigurable interactive volume rendering system. In HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, pages 137–146, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

18. Woo, M., Neider, J., Davis, T., and Shreiner, D.: OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2. Boston, MA, USA, Addison-Wesley Longman Publishing Co., Inc., 1999.

19. Bhaniramka, P. and Demange, Y.: Opengl volumizer: a toolkit for high quality volume rendering of large data sets. In VVS '02: Proceedings of the 2002 IEEE symposium on Volume visualization and graphics, pages 45–54, Piscataway, NJ, USA, 2002. IEEE Press.

20. LaMar, E., Hamann, B., and Joy, K. I.: Multiresolution techniques for interactive texture-based volume visualization. In VIS '99: Proceedings of the conference on Visualization '99, pages 355–361, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

21. Weiler, M., Westermann, R., Hansen, C., Zimmermann, K., and Ertl, T.: Level-of-detail volume rendering via 3d textures. In VVS '00: Proceedings of the 2000 IEEE symposium on Volume visualization, pages 7–13, New York, NY, USA, 2000. ACM Press.

22. Plate, J., Tirtasana, M., Carmona, R., and Fröhlich, B.: Octreemizer: a hierarchical approach for interactive roaming through very large volumes. In VISSYM '02: Proceedings of the symposium on Data Visualisation 2002, pages 53–ff, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

23. Ning, P. and Hesselink, L.: Vector quantization for volume rendering. In VVS '92: Proceedings of the 1992 workshop on Volume visualization, pages 69–74, New York, NY, USA, 1992. ACM Press.

24. Schneider, J. and Westermann, R.: Compression domain volume rendering. In VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03), page 39, Washington, DC, USA, 2003. IEEE Computer Society.

25. Guthe, S., Wand, M., Gonser, J., and Straßer, W.: Interactive rendering of large volume data sets. In VIS '02: Proceedings of the conference on Visualization '02, Washington, DC, USA, 2002. IEEE Computer Society.

26. Porter, T. and Duff, T.: Compositing digital images. In SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques, pages 253–259, New York, NY, USA, 1984. ACM Press.

27. Molnar, S., Cox, M., Ellsworth, D., and Fuchs, H.: A sorting classification of parallel rendering. IEEE Comput. Graph. Appl., 14(4):23–32, 1994.

28. Palmer, M. E., Totty, B., and Taylor, S.: Ray casting on shared-memory architectures: Memory-hierarchy considerations in volume rendering. IEEE Concurrency, 6(1):20–35, 1998.

29. Schroeder, W., Martin, K. M., and Lorensen, W. E.: The visualization toolkit (2nd ed.): an object-oriented approach to 3D graphics. Upper Saddle River, NJ, USA, Prentice-Hall, Inc., 1998.

30. Amin, M. B., Grama, A., and Singh, V.: Fast volume rendering using an efficient, scalable parallel formulation of the shear-warp algorithm. In PRS '95: Proceedings of the IEEE symposium on Parallel rendering, pages 7–14, New York, NY, USA, 1995. ACM Press.

31. Bajaj, C., Ihm, I., Park, S., and Song, D.: Compression-based ray casting of very large volume data in distributed environments. In Proceedings of HPCAsia, pages 720–725, 2000.

32. Coelho, A., Nascimento, M., Bentes, C., de Castro, M., and Farias, R.: Parallel volume rendering for ocean visualization in a cluster of pcs. In 6th Brazilian Sysmposium on Geoinformatics, GEOINFO 2004, 2004.

33. Hsu, W. M.: Segmented ray casting for data parallel volume rendering. In PRS '93: Proceedings of the 1993 symposium on Parallel rendering, pages 7–14, New York, NY, USA, 1993. ACM Press.

34. Camahort, E. and Chakravarty, I.: Integrating volume data analysis and rendering on distributed memory architectures. In PRS '93: Proceedings of the 1993 symposium on Parallel rendering, pages 89–96, New York, NY, USA, 1993. ACM Press.

35. Ma, K. L., Painter, J. S., Hansen, C. D., and Krogh, M. F.: A data distributed, parallel algorithm for ray-traced volume rendering. In PRS '93: Proceedings of the 1993 symposium on Parallel rendering, pages 15–22, New York, NY, USA, 1993. ACM Press.

36. Ma, K.-L., Painter, J. S., Hansen, C. D., and Krogh, M. F.: Parallel volume rendering using binary-swap compositing. IEEE Comput. Graph. Appl., 14(4):59–68, 1994.

