AF INSTITUTE OF TECHNOLOGY

Distributed Ray Casting for High-Speed Volume Rendering

THESIS

Patricia L. Brightbill

Captain

AFIT/GCS/ENG/92M-01

DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/92M-01

Distributed Ray Casting for High-Speed Volume Rendering

THESIS

Patricia L. Brightbill

Captain

AFIT/GCS/ENG/92M-01

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

[illegible]

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE January 1992	3. REPORT TYPE AND DATES COVERED Master's Thesis
4. TITLE AND SUBTITLE Distributed Ray Casting for High-Speed Volume Rendering		5. FUNDING NUMBERS
6. AUTHOR(S) Patricia L. Brightbill, Capt, USAF		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology, WPAFB OH 45433-6583		8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCS/ENG/92M-01
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES		
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited		12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The volume rendering technique known as ray casting or ray tracing is notoriously slow for large volume sizes, yet provides superior images. A technique is needed to accelerate ray tracing volumes without depending on special purpose or parallel computers. The realization and improvements in distributed computing over the past two decades has motivated its use in this work. This thesis explores a technique to speedup ray casting by distributed programming. The work investigates the possibility of dividing the volume among general purpose workstations and casting rays (using Levoy's front-to-back algorithm) through each subvolume independently. The final step being the composition of all subvolume rendered images. Results indicate a 75 percent savings in rendering time by distributed processing over eight processors versus a single processor.

14. SUBJECT TERMS Computer Graphics, Medical Imaging, Volume Rendering, Ray Casting, Distributed Computing, Distributed Ray Casting, Parallel Ray Casting			15. NUMBER OF PAGES 61
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

AFIT/GCS/ENG/92M-01

Distributed Ray Casting for High-Speed Volume Rendering

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the Requirements
for the Degree of Master of Science in Computer Systems

Patricia L. Brightbill, B.S. in Computer Science
Captain, USAF

January, 1992

Approved for public release; distribution unlimited

Table of Contents

Figures

Tables

Acknowledgements

I would like to acknowledge several individuals to whom I am indebted for their assistance in accomplishing this work. Their order of acknowledgment has no bearing on contribution or significance. I thank them all.

Special thanks goes to my advisor, Lt Col Martin Stytz, for his unending support, flexibility, and availability to listen to my rambling thoughts. His wisdom in this area helped me throughout this work. I also want to express my sincere gratitude for his understanding demeanor and altruism which helped me immensely during my personal challenges these past few months.

Lt Col Phil Amburn also deserves notable mention as my committee member and instructor. His love of computer graphics has worn off on me and influenced my decision to pursue research in this area, as well as seek future employment in this area. His technical guidance in my work has kept me from spinning my wheels endlessly.

I must thank Tracy Suchar for her dedication and expertise in making this document look so professional. Her enthusiasm and willingness to help me out whenever I was in dire straits has lowered my stress level at those critical times. I thank her and her husband, Bill Sommers, for their friendship and lent ears over these many months. I especially appreciate their non-vacillating stance on hospitality.

Thanks go to all my friends that I have made at AFIT. Some of those friends, namely Pat Rizzuto and Rob Parrott, shared the same research area as me and I am thankful for their suggestions, criticisms, and white-board scribble sessions.

I thank God for helping me get through the good, the bad, and the rough times.

Finally I would like to thank all of my family, without their love and support I don't know where I would be. And my dog, Casey, who provided me with companionship, love, and laughter into the wee hours of the morning.

Abstract

The volume rendering technique known as ray casting or ray tracing is notoriously slow for large volume sizes, yet provides superior images. A technique is needed to accelerate ray tracing volumes without depending on special purpose or parallel computers. The realization and improvements in distributed computing over the past two decades has motivated its use in this work.

This thesis explores a technique to speedup ray casting by distributed programming. The work investigates the possibility of dividing the volume among general purpose workstations and casting rays (using Levoy's front-to-back algorithm) (Levoy, 1990) through each subvolume independently. The final step being the composition of all subvolume rendered images.

Results indicate a 75 percent savings in rendering time by distributed processing over eight processors versus a single processor.

I. Introduction

1.1 *Background*

Volume rendering is a method of projecting a three-dimensional (3D), volumetric scene onto a 2D image plane. The distinction of volume rendering is its processing on the acquired volume data directly; no object boundary or surface extraction is used in generating the image (Udupa, 1991:3) (Talton, 1987:121). One advantage to volume rendering is that no binary classification is necessary, thus poorly defined features can be seen (Fuchs, 1989:47). Another advantage of volume rendered medical images is the ability to view the body at any angle or distance with selected tissues or substances appearing semitransparent in color and others opaque.

Volume data is collected through a variety of means. Medicine has extensively used Computerized Tomography (CT) and Magnetic Resonance Imaging (MRI) scanners for data acquisition. Other medical imaging devices include Ultrasound Tomography, Positron Emission Tomography (PET) and Single Photon Emission Computerized Tomography (SPECT) scanners (Stytz, 1989:11). Scientists use scientific measurements and computer simulations using finite element models of stress and fluid flow to acquire volume data (Drebin, 1988:65). The methods of interest in this research are CT and MRI.

The data model used to represent the raw data in medical imaging is the voxel model (Stytz, 1989:62). A voxel is a parallel-piped-shaped element of a 3D volume. It represents the sampled region where data is acquired through CT and MRI scanning. The values associated with the voxels of a volume are the density values output by the medical imaging device.

The volume rendering technique of interest in this research is the one that casts rays through a volume of acquired data and samples the data at uniform increments along the ray to determine the color and opacity of a pixel—stopping when accumulated opacity is maximum, that is, an opaque sample encountered. This technique is known as ray casting, ray tracing, or sometimes referred to as a backward mapping algorithm. The sampling of data involves interpolation of color and opacity from the voxels surrounding each sample point. Levoy points out the main disadvantage in this technique," Since all voxels participate in the generation of each image, rendering time grows linearly with the size of the data set" (Levoy, 1990:246) (Fuchs, 1989:47).

Consequently, faster rendering methods are needed to stimulate their practical use and acceptance in the medical field. The goal in volume rendering speedup is to achieve interactive visualization of a volume.

Various volume rendering speedup techniques have been investigated in the literature over the past ten years. One focus of attention rests upon ray tracing (or ray casting) because "the calculation speed of the ray tracing method is undoubtedly one of the basic problems that must be dealt with" (Fujimoto, 1986:16). Arvo and Kirk point out that acceleration of ray tracing is achievable by execution on a collection of general-purpose computers (Glassner, 1990:249). Most implementations of fast ray tracers, however, exist on special purpose machines or strictly parallel architectures. Because it is not cost effective to limit implementation to special-purpose architectures, alternate speedup methods with a software approach are needed. This is especially true today with the interoperability of systems across networks. Interoperability allows programmers flexibility without placing unnecessary restrictions on them. All that awaits is the appropriate application to benefit from this technology.

Distributed programming is based on process interoperability. Andrews defines a distributed program as "a concurrent (or parallel) program in which processes communicate by message passing" (Andrews, 1981:50). Typically, a distributed program is executed on autonomous computers connected by a communications network (Singhal, 1991:12) (Andrews, 1981:50). Examples of distributed programming in our daily use of computers are network file servers, remote login, and electronic mail. Applications using distributed programming can benefit from high-speed computation across general-purpose computers.

1.2 *Problem*

Considering today's computing technology, what technique can accelerate the ray tracing step in volume rendering? I propose distributing the ray tracing of a volume across general - purpose workstations, like the Sun Microsystems SPARC stations to provide a solution to the speed problem of volume rendering.

This research effort investigates the possibility of dividing a volume data set among workstations and casting rays (using a front-to-back algorithm) (Levoy, 1990) through each subvolume independently. Research will show a 75 percent speedup while maintaining image quality. The goal is to speed up this particular volume rendering technique and demonstrate that ray casting can be made parallel at the subvolume level. As this parallelism can be demonstrated, so too can intermediate viewing of the volume at the subvolume level be demonstrated. The intermediate viewing is a benefit not realized with rendering the entire volume as a whole.

1.3 *Assumptions/Materials and Equipment*

The first assumption of this thesis is the availability and operability of a distributed network environment for programming distributed applications. This is available on the Air Force Institute of Technology Network (AFITNET) using the standard networking protocol, TCP/IP. AFITNET connects a wide variety of computers together, such as Silicon Graphics Inc. (SGI) and a multitude of Sun Microsystems platforms.

The Sun systems available are based on Sun's proprietary Symmetric Processor ARChitecture (SPARC). These consist of Sun SPARC-2, and Sun-4/260, all running SunOS 4.1.1. Sun provides an Open Network Computing (ONC) environment on its systems that supports distributed computing through the provision of distributed services. A few of these services are Network File System (NFS) and Remote Execution (REX). NFS provides transparent access to remote files over a network, making it appear that they are local. REX service allows you to transparently execute a program remotely among several processors (distributed programming). The way to use REX is through the **on** command. Investigation into the REX service on the SPARC workstations available to me found this service unreliable. However, the UNIX alternative, **rsh** command, proved worthy despite its inherent slower execution time (SUN, 1991).

Another assumption is the availability of quality graphics hardware and software for viewing the medical images this research will produce. The hardware available consists of the SGI IRIS 3100 and 4D monitors, Sun 4/260 with TAAC-1 monitor, and Sun 4 monitors. The software selected for viewing images is the Utah RLE library toolkit.

Lastly, medical data (MRI, CT, PET, or Ultrasound) is assumed to be available for testing. CT data of a rib section and a dog's heart, as well as MRI data of a head was obtained for this and other research.

1.4 *Scope and Limitations*

This thesis effort implements a distributed volume renderer. However, no user interface design falls within the scope of this thesis. The distributed computing is restricted to eight workstations, one per octant. To ensure flexibility, variable input data file sizes is an attribute built into the code.

1.5 *Approach/Methodology*

Instead of duplicating efforts, the intent is to borrow and/or model other design and implementations for the volume rendering pipeline steps based on availability. The four major steps in the rendering pipeline are loading the volume data into machine representation, scene processing (Udupa, 1991:45), casting the rays, and displaying the scene.

Developing a volume renderer without subdivision on a single processor is approached first. Once verification of the rendering pipeline is assessed, distributing the volume renderer can take place.

The first step into distributed programming is to become familiar with the Sun ONC environment. Investigation into ways of implementing distributed computing is accomplished. This will include analysis of remote process access to data; whether to send the subvolume data to nodes as opposed to NFS access of the entire volume.

After I selected a distributed computing paradigm, I incrementally developed the distributed volume renderer. My approach was to scope the distributed programming down for the first iteration, gaining confidence in the approach, then advancing to the desired complexity of eight remote machines.

Experimentation is done on various sizes and complexities of volume data files. Comparison of times and images between the distributed rendering and the single processor rendering can validate the thesis.

1.6 *Thesis Overview*

The remainder of this thesis is divided into four chapters. Chapter II will provide the review of literature in this research area. The subjects to review include volume rendering and methods to speed it up, as well as distributed computing. Chapter III presents my development of a distributed volume renderer including design and implementation. The results of my implementation are then discussed in Chapter IV. Chapter V presents the conclusions drawn and recommendations for future work in this area.

II. Literature Review

2.1 *Overview*

This chapter's purpose is to give background information on increasing the speed of ray tracing in volume rendering. There has been much research done into speedup of ray tracing geometric scenes, but this is not to be confused with the ray tracing of interest here—ray tracing volume data. Before examining how ray tracing speed has been improved, Levoy's volume rendering pipeline (Levoy, 1990:247) is presented in detail. Understanding the steps involved in the volume renderer will aid in seeing how its performance can be improved. To motivate its application to this research, distributed computing is discussed in detail.

2.2 *Volume Rendering*

The 3D imaging of a volume using ray tracing entails three top-level steps: loading the volume slice data into machine representation, ray tracing, and displaying the scene. The ray tracing discussed throughout this section is based on Levoy's front-to-back image-order method (Levoy, 1990:247). The ray tracing step breaks down into the following pseudo code:

```
for each voxel
   shade and classify

for each pixel
   transform from image space to object space
   cast a ray
   if it intersects  the volume
      then for each step along ray until it exits volume
         sample by interpolating surrounding voxels
         composite colors and opacities
   composite background

assign color to pixel
```

The shading and classification step (Levoy, 1988:31-33) can be considered a preprocessing step. It is detailed in Figure 2.1 (see page 2-2) as a pipeline. An appropriate shading model, such as Phong shading, can be selected for the shading step. The classification method can use the magnitude of the gradient vector for opacity fall-off effect or surface boundary enhancement.

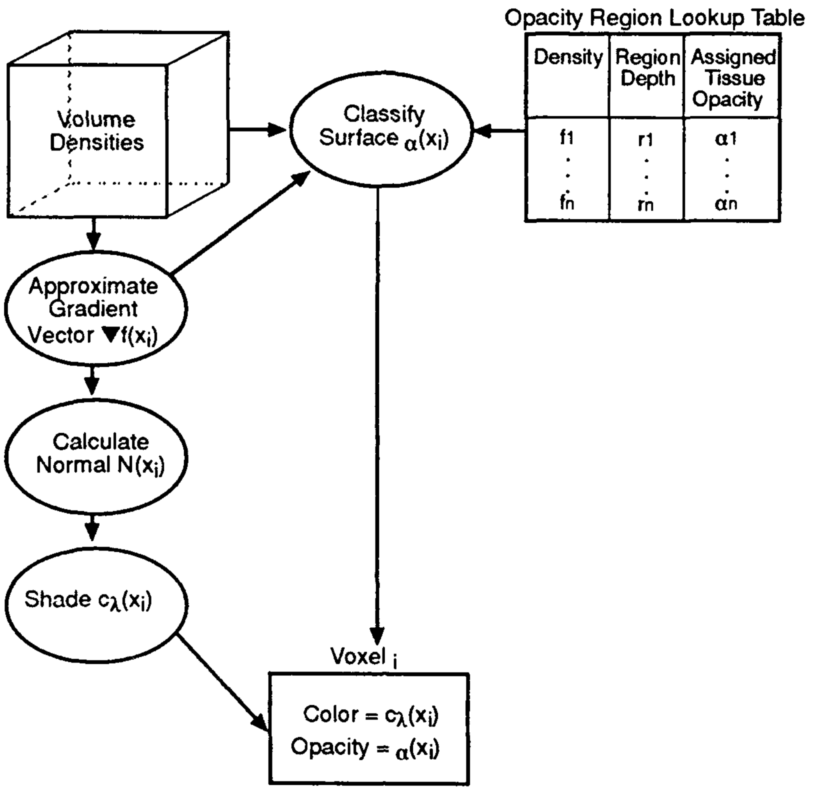

Figure 2.1 Shading and Classification Pipeline

The rest of the ray tracing steps are shown in Figure 2.2 (see page 2-3) as a pipeline. If a ray doesn't intersect a volume, then it bypasses the sampling steps, rendering the pixel with a background color and no transparency.

2.3 *Speedup Techniques*

2.3.1 *Ray Tracing Speedup.* Ray tracing is handicapped in speed by the size of the volume. This is true for the preprocessing step, but more significantly so in the case of tracing the rays. Speedup techniques for volumetric ray tracing come in various forms, either hardware (Kaufman, 1986) (Baum, 1990), algorithmic (Levoy, 1990), data structures (Levoy, 1990), reduction in rays cast (Levoy Dissertation,1989) or some combination thereof.

Hardware speedups generally involve the use of parallel architectures or custom graphics hardware to take advantage of the inherent independent nature of ray tracing or to speedup the pipeline of imaging. Kaufman reviews a newer architecture, one that is voxel-based, that is, uses a 3D cubic frame buffer. GODPA, PARCUM, 3DPPPP, and CUBE are four voxel-based

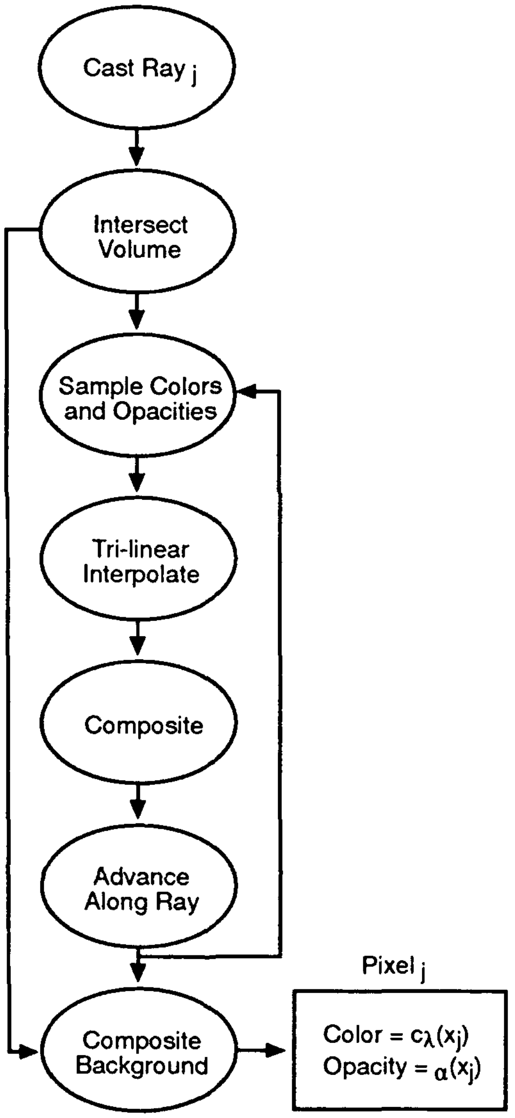

Figure 2.2 Ray Casting

architectures he surveys. GODPA and CUBE are the only two applicable for medical imaging, with CUBE coming the closest to ray traced images because it can render semitransparent images (Kaufman, 1986).

Marc Levoy presents an algorithmic approach to optimizing the ray tracing. His method is based on the observation that once a ray strikes an opaque object or has progressed far enough, the color of the ray stabilizes, so ray tracing can be terminated (Levoy, 1990:250). Levoy's data structure approach to speedup is to represent spatial coherence of the volume data in a pyramidal data structure (complete octree) and employs an algorithm to trace rays through the octree (Levoy, 1990:249). Levoy then combines both these optimizations to produce results showing a savings of more than an order of magnitude.

By reducing the number of rays traced, Levoy can produce intermediate images, progressively refining until the final image is rendered. He uses an adaptive rendering algorithm based on estimated image complexity at sample regions of the image plane—the higher the complexity, the more rays cast. His results showed that performance time was proportional to number of rays cast and precise selection of paramaters were crucial (Levoy 1989:36-40).

2.3.2 *Related Speedup.*

An important area of work, as it relates to this thesis, is the parallelization of geometric ray tracing achieved by Mark Dippe' and John Swensen (Dippe, 1984). The relevance of their work is that they parallelized at 3D space (world space) instead of 2D space (image plane), as most parallel ray tracers do, and showed a speedup. In other words, they subdivided 3D space into subregions and distributed the objects defined within the subregions across a would-be 3D array of independent computers. This method is in contrast with the more popular method of dividing the screen into subscreens and distributing them among processors (Deguchi, 1986). In Dippe and Swensen's method, rays are initially cast from the computer with the subregion containing the eye. Rays are processed (intersected with objects) in the subregions along their paths. Rays exiting one subregion are passed onto the neighboring subregion until they terminate. An equal workload among the computers is achieved through adaptive subdivision. Neighboring computers communicate via messages to determine if their work loads differ by some threshold.

Dippe' and Swensen's concept of dividing world space among processors instead of screen space for geometric ray tracing optimization can be applied to volume ray tracing. The parallel concept is dividing object space (volume) among processors for volume ray tracing optimization. Although no geometrically defined objects are contained within each subvolume, the ray does intersect with each voxel along its path when it is traced. Another difference in concepts is that object space does not contain the eyepoint. To fit in this scenario, the rays could just originate from a host computer.

Upson (Upson, 1988:61) and Westover (Westover, 1990 & 1988:9) agree that the alternative method of volume rendering, mapping the data onto the image plane (forward mapping), is well suited for parallel processing. Westover, in fact, applies the parallelization at the subvolume level concept to a forward mapping algorithm of volume rendering. But, parallelizing a backward mapping algorithm is of interest in this research. On the other hand, Upson claims the ray casting (backward mapping) method is difficult to parallelize since it "parallelizes at the pixel level" (Upson, 1988:61). Upson believes ray casting is "more efficient for conventional machine architectures" because it requires a "global shared memory" to contain the entire volume at all times throughout the algorithm (Upson, 1988:61).

2.4 *Distributed Computing*

The type of distributed computing of concern in this review is the transparent execution of an application program across a network of independent, general-purpose and special-purpose computers (Andrews, 1991:50). Peter Wayner provides an exciting scenario in the future as an example of distributed system:

> Imagine you've just walked into a hospital of the future. Everything from the heart monitors in the Intensive Care Unit to the sterile robots operating the vacuum cleaners in the hallways is computerized...a doctor approaches a computer terminal and opens a file on a patient who is resting comfortably on the fourth floor. The physician receives data from the patient's medical history, as well as current readings from bedside monitoring systems upstairs.
>
> Determining that the patient's medication needs to be changed, the doctor hits a few keys, and new prescriptions appear on the attending nurse's terminal and in the patient's file. Meanwhile, the accounting system silently tracks this activity... (Wayner, 1990:58).

The motivations for distributed computing are that it shares expensive resources, maximizes utilization of heterogeneous computers, increases computing power, and provides a fault tolerance capability (SUN, 1991).

Distributed computing allows the ever changing hardware environments to interact via standards. Currently, Sun Microsystems is an example of one company that provides a platform for distributed computing with its Open Network Computing (ONC) environment. ONC uses established industry standards to provide an open architecture for portability (SUN, 1991). Tools are provided by ONC allowing the programmer to control and implement distributed computing.

Gregory Andrews (Andrews,1981) presents a detailed description of distributed programming. He defines distributed computations as concurrent programs in which processes communicate by message passing across shared channels. There are several ways these processors can interact in a distributed system; he presents the paradigms for these interactions with examples of each. Filters, clients, servers, and peers are what he describes as the four basic kinds of processes in a distributed program. The important ones for this thesis are clients and servers. Andrews defines them as "A client is a triggering process; a server is a reactive process" (Andrews, 1981:51).

One of Andrews' process-interaction paradigms of interest involves server replication by dividing the task at hand into independent subtasks that are solved concurrently (Andrews, 1981:82). Andrews classifies the interaction paradigm in mind here as "replicated workers sharing a bag of tasks" (Andrews, 1981:51). One of his solutions to the divide-and-conquer problem is to use one *administrator* process that generates the first problem and gathers results from several *worker* processes. The *workers* solve the subproblems that they receive from a shared channel, bag. He makes an important point, that often in practice a channel can have only one receiver, but this can be remedied by having the *administrator* simulate a shared channel by also acting as a server process with which the *workers* communicate.

2.5 Conclusions

The ideas presented for speedup of volume rendering are a small portion of the pool of thought in this area. It is well acknowledged that faster medical imaging methods are needed and research continues in this area. These methods, as discovered, can be incorporated into existing heterogeneous networks through distributed computing. The time is here to reap the benefits of distributed computing with medical imaging, if not for cost savings alone.

2.6 Summary

This section provided the background necessary to understand my development contained in the next chapter. With an understanding of the ray tracer process, dividing object space, and the divide and conquer distributed paradigm, my solution to speedup of the ray tracer can be realized.

III. Design and Implementation

3.1 *Overview*

The sequence of events in developing the Distributed Volume Renderer (DVR) software was an iterative process through analysis, design, implementation, and testing. This process was applied first to developing the Volume Renderer (VR) on a single processor, then to developing a distributed version. The incremental approach allowed me to focus on the VR requirements separately from the distributed ones.

This chapter presents the final design and implementation in the order they were accomplished. The VR development is discussed in the first section, while the distributed development follows. Before these two abstractions are presented, some generalities about my design approach and implementation specifics must be understood.

3.1.1 *Design.* My decision to program in an object-oriented language drove my design toward an object-oriented approach. Although an object-oriented design was not necessary, it was advantageous to do so. Because the classes and methods were developed with the data abstraction and inheritance capabilities of C++ in mind, the coding step became simpler.

3.1.2 *Implementation.* The system was programmed in C++ using the GNU C++ compiler, version 1.37. All compilations of code were done on a Sun 4/260 workstation running Sun OS 4.1.1. Some of the system libraries used were the RLE library, version 3.0, and the AFITCOM library. RLE images were viewed using the appropriate RLE viewing utility for either a Sun monitor, TAAC-1 monitor, or Silicon Graphics monitor. The VR code was developed and tested on Sun 4 and SPARC workstations. The DVR code was developed on Sun 4 workstations and tested on a network of Sun SPARC workstations with a MicroVax III file server.

3.2 *Volume Renderer*

The goal of the VR development was to design and implement a front-to-back image-order volume rendering algorithm using an object oriented paradigm. Levoy (Levoy, 1990 & 1988) provides the details for this type of VR from which development was based.