37. Elvins, T. T.: Volume rendering on a distributed memory parallel computer. In VIS '92: Proceedings of the 3rd conference on Visualization '92, pages 93–98, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

38. Mller, C., Strengert, M., and Ertl, T.: Optimized Volume Raycasting for Graphics-Hardware-based Cluster Systems. In Eurographics Symposium on Parallel Graphics and Visualization (EGPGV06), pages 59–66. Eurographics Association, 2006.

39. Gribble, C., Parker, S., and Hansen, C.: Interactive volume rendering of large datasets using the silicon graphics onyx4 visualization system. Technical report, University of Utah, 2004.

40. Heirich, A. and Moll, L.: Scalable distributed visualization using off-the-shelf components. In PVGS '99: Proceedings of the 1999 IEEE symposium on Parallel visualization and graphics, pages 55–59, New York, NY, USA, 1999. ACM Press.

41. Lombeyda, S., Moll, L., Shand, M., Breen, D., and Heirich, A.: Scalable interactive volume rendering using off-the-shelf components. In PVG '01: Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphics, pages 115–121, Piscataway, NJ, USA, 2001. IEEE Press.

42. Frank, S. and Kaufman, A.: Distributed volume rendering on a visualization cluster. In CAD-CG '05: Proceedings of the Ninth International Conference on Computer Aided Design and Computer Graphics (CAD-CG'05), pages 371–376, Washington, DC, USA, 2005. IEEE Computer Society.

43. Muraki, S., Lum, E. B., Ma, K.-L., Ogata, M., and Liu, X.: A pc cluster system for simultaneous interactive volumetric modeling and visualization. In PVG '03: Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics, page 13, Washington, DC, USA, 2003. IEEE Computer Society.

44. Garcia, A. and Shen, H.-W.: An interleaved parallel volume renderer with pc-clusters. In EGPGV '02: Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization, pages 51–59, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

45. McCorquodale, J. and Lombeyda, S. V.: The volumepro volume rendering cluster: A vital component of parallel end-to-end solution. Technical report, California Instutue of Technology, 2003.

46. Schwarz, N., Venkataraman, S., Renambot, L., Krishnaprasad, N., Vishwanath, V., Leigh, J., Johnson, A., Kent, G., and Nayak, A.: Vol-a-tile " a tool for interactive explo-

ration of large volumetric data on scalable tiled displays. In VIS '04: Proceedings of the conference on Visualization '04, page 598.19, Washington, DC, USA, 2004. IEEE Computer Society.

47. Allard, J. and Raffin, B.: A shader-based parallel rendering framework. In VIS '05: Proceedings of the conference on Visualization '05, pages 127–134, 2005.

VITA

NAME	Nicholas Schwarz
EDUCATION	M.S., Computer Science, University of Illinois at Chicago, 2007
	B.S., Computer Science, University of Illinois at Chicago, 2003
EXPERIENCE	Graduate Research Assistant, Electronic Visualization Laboratory, University of Illinois at Chicago, 2003 - 2007
	Intern Programmer, National Center for Microscopy and Imaging Research, University of California, San Diego, 2005
	Undergraduate Teaching Assistant, Department of Computer Science, University of Illinois at Chicago, 2002
PUBLICATIONS	Singh, R., Schwarz, N., Taesombut, N., Lee, D., Jeong, B., Renambot, L., Lin, A., West, R., Otsuka, H., Peltier, S., Martone, M., Nozaki, K., Leigh, J., Ellisman, M., Real-time Multi-scale Brain Data Acquisition, Assembly, and Analysis using an End-to-End OptIPuter, International Journal of Future Generation Computer Systems, Elsevier, 22.8 (2006), p. 1032-1039.
POSTERS	Schwarz, N., van Keken, P., Renambot, L., Tromp, J., Komatitsch, D., Johnson, A., Leigh, J., Distributed Volume Rendering of Global Models of Seismic Wave Propagation, Eos Trans. AGU, 85(47), Fall Meet. Suppl., Abstract SF13A-0702, San Francisco, California, 2004.
	Schwarz, N., Venkataraman, S., Renambot, L., Krishnaprasad, N., Vishwanath, V., Leigh, J., Johnson, A., Kent, G., Nayak, A., Vol-a-Tile - a Tool for Interactive Exploration of Large Volumetric Data on Scalable Tiled Displays, IEEE Visualization 2004, Austin, Texas, 2004.