3.2.1 *Design.* The preliminary design of the VR is encapsulated in Figure 3.1 (see page 3-2); it shows the top level data-flow diagram. The VR reads control parameter files which

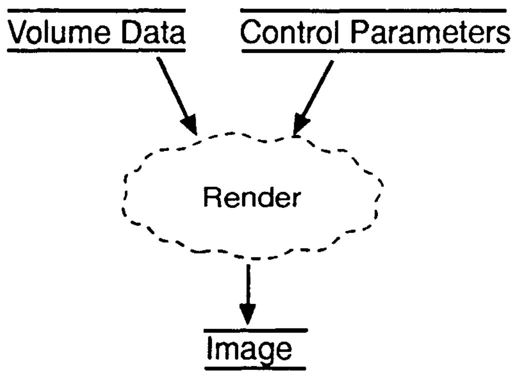

Figure 3.1 Volume Renderer Top Level DFD

effect the instantiation of objects within the program. The VR also reads the data values from the data files into a volume object. The volume is rendered using Levoy's (Levoy, 1990 & 1988) method and a pixel image is produced in RLE format.

The decision for external control input to the system was borrowed from the General Purpose Renderer (GPR) developed at AFIT. As in the GPR, a control file allows the user to set parameters, such as the eyepoint and light source, at program runtime. This idea was extended to include parameters identifying characteristics about the volume data, such as data type and data size. The formats for these two control files are in Appendix A.

The other external data depicted in Figure 3.1 is the volume data. Based on available data at design time, several constraints were placed on the volume data input. The constraint on data type limits the data to be within the range of 0 to 255, corresponding to one byte in length. All test data was found to have this characteristic for the data values. Additionally, all CT and MRI data is assumed to consist of a set of files, each file representing a separate slice in the sequence as acquired. The sequence of data files must reflect data continuity from slice to slice, as well as across a slice of data. This does not mean that there must be zero interslice distance; rather, the slices must be consecutive or the interpolation done between slices will not be accurate. The size constraint on data is an implementation issue requiring no constraint at design time.

To pick out the objects and methods from the problem space that would become classes and class relationships in my class diagram, a closer look at the rendering algorithm was necessary. Levoy's volume rendering pipeline (Levoy, 1988 & 1990), reproduced in Figure 3.2 (see page 3-3) provided me with the objects and some methods required specific to a volume renderer. The objects are a volume, voxels, a ray, and pixels. Some methods are found in the action steps

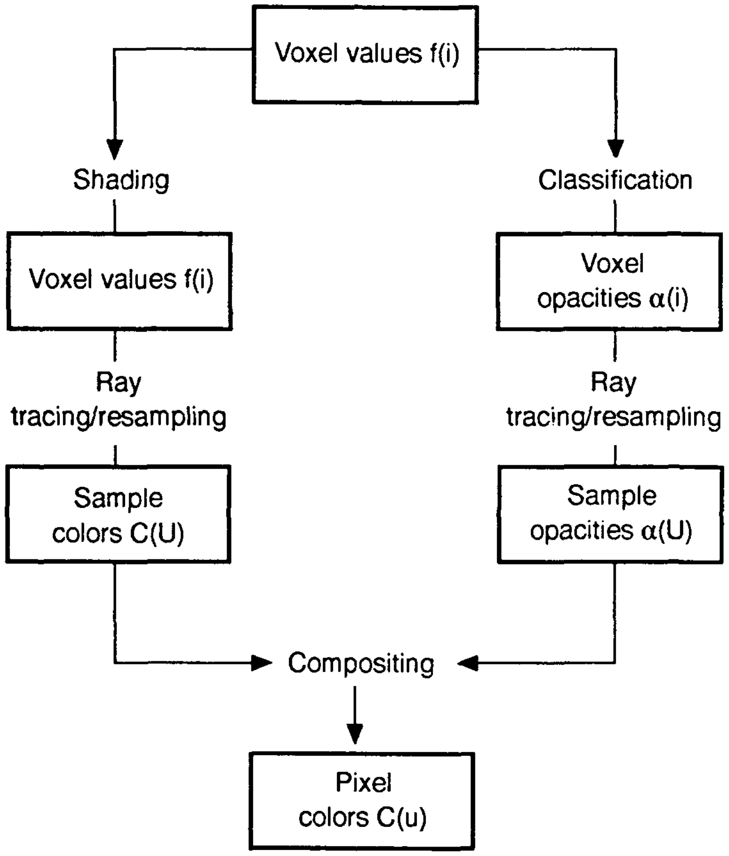

Figure 3.2 Overview of Volume Rendering Algorithm

shown in the pipeline, and are associated with the objects doing the action. For example, a volume is *shaded* and a ray is *traced.* Of course, this is just an overview; other methods came from Levoy's front-to-back algorithm (Levoy, 1990:248). Additional objects and methods required for 3D graphics were borrowed from the GPR design.

Figure 3.3 (see page 3-4) shows the final design of the VR's classes, class relationships, and class utilities. The notation in this diagram is based on Booch's notation (Booch, 1991:158-161) and defined in Figure 3.3's legend. The **ALPHASHADE-BUFFER** class *inherits* the **BUFFER** class, and the **BUFFER** class *instantiates* one or more of the **PIXEL** class. Simply put, an AlphaShade-Buffer is a "kind of" (Booch, 1991:56) Buffer, while a Buffer "contains" (Booch, 1991:116) many Pixels. In this context an AlphaShadeBuffer differs from just a Buffer in that both color and opacity play a part in coloring a Pixel. A preliminary design decision allowed for more than

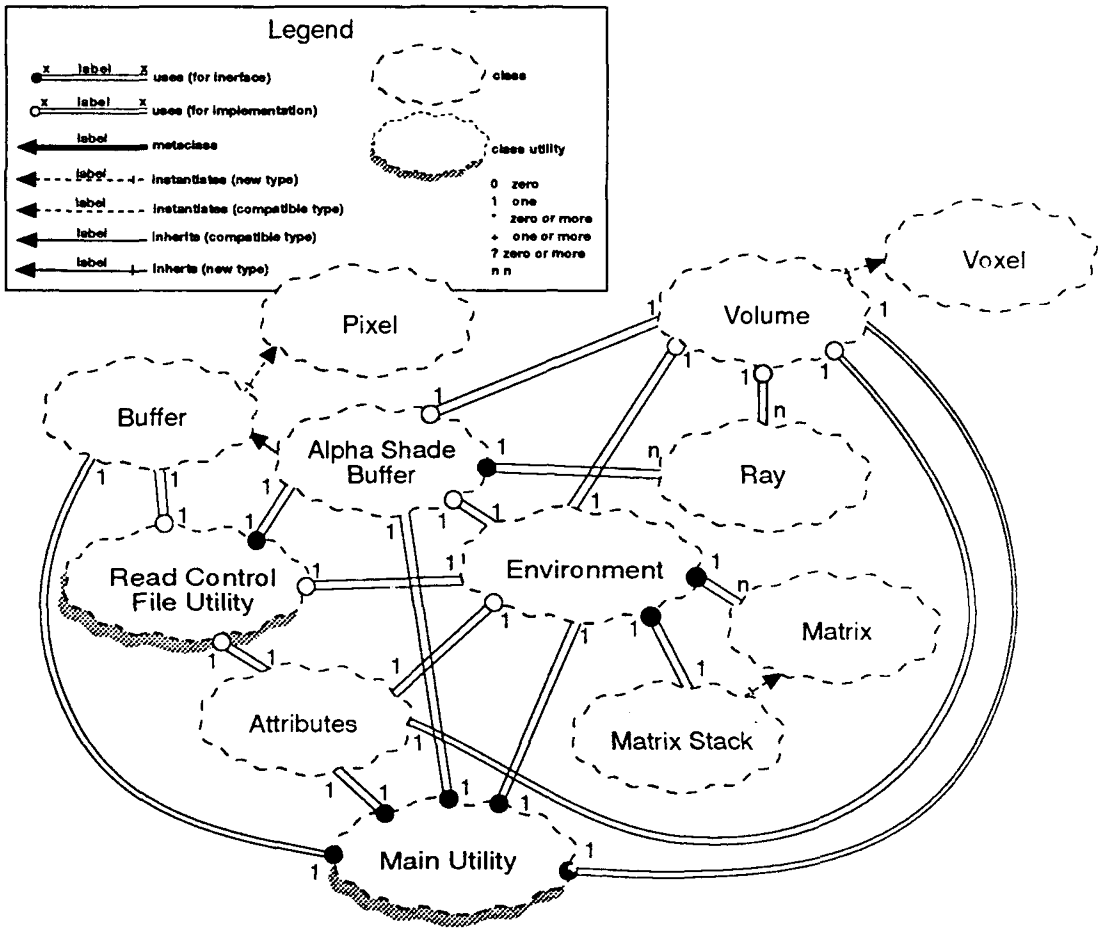

Figure 3.3 Volume Renderer Class Diagram

one buffer type, that is, a different buffer for each shading algorithm. Thus, the justification for Buffer as a super-class.

As was mentioned, some classes and methods were borrowed from the GPR and incorporated into the VR. These classes are the ENVIRONMENT, ATTRIBUTES, MATRIX, MATRIXSTACK, BUFFER, and PIXEL classes. The reason these classes were selected for reuse was due to the common computer graphics methods (primitives) they contained. For example, I required a method to perform Phong shading, and the ENVIRONMENT class supplied this method already.

Some methods supplied by GPR, however, were not suitable. For example, I needed to modify the viewing methods within the ENVIRONMENT class since the VR depends on a parallel orthographic projection, as opposed to a perspective projection. The viewing method used is shown in Figure 3.4 (see page 3-5) (Levoy, 1990:247). The distinction of this viewing method

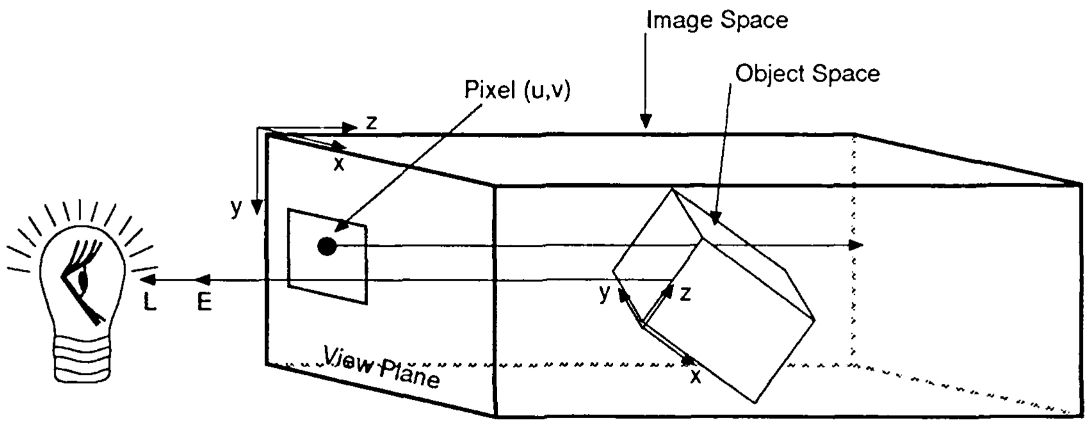

Figure 3.4 Orthographic Projection

comes about when Phong shading is performed. Here the direction of the eyepoint and light is constant for all points in object space (Levoy, 1988:31). During ray casting, all rays originate from a pixel in image space and have a parallel direction through image space, perpendicular to the view plane. The parallel orthographic viewing methods transform ray points from image space to object space for sampling of the volume.

3.2.2 *Implementation.* My implementation of the VR pipeline, as shown in Figure 3.5 (see page 3-6), varies slightly from Levoy's pipeline, other than showing more detail. The figure gives an overview of the VR algorithm by describing the three major class methods in the sequence they occur. These three steps are shade and classify a volume, shade the pixels of a buffer, and store the buffer as an RLE image file. One difference in this implementation design from Levoy's lies within the choice of data structures. I incorporate colors, opacities, and density values into a single object, Voxel, which is the element of the three-dimensional array object, Volume, instead of his suggested use of three separate arrays. Because Levoy promotes the use of several arrays, his shading and classification steps are done independent of and separately from each other. This is unnecessary with my design, since shading and classification can be done jointly while stepping through the *single* array of voxels. In fact, it is more efficient to do them together since both require the calculation of a gradient vector at each voxel.

An implementation of Levoy's front-to-back rendering algorithm was found in Ohio State University's graphics software package for visualization, apE, version 2.0. apE contains a tool

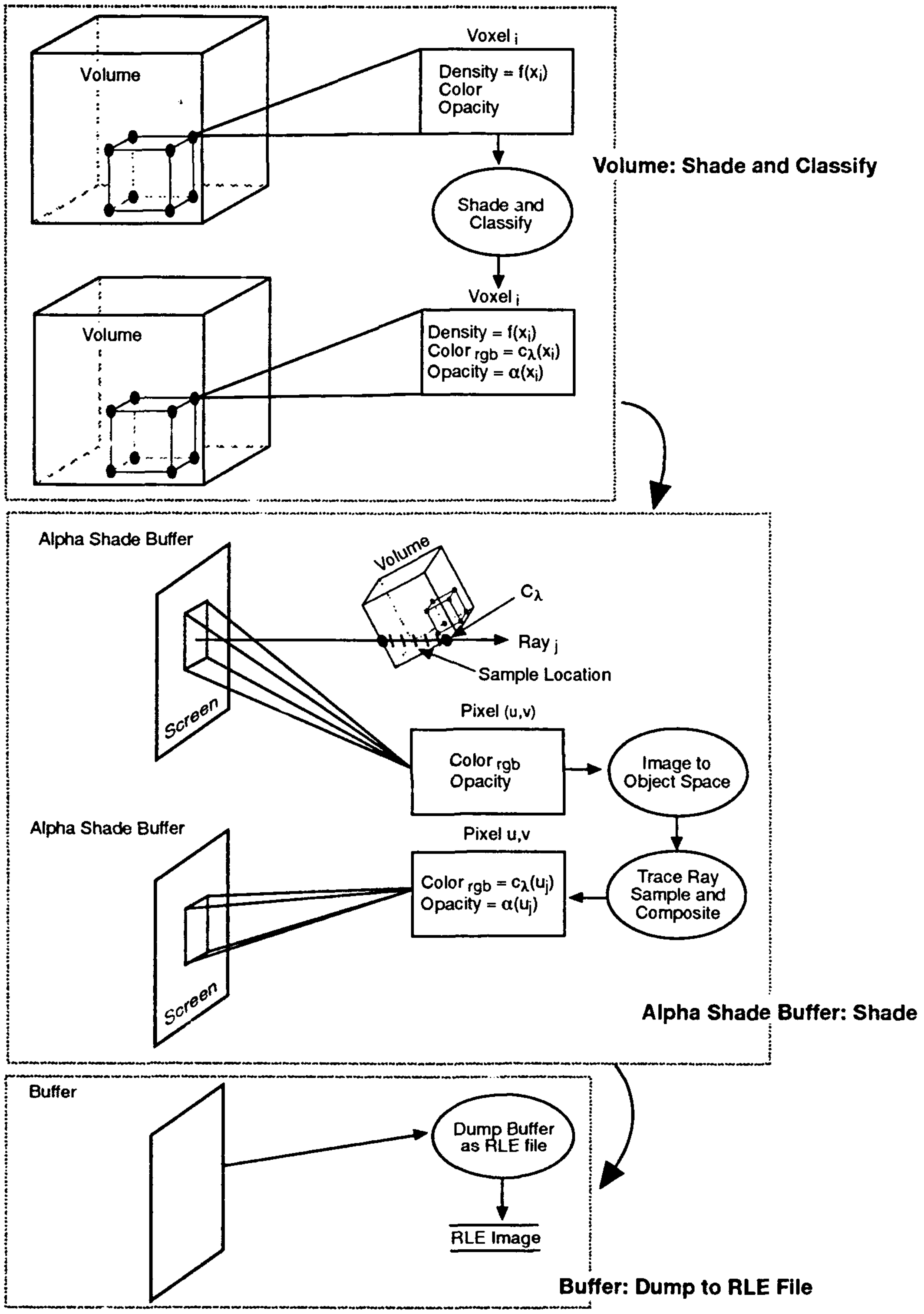

Figure 3.5 Volume Renderer Pipeline

called "opac," which is a volume renderer that uses Levoy's method (apE, 1990:88). My implementation uses "opac" tool source code wherever possible.

There is a difference, however, between Levoy's and apE's equations for color and opacity composition along the ray. Levoy relates the color and opacity sampled at a location along the ray $(C_s$ and $\alpha_s)$ to the color and opacity of the ray before the sample $(C_{in}$ and $\alpha_{in})$ with the following transparency formulas:

$$C_{out} = C_{in} + (1 - \alpha_{in})(\alpha_s)(C_s) \tag{1}$$

$$\alpha = \alpha_{in} + (1 - \alpha_{in})\,\alpha_s \tag{2}$$

where the initial values for C_{in} and α_{in} start at zero. After all samples along the ray have been processed, a fully opaque $(\alpha = 1)$ background color is composited with the last color and opacity computed for the ray. This last step forces the final α_{out} equal to 1, making the normalization step

$$C = \frac{C_{out}}{\alpha_{out}} \tag{3}$$

unnecessary. apE's implementation, and mine, differs from Levoy's by initializing the opacity to one and uses the following transparency formulas:

$$C_{out} = C_{in} + (\alpha_{in})(\alpha_s)(C_s) \tag{4}$$

$$\alpha_{out} = \alpha_{in} - \alpha_{in}(\alpha_s) \tag{5}$$

Despite the differences in these equations, the result is the same; only the initial values of opacity changes the equations. Levoy must subtract the α_{in} value from 1 in Equation 1 to get the *remaining* opacity level of the ray thus far. This value is then used to scale the sampled color contribution. apE's implementation uses α_{in} to hold the *remaining* opacity level all along. Thus, apE's implementation is more efficient because it involves two less subtractions in each step. This is a significant savings considering how many samples are taken along a ray, and that there are as many rays as pixels.

3.3 *Distributed*

3.3.1 *Analysis.*
The goal of the DVR was to produce a faster VR by dividing the task among eight processors. This translates into having eight remote processes (servers) rendering subvolumes concurrently with a local process (client) in control that combines the results. All this coordination would require interprocess communication via message passing. As discussed in Chapter II, Andrews classifies the process interaction paradigm in mind here as "replicated *workers* sharing a bag of tasks" (Andrews, 1981:51). Using Andrews' terminology applied to this problem, the DVR must have an *administrator* process subdvide the volume rendering among eight independent *worker* processes, each independently solving subproblems (Andrews, 1981:86). The *workers* must then send their results back to the *administrator*, whereupon the eight results are combined into a final result. In this design there is no recursion within each of the *workers*, in which case there would be an adaptive subdivision of the volume.

3.3.2 *Approach.*
I developed the DVR using an incremental approach that produced prototypes with increased functionality. The first iteration began with a single client and single server just communicating across a network. The next step was to have the server execute the VR developed previously, that is, a *single* server rendering the entire volume. The next iteration was a significant step, in that it divided the volume and communicated to more than one server. This iteration divided the volume in half, rendering an image for each half, but no compositing of the halves was accomplished. Another iteration advanced to eight servers where the volume was divided into octants as shown in Figure 3.6 (see page 3-9). The final iteration had the client composite the eight octant images.

3.3.3 *Design.*
A top level data-flow diagram of the final DVR is shown in Figure 3.7 (see page 3-9). The change between Figure 3.7 and Figure 3.1 (see page 3-2) is reflected in the multiple RLE files created. This is because each server generates one. I made the design decision to have the servers generate an RLE file and pass the file name in a message, versus passing the entire Buffer as a message to the client, for several reasons. The major drive was to avoid tying up channel communication between client and server until such a large amount of data could be received. Instead, the client could gather the data from a file when ready. Furthermore, the message size required for a Buffer would necessitate attention to synchronization and channel capacity issues, which could further complicate the message passing. Lastly, the process of dumping the Buffer as an RLE file was already encoded into the VR; no change to

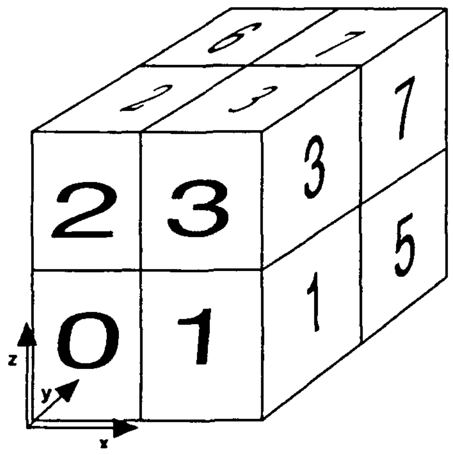

Figure 3.6 Octant Subdivision

the VR was necessary. However, an additional method to read an RLE file into memory was necessary. The important benefit of this design decision is the ability to view the volume in stages. Each RLE image gives a cut-away view of the big volume by generating an image of just the octant.

A better understanding of the interaction between client and servers is shown by a level 1 data-flow diagram in Figure 3.8 (see page 3-10). The client is the local process and the servers are the remote (identical) processes. This figure also shows dedicated communication channels between the client and each server. The channels are used for sending and receiving messages. The control parameter input designed at the client side reflects the user specification of which servers to distribute the rendering on.

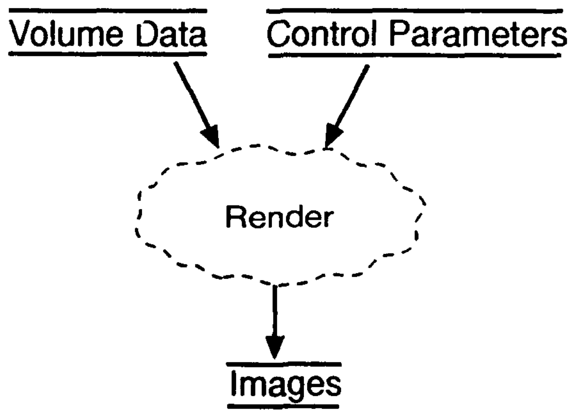

Figure 3.7 Distributed Top Level DFD

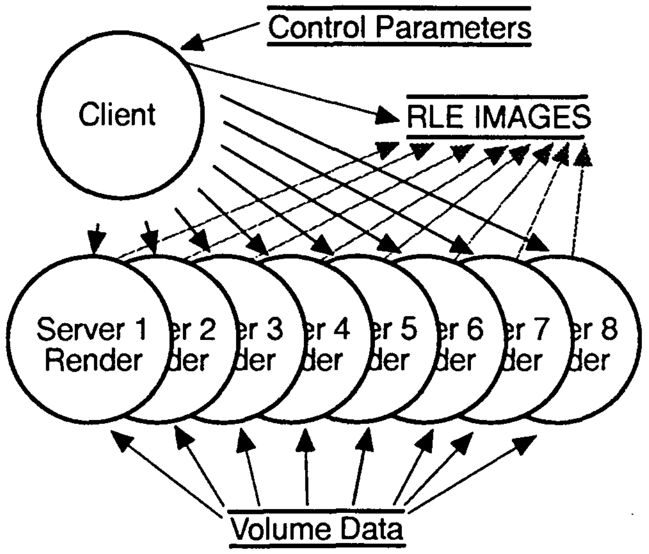

Figure 3.8 Distributed Level 1 DFD

My design decision, depicted in Figure 3.8 where volume data and control parameters from the client are input into the servers, stems from my design goals. These goals were to minimize the changes to the VR already implemented and minimize the amount of channel traffic. Of course, this decision was influenced by the distributed architecture—a central file server for networked workstations.

The client's classes are depicted in Figure 3.9 (see page 3-11) while the server's classes remain the same as those shown in Figure 3.3 (see page 3-4). Since the design parallelized at the data (volume) level instead of somewhere within the rendering algorithm, the client shares only a few classes with the server. These classes are the ones from which the server results are instantiated, namely the BUFFER and PIXEL classes. I decided to create a subclass of the BUFFER different class from the ALPHASHADEBUFFER class. The reason is the function of a Buffer at the client side is to shade its Pixels by compositing the Buffer results from the servers, not by casting Rays into a Volume. This fundamental difference in purposes led me to create the ALPHABUFFER class for the client.

The NETWORK class, seen in the figure, was needed for the client process because the simultaneous communication with eight servers can become rather complex. Thus, a separate class to handle the state of the servers is justified. From the server perspective, the simplicity of communicating with a single client did not warrant a separate class description. Instead, the design dictates the server handle this within its main routine.

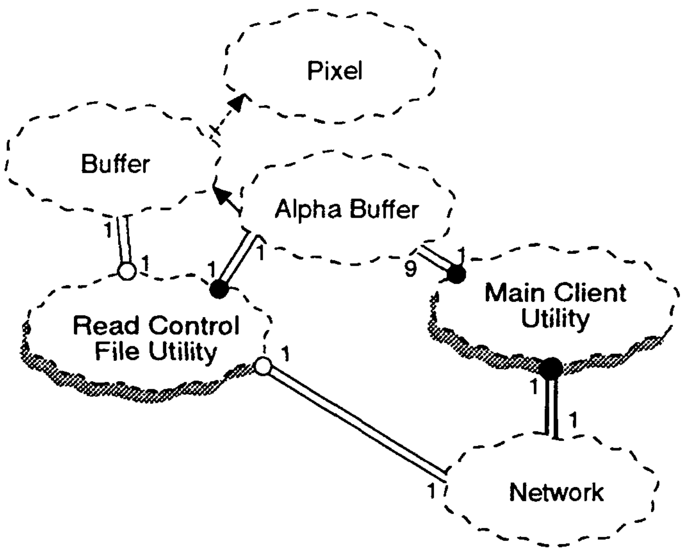

Figure 3.9 Distributed Client Class Diagram

3.3.4 *Implementation.*

The DVR implementation was greatly influenced by the decision to use an established AFIT message based communications library, AFITCOM. Since the AFITCOM library is socket based, communication was constrained between any two processes via a two-way channel using read and write calls. The AFITCOM library was designed for applications characterized by a single local host communicating with several remote hosts. This fits in nicely with the DVR design goal, one client (local host) communicating with several servers (remote hosts). As a side note, since the AFITCOM library was implemented in C, interface to these routines was accomplished via the C++ "extern C" utility and some minor modifications to the library.

Because the AFITCOM library is socket based, it is necessary to have a remote process already running and established with its socket. The implementation of starting a remote process was accomplished in UNIX by using a pipe to a remote shell, **rsh**, command. The rest of the network setup was modeled after another application using AFITCOM and written by the developer of AFITCOM (Clay,1991). The pipeline for network communication between client and server is shown in Figure 3.10 (see page 3-13).

As was discussed in the approach section, once the network communication was implemented, the next step was to implement subdivision of the volume. I will skip discussion of the volume division into halves and jump to discussion of the division of the volume into octants. This is because division into halves was just an intermediate step to experiment with distributed programming. The subdivision of the volume required substantial changes to the VOLUME and RAY classes and their methods. Not all the changes can be addressed here, even though a majority of the effort was spent modifying these classes.

I discovered the division of the volume into octants was not a simple splitting along each axis. The shading and classification step of the VR uses an approximation to a gradient vector for the calculation of the normal vector at each voxel. It approximates the gradient vector by using a central difference of surrounding voxel values. To correctly render the image, I included a pad, one voxel wide, with each octant from its three bordering octants. Figure 3.11 (see page 3-14) shows how this produces a duplication of voxels within each octant along the volume divisions, but precautions are taken against resampling the pad space of an octant. I will refer to the octant with pad as the "padded octant" while "octant" refers to the volume subdivision without a pad. The padded octant 1 is shown shaded in the figure.

The pad space is used again for accuracy during the trilinear interpolation step done at sample points along a ray. If the sample point lies within the octant's bordering voxels, then the pad is used to make up the eight surrounding voxels for the trilinear interpolation of their colors and opacities. Otherwise, the pad is ignored during the volume rendering. For example, ray intersection tests only consider the boundaries of the octant, not the padded octant.

In compositing the color and opacity of a ray, special consideration had to be made for the case where a ray exited an octant on a face that bordered another octant behind it. In this situation, no background color can be composited into the ray, because the resulting opacity will always be zero. A zero opacity from a forward octant would allow no color contribution from the octant behind it during composition of the buffers.

Once the particulars of volume subdivision and rendering of the octants was accomplished, focus switched to the client to accomplish the compositing of the eight buffers. Each RLE file stored by a server contains a color and opacity value per pixel reflecting the final color and opacity (C_{out} and α_{out}) of the ray cast from that pixel through that server's assigned octant. The composition of the buffers, then, needs to use the same concept as the transparency formulas given in Equations 4 and 5 (see page 3-7) with a slight change to the sample opacity terms. This change is that the opacity term of every buffer, except the first buffer, must be subtracted from 1 to get the true opacity value of the octant, instead of the remaining opacity left behind the octant. The distributed transparency formulas then become as follows:

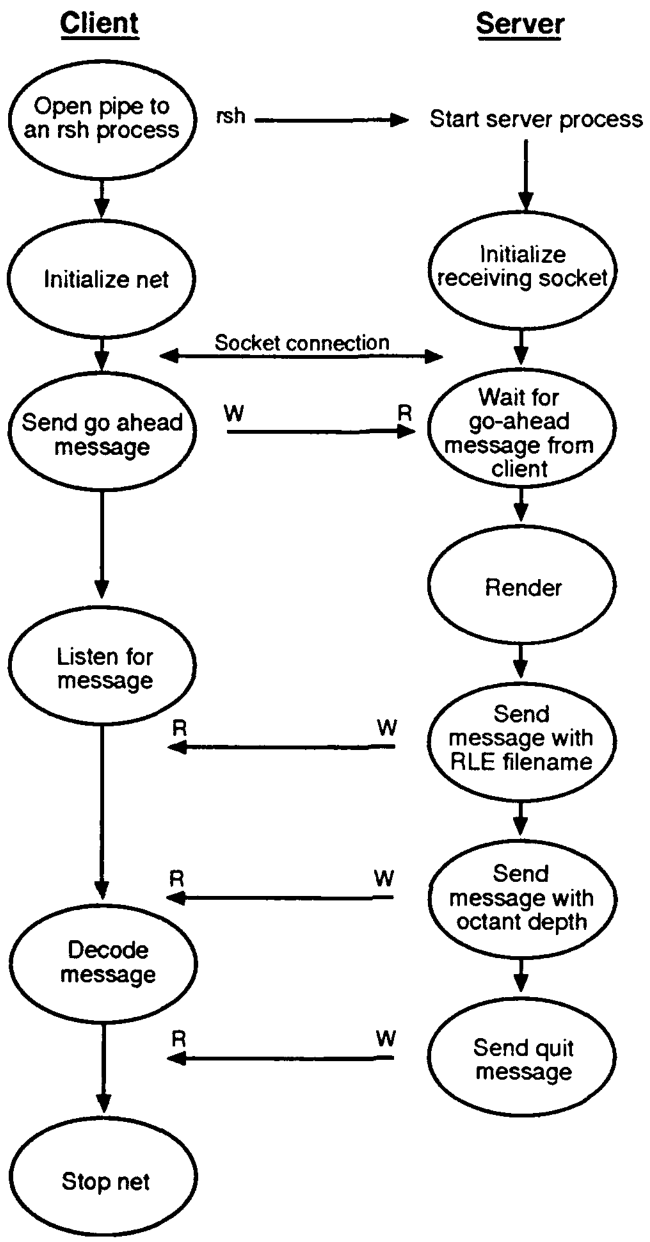

Figure 3.10 Client-Server Message Passing

$$C_{out} = C_{in} + (\alpha_{in})(1 - \alpha_s)(C_s) \tag{6}$$

$$\alpha_{out} = \alpha_{in} - \alpha_{in}(1 - \alpha_s) \tag{7}$$

where C_{in} and α_{in} are initialized to the pixel values of the closest octant's buffer and C_s and α_s are the pixel values of the next closest octant's buffer. Thus, the order of the buffers

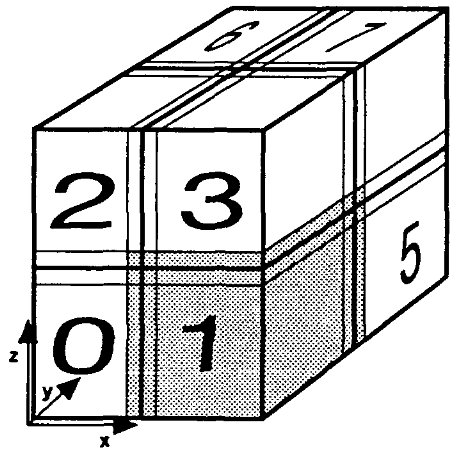

Figure 3.11 Padded Octant

must also be in front-to-back image-order for compositing. This ordering was achieved by implementing a bubble sort algorithm on the buffers based on the depth in Image space of each servers's octant.

My implementation loops through the buffers in depth order, and within this loop, loops through each pixel of a buffer compositing two pixels at a time. I realized a way to speedup the inside pixel loop for compositing by making several observations about the rendered octants. One point is that a majority of pixels from rendering an octant would contain the background color. So, if a front pixel and a back pixel are background or if either one is background, no composition needs to take place; one or the other color is used. Another point is that since a ray begins with no color, it is possible to terminate with no color, if it exists on a shared face. In this instance, no composition needs to take place either.

3.4 *Summary*

This chapter presented the detail of the software development for this research. It opened with background about design and implementation common to the both parts of the effort. Next, the specifics about the VR development were summarized. Concentration was more on the last section which dealt with the distributed development, because it was primarily original work. With this knowledge about the implementation of the system understood, the next chapter will discuss the results.

IV. Results

4.1 *Overview*

The goal of the VR development was to obtain an RLE image from CT, MRI, or test data that is correct. It was not an objective to fine tune the selection of isovalues, their opacities or transition regions in an effort to get the best appearing image. Instead, the effort was to produce a believable 2-D projection of a semitransparent volume. Since the accuracy is hard to determine from a source which I am not able to examine personally for comparison, the judgement of accuracy is left for the experts. A believable image is one which portrays no obvious anomalies and realistically appears like the object scanned with the desired transparency shown.

The primary goal of the DVR development was to obtain an RLE image (from the same data as the VR used) that appeared the same as the image generated from a single processor. The other goal was to show a savings in computation time over the time it takes the single processor VR to generate an image from an identical view.

This chapter presents the results obtained from executing the VR and the DVR with CT data and artificial data. The results demonstrate that the differences in images generated from the CT chest are indistinguishable. However, visible differences in the artificial data images exist. These differences are investigated in this chapter as well. The renderings of CT dog heart demonstrate cut-away viewing of an entire volume. Finally, barcharts and tables are presented to demonstrate the considerable savings in processing time achieved by distributed computing. But first, some background is given on the test data used.

4.2 *Test Volume Data and Output*

The types of data used for testing were CT data of a chest region, CT data of a dog heart, and artificial data of 3 concentric boxes. The CT chest data set consists of a 240 x 164 x 175 (X x Y x Z) volume using an interslice distance of 5 mm for 30 slices of data. The CT dog heart data set consists of a 202 x 132 x 144 (X x Y x Z) volume using a 0 mm interslice distance. The box data originated from 20 slices with an interslice distance of 10 mm yielding a 200 x 200 x 210 (X x Y x Z) volume.

The test runs consisted of isometric views from various rotated angles and top, right, left and bottom side views. The light source was always coincident with the eyepoint. All runs of the CT chest data were rendered onto a 244 x 180 (X x Y) pixel plane. The CT dog heart data was rendered onto a 200 x 200 (X x Y) pixel plane. The boxes were rendered onto a 240 x 240 (X x Y) pixel plane.

The naming convention used to label these images and timings correspond to the viewpoint of the image taken. The numbers represent the Z-height of the eyepoint in object space or X/Y position along top or bottom of volume. Letters are abbreviations for view, such as, RS for Right Side, 3F for 3D Front, and L0 for Left Side at height 0. Those with just numbers and no letters were taken from the right front corner at the height reflected by number. The abbreviation SP means Single Processor and Distr means Distributed. Results were saved within appropriate directories as a file called res.rle or spres.rle, where sp designated the single processor image.

4.3 *Images*

4.3.1 *CT Volumes.*

The results of the single processor VR and the DVR execution with the CT chest volume can be seen in Figures 4.1 and 4.2 (see page 4-3). The images were rendered with skin and fat semitransparent. Each picture displays a pair of images with the single processor image to the left matched with its distributed image to the right. The pictures clearly show no visible inconsistencies between the two rendering methods. All the images seen in Figure 4.3 (see page 4-4) were generated using the DVR and demonstrate the consistency of the DVR in producing correct images of twenty degree increment rotations for the chest.

The images seen in Figure 4.4 are of a CT dog heart. The eyepoint was defined inside the volume and the volume was rendered by DVR. These images demonstrate the capability of DVR to render cut-away views of the entire volume.

The final images of the chest, shown in Figure 4.5 (see page 4-5), illustrate one of the benefits gained by distributed processing—the ability to intermediately view the volume. These images are the subvolume images rendered by each of the eight processors. The black areas in some of the images represent the octant borders where further compositing with the octant behind it must be done at the administrator as part of the final buffer composition step. An added benefit realized by these octant images is the cut-away viewing capability.

4.3.2 *Artificial Box Volume.*

The results of the single processor VR and the DVR execution with the artificial box volume can be seen in Figures4.6 and 4.7 (see page 4-6). All images in Figure 4.6 were rendered with the outer two boxes semitransparent and the center box opaque. The single processor images in Figure 4.7 were rendered with the inner two boxes

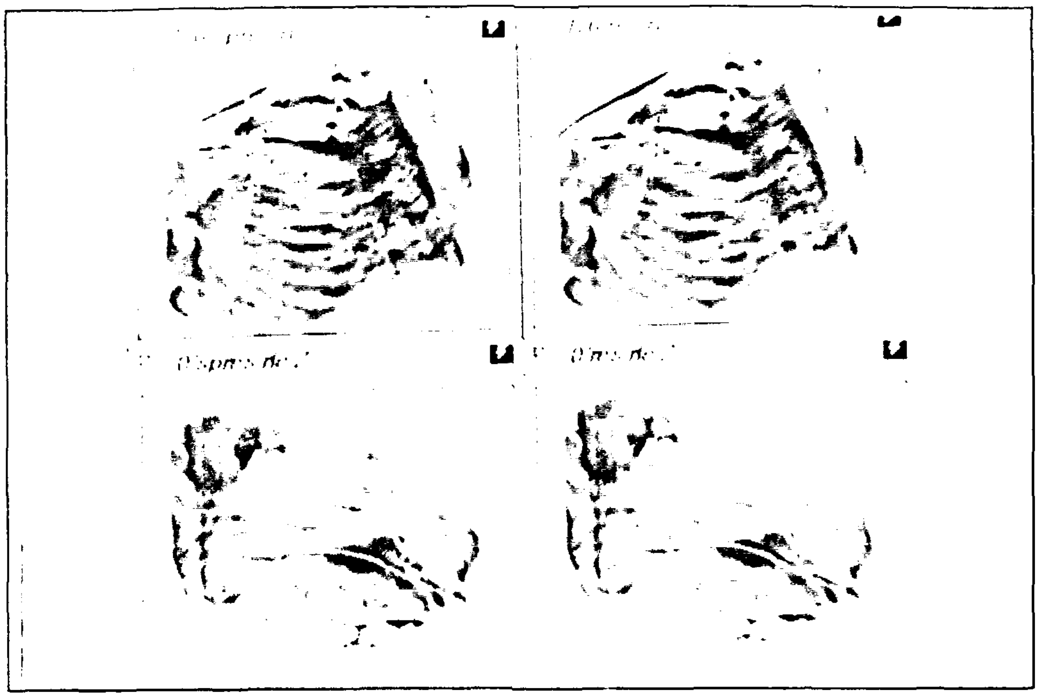
Figure 4.1 CT Chest Volume Rendered by Single Processor (left) and Distributed Processors (right).

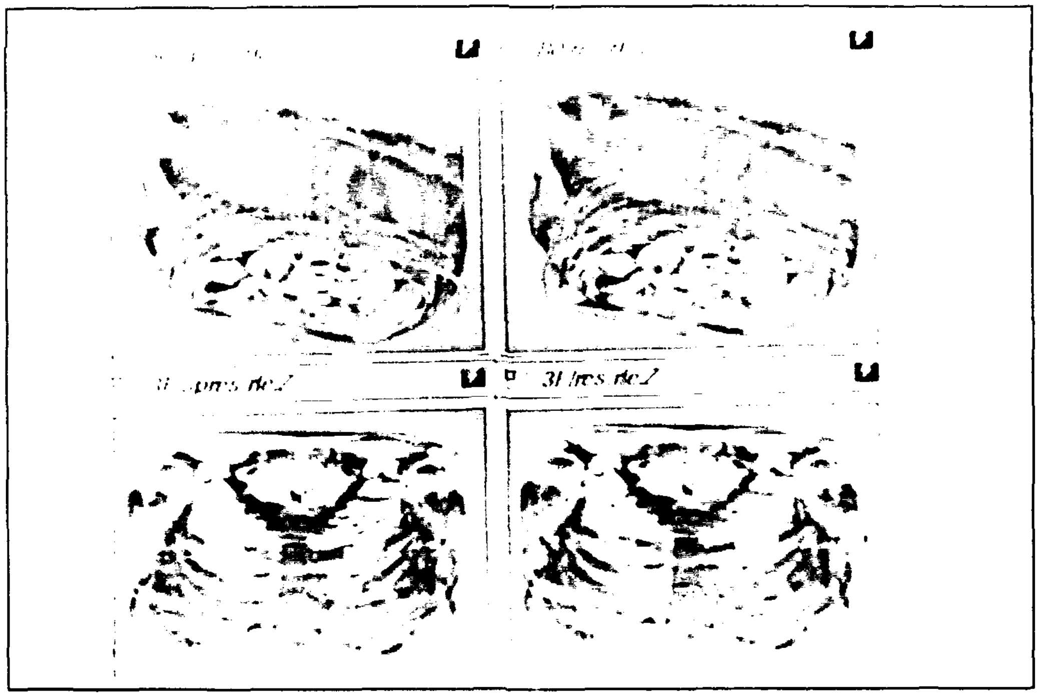
Figure 4.2 CT Chest Volume Rendered by Single Processor (left) and Distributed Processors (right).

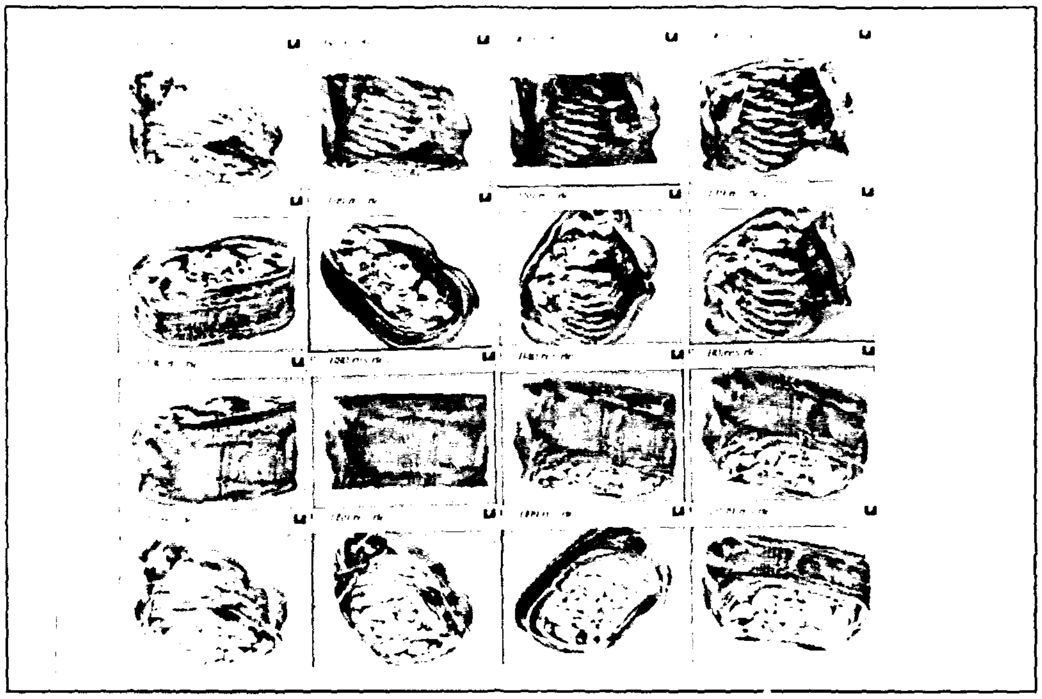
Figure 4.3 Rotated Views of CT Chest Volume Rendered by Distributed Processors.

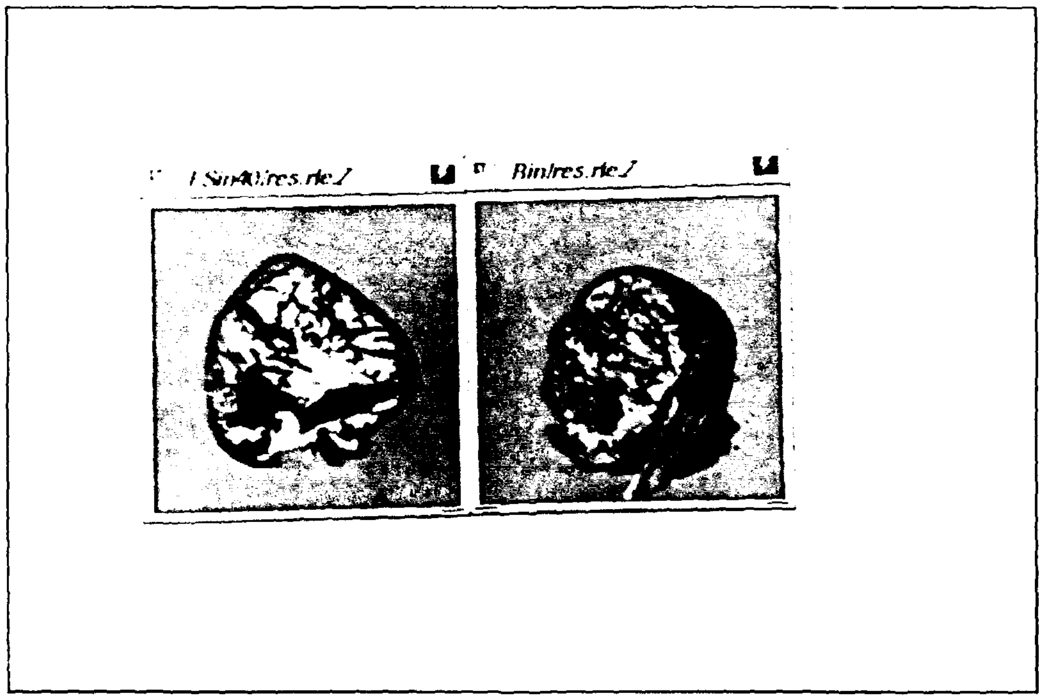

Figure 4.4 Cut-Away Views of CT Dog Heart Volume Rendered by Distributed Processors.

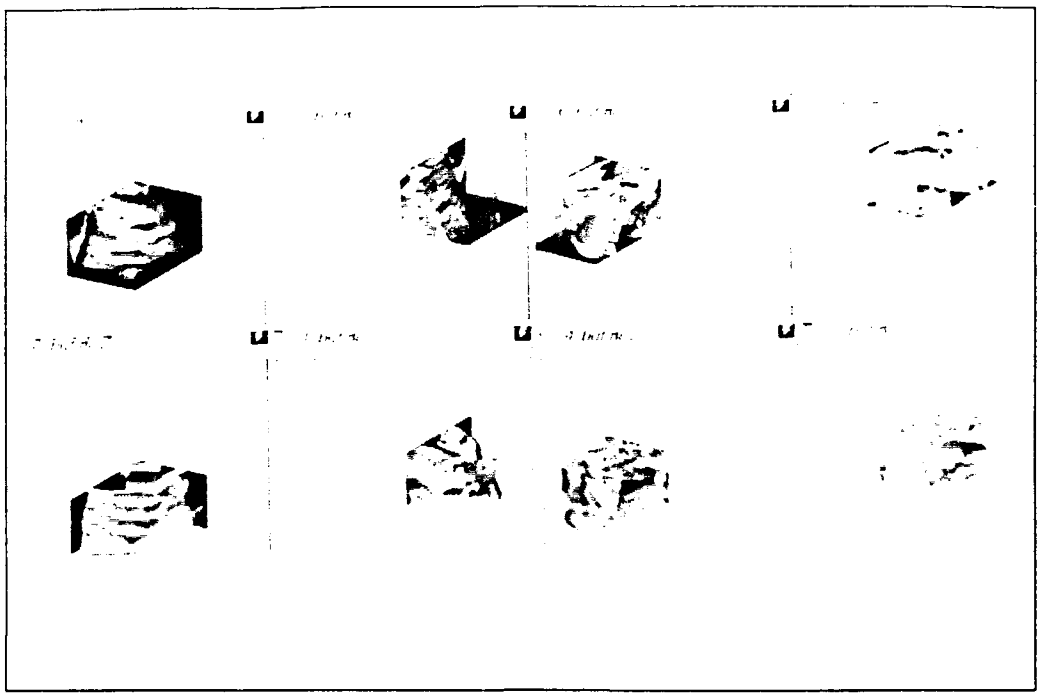

Figure 4.5 CT Chest Distributed Volume Rendering of Octants.

opaque and the outer box semitransparent. The distributed images in Figure 4.7 (see page 4-6) were rendered with the outer two boxes semitransparent. Again, each picture compares the single processor rendered image to the left with that of the distributed to the right. The front view and side view as seen in Figure 4.6 reflect no inconsistencies. However, as seen in Figure 4.7, (see page 4-6) isometric views produce some anomalies. To understand where the box borders lie in relationship to the octant divisions see Figure 4.8 (see page 4-7).

The inconsistencies apparent with the artificial data have been a challenge to understand, but several conclusions can be drawn. The anomalies are consistently along the octant (subvolume) borders, except for straight-on views of the volume where no anomalies are seen. The major differences between the CT data and the artificial data is that the artificial volume is uniform within each of the boxes. This can explain why the border problems are so evident in the box images. The reason is that any incorrect sampling (like oversampling) done along one ray into a uniform volume will produce an incorrect color for that pixel and will be very apparent next to the adjacent correctly sampled parallel rays which generate the correct pixel color. Whereas, in nonuniform volumes, such as the chest volume, adjacent ray colors can vary widely depending on the body tissue encountered. Since the anomalies do not exist with

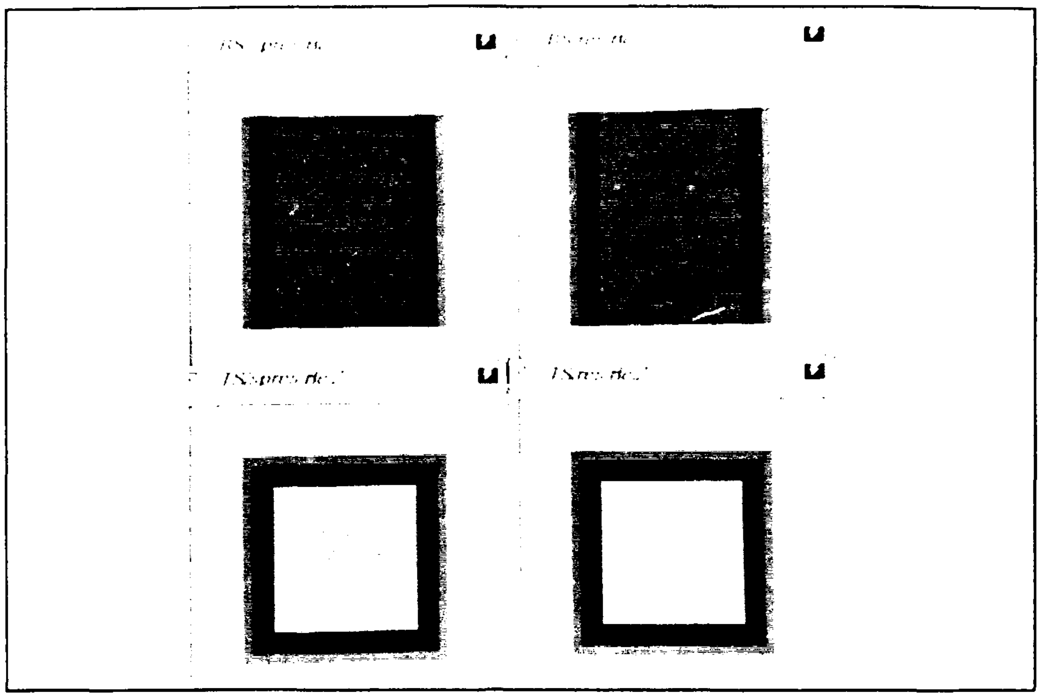

Figure 4.6 Artificial Box Volume Rendered by Single Processor (left) and Distributed Processors (right). Top and Side View.

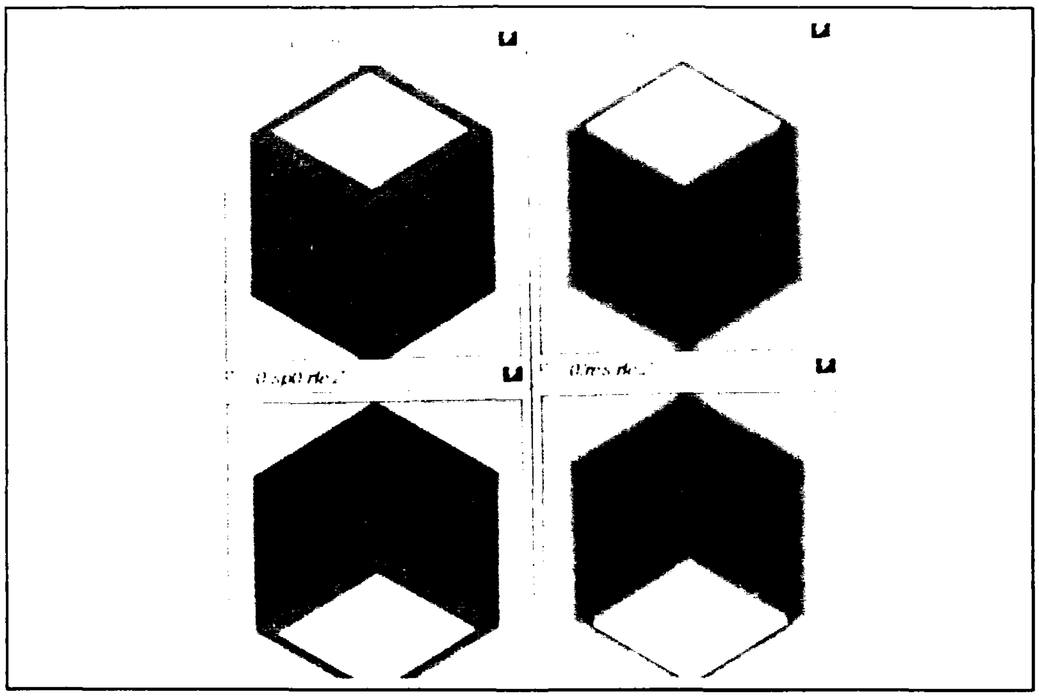

Figure 4.7 Artificial Box Volume Rendered by Single Processor (left) and Distributed Processors (right). Isometric Views.

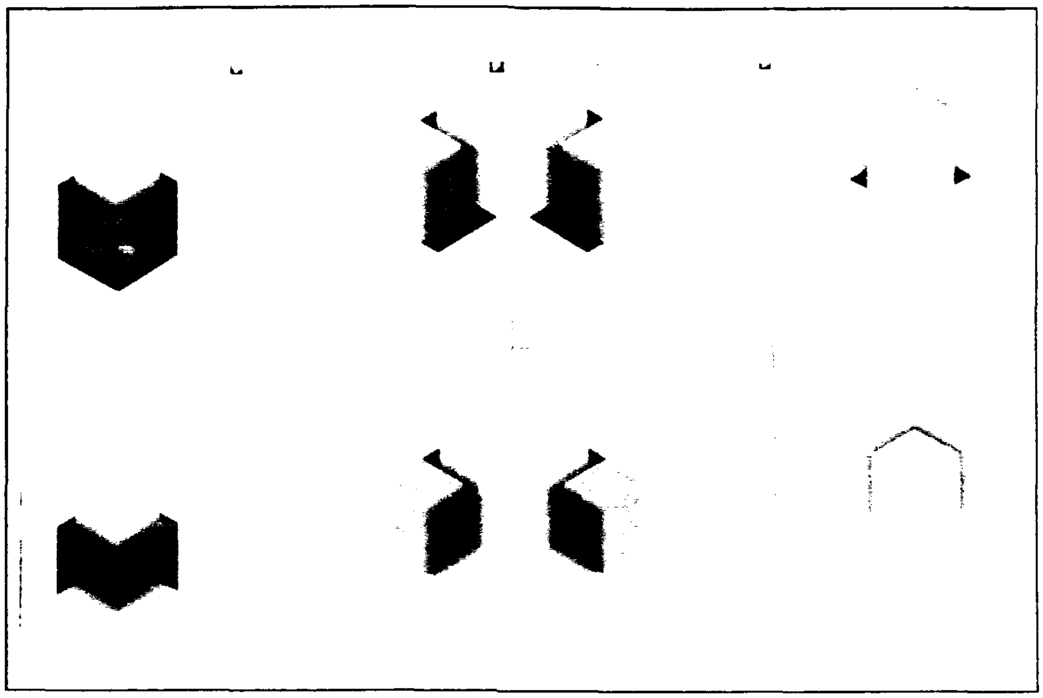

Figure 4.8 Artificial Box Distributed Volume Rendering of Octants.

straight-on views, the problem must then be an error in the code. I believe the error lies within the sampling of the ray at the octant borders.

During testing and debugging I dismissed several possible problems. I verified that the ray exits one octant and enters the next at the same point; so there are no breaks in the simulated single ray. However, this may relate to one source of the problem. In the implementation, the exit point of the ray is not the same as the last sampled point; because the exit point equals the exit point on the face of the volume and the last sample point is somewhere within the voxel containing that exit point. However, the entry point of a ray is always the first sample point of the ray into that volume. Recall that Levoy's algorithm calls for equally spaced samples along the ray. In the distributed implementation the last sampled point of the ray in one octant is not guaranteed to be an equal stepsize from the entry point (which is the first sample point) of the bordering octant.

I also dismissed the idea that the anomalies in the box volume were caused by the central differencing being used to approximate the normal vector at each voxel. This is done during the shade and classification step. I thought that at voxels at the volume boundary (specifically the shared voxels between octants) either forward or backward differencing is done, instead of central differencing, and this would cause inconsistent shading of the shared voxels between octants. However, this was disproved for the box volume because it is uniform and the shared voxels had equal colors assigned at this step.

I am confident that the final composition of all eight server buffers is correct. I believe it considers every case correctly with background and non-background colored pixels and composites as it should theoretically.

4.4 *Timing*

The timing is broken into three categories, preprocessing, ray casting, and a combination of the two (reflecting the total time to render). To equally compare ray casting times between single processor and distributed runs, the distributed ray casting times must include the compositing of the buffers done by the client. The time to write the screen buffers to files is considered part of the overhead for the distributed version, therefore is not considered when comparing times to preprocess, ray cast or render. Although, this overhead and other communication overhead is considered in the total execution times. Incidentally, the preprocessing, ray casting, and total times were taken from cpu clock times in microseconds, whereas the total execution times were computer wall clock times. The percent savings formula used in the following tables was [1-(Distr time / SP time)].

All test runs were executed on the same eight servers. Timings may vary depending on system load at the time of execution. Each set of executions (single and distributed) were accomplished by batch file runs of all views for CT chest and artificial box volumes.

4.4.1 *CT Chest Volume.* The timing differences between a single processor and distributed processor execution of the CT chest volume rendering is encapsulated in Table 4.1 (see page 4-10). The fourteen samples reflect fourteen runs from different views of the same CT chest data. The table shows a significant savings in every category between 70 and 85 percent.

To gain a better appreciation for the time saved by distributing the computation, Figures 4.9-4.12 (see pages 4-11 through 4-12) compare the timings in barchart style. All figures were generated from the columns and rows in Table4.1. Figures 4.9 (see page 4-11) shows the greatest savings of all categories. This is expected because preprocessing entails stepping through the entire volume. Since each server now handles one eighth of the volume, the time to preprocess is decreased. Figure 4.10 (see page 4-11) demonstrates the dependence of ray casting on the view taken. The 3D front view takes the longest amount of time while the right side view is the quickest. These deferences are reflected in the single processor times in Figure 4.11 (see page 4-12). Figure 4.12 (see page 4-12) demonstrates the average times of all views for all categories. This encapsulates overall comparisons into one figure.

Total program execution times averaged 10:05 (min:sec) for SP and 4:14 for distributed. This yields a 58 percent savings.

4.4.2 *Artificial Box Volume.*

The timing comparisons and savings for the box volume are encapsulated in Table 4.2 (see page 4-13). Savings of the same order as the CT volume are shown here as well. Although, no comparisons can really be made between the CT data runs and the artificial data runs since neither their construction nor sizes are close to each other. Again, more understanding can be gained pictorially, see the barcharts for the box volume in Figures 4.13-4.16 (see pages 4-14 through 4-15). Figure 4.13 again shows the greatest difference in times comes from the preprocessing step, as expected. Figure 4.14 (see page 4-14) also demonstrates how ray casting time depends on the view. In this volume the top side view takes the longest to ray cast. This is expected because the height of the volume is the greatest dimension. Notice, however, how the top side view takes the least amount of total time out of the distributed runs, see Figure 4.15 (see page 4-15). Figure 4.16 (see page 4-15) provides an overall average time for all categories.

Total program execution times averaged 6:32 for SP and 2:45 for distributed. This yields a 58 percent savings by distributed processing.

4.5 *Summary*

In this chapter I presented the results of the single processor VR and DVR. The results at the subvolume level was proven by the lack of inconsistencies between the images rendered on a single processor and those rendered on multiple processors (excluding the artificial data). The capability of intermediate volume viewing was shown here with the rendered images of the octants. Cut-away viewing was also shown here. Lastly, significant savings in rendering time, 75 percent, and total execution time, 58 percent, was realized by distributing the volume renderer.

The results of the box volume shown here indicates my program still has some error(s), but

Table 4.1 Table of CPU Clock Times (in Microseconds) for Single Processor and Distributed Implementations of Volume Renderer Shown by Steps and Total as Executed by View of CT Chest Data.

	Preprocessing SP	Preprocessing Distributed	% Savings	Ray Casting SP	Ray Casting + BuffComposite Distributed	% Savings	Preprocess + Ray Casting SP	Preprocess + RC + BuffComp Distributed	% Savings
0	234,390,624	38,813,031	83.44%	355,402,450	78,159,374	78.01%	589,793,074	153,393,864	73.99%
20	234,273,962	38,038,062	83.76%	355,585,776	78,182,289	78.01%	589,859,738	152,593,896	74.13%
40	234,307,294	38,283,885	83.66%	353,902,510	77,330,240	78.15%	588,209,804	152,216,828	74.12%
60	234,257,296	38,473,461	83.58%	350,219,324	76,986,504	78.02%	584,476,620	152,227,244	73.95%
80	234,257,296	38,806,781	83.43%	345,136,194	76,363,612	77.87%	579,393,490	151,487,690	73.85%
88	234,823,940	38,769,283	83.49%	353,735,850	75,617,809	78.62%	588,559,790	150,339,820	74.46%
96	234,240,630	38,163,057	83.71%	354,602,482	75,565,727	78.69%	588,843,112	107,531,115	81.74%
116	233,973,974	38,106,809	83.71%	357,385,704	76,542,772	78.58%	591,359,678	150,721,054	74.51%
136	233,907,310	38,217,221	83.66%	360,018,932	77,423,986	78.49%	593,926,242	152,158,497	74.38%
156	234,090,636	38,446,379	83.58%	362,785,488	78,542,692	78.35%	596,876,124	152,898,051	74.38%
176	234,257,296	38,269,303	83.66%	362,968,814	78,323,950	78.42%	597,226,110	152,212,661	74.51%
3DFront	233,840,646	37,948,482	83.77%	406,333,746	79,038,505	80.55%	640,174,392	154,064,671	75.93%
RTside	238,340,466	38,681,786	83.77%	318,453,928	80,347,186	74.77%	556,794,394	152,952,615	72.53%
TopSide	238,423,796	37,710,992	84.18%	343,636,254	71,684,633	79.14%	582,060,050	145,687,922	74.97%
Average	234,813,226	38,337,752	83.67%	355,726,247	77,150,663	78.31%	590,539,473	148,606,138	74.84%

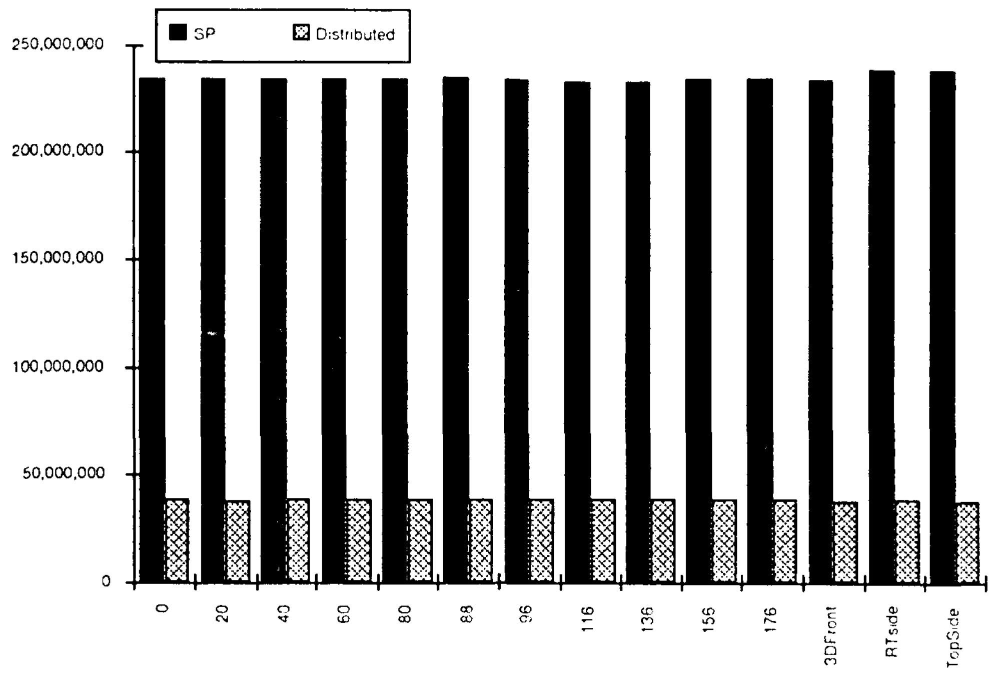

Figure 4.9 Comparison of CPU Clock Times (in Microseconds) for the Preprocessing Step of Volume Renderer Between Single Processor (Black) and Distributed (Hashed) Implementations on CT Chest Data.

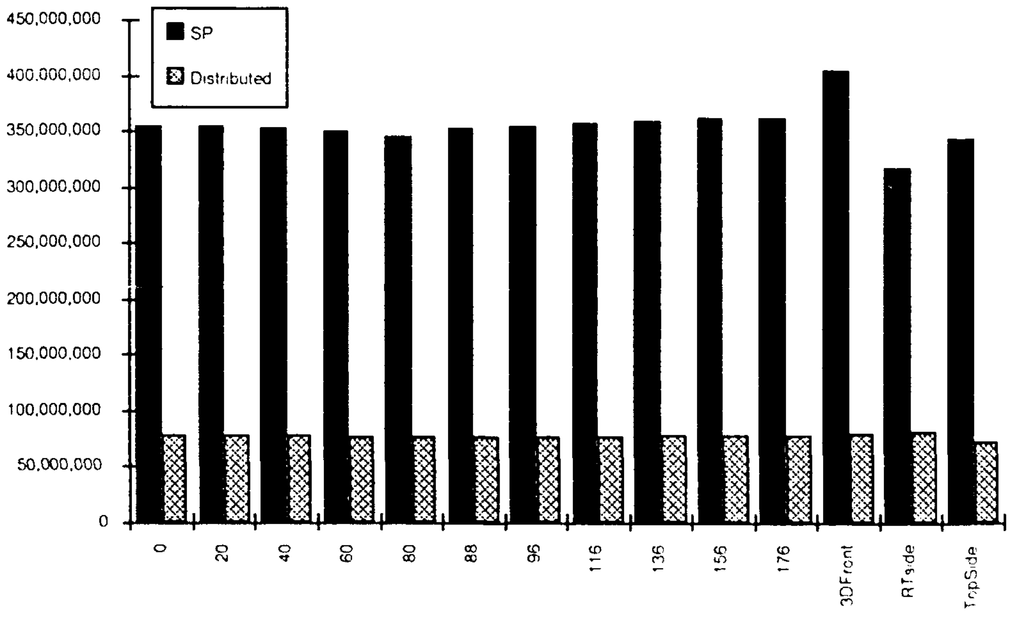

Figure 4.10 Comparison of CPU Clock Times (in Microseconds) for the Ray Casting Step of Volume Renderer Between Single Processor (Black) and Distributed (Hashed) Implementations on CT Chest Data.

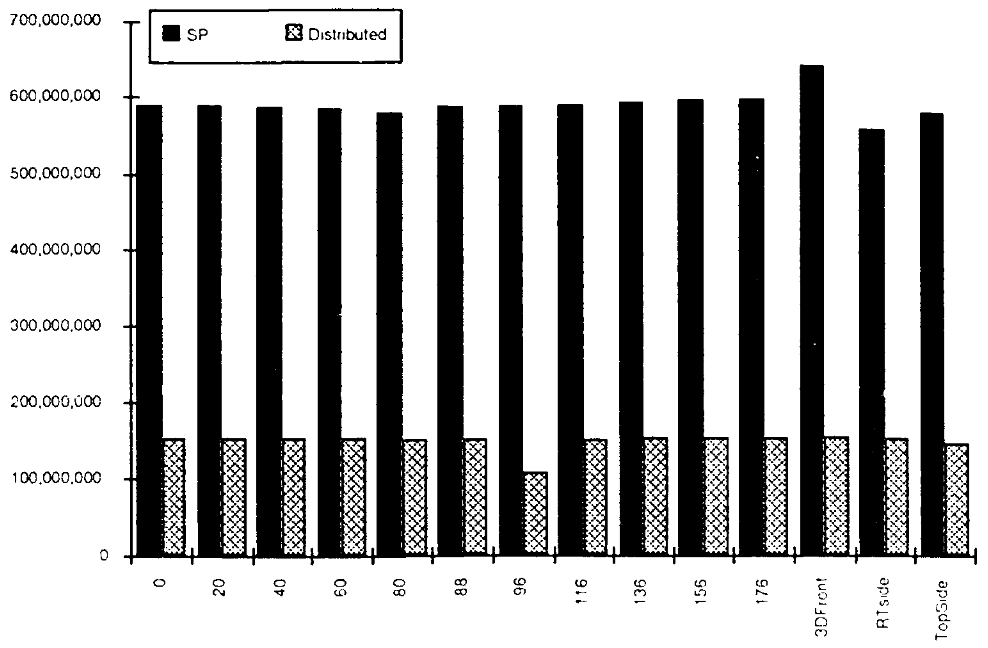

Figure 4.11 Comparison of CPU Clock Times (in Microseconds) for the Total of Preprocessing and Ray Casting Steps of Volume Renderer Between Single Processor (Black) and Distributed (Hashed) Implementations on CT Chest Data.

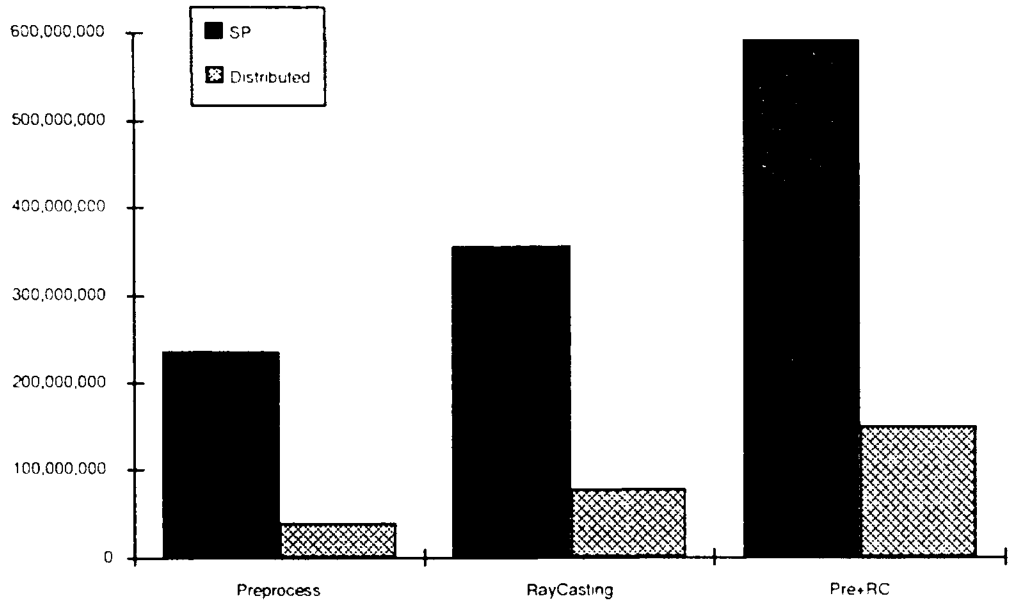

Figure 4.12 Comparison of Average CPU Clock Times (in Microseconds) for All Runs by Steps and Total on CT Chest Data.

Table 4.2 Table of CPU Clock Times (in Microseconds) for Single Processor and Distributed Implementations of Volume Renderer Shown by Steps and Total as Executed by View of Artificial Box Data.

	Preprocessing SP	Preprocessing Distributed	% Savings	Ray Casting SP	Ray Casting + BuffComposite Distributed	% Savings	Preprocess + Ray Casting SP	Preprocess + RC + BuffComp Distributed	% Savings
0	146,744,130	25,076,080	82.91%	223,124,408	53,156,207	76.18%	369,868,538	98,439,812	73.39%
20	147,460,768	25,076,080	82.99%	222,474,434	52,883,301	76.23%	369,935,202	97,825,254	73.56%
40	147,460,768	25,107,329	82.97%	219,791,208	52,524,982	76.10%	367,251,976	97,633,595	73.42%
60	147,427,436	25,217,741	82.89%	217,457,968	51,510,440	76.31%	364,885,404	96,769,046	73.48%
80	147,510,766	25,140,661	82.96%	216,024,692	50,725,054	76.52%	363,535,458	96,525,306	73.45%
100	147,444,102	25,288,572	82.85%	214,574,750	49,710,512	76.83%	362,018,852	94,600,383	73.87%
120	143,360,932	24,474,021	82.93%	225,190,992	50,593,810	77.53%	368,551,924	95,[illegible],948	74.21%
140	143,327,600	24,517,769	82.89%	228,040,878	51,622,935	77.36%	371,368,478	96,339,896	74.06%
160	143,344,266	24,586,517	82.85%	228,790,848	53,254,120	76.72%	372,135,114	97,087,783	73.91%
180	143,527,592	24,459,438	82.96%	232,574,030	53,097,876	77.17%	376,101,622	97,729,424	74.02%
200	143,810,914	24,534,435	82.94%	232,040,718	53,281,202	77.04%	375,851,632	98,008,580	73.92%
3DFront	143,327,600	24,640,681	82.81%	301,237,950	57,983,097	80.75%	444,565,550	101,768,846	77.11%
RTside	142,744,290	24,394,858	82.91%	280,755,436	54,733,227	80.51%	423,499,726	99,889,754	76.41%
TopSide	142,877,618	24,290,695	83.00%	325,086,996	63,793,282	80.38%	467,964,614	80,544,695	82.79%
Average	145,026,342	24,771,777	82.92%	240,511,808	53,490,717	77.76%	385,538,149	96,300,166	75.02%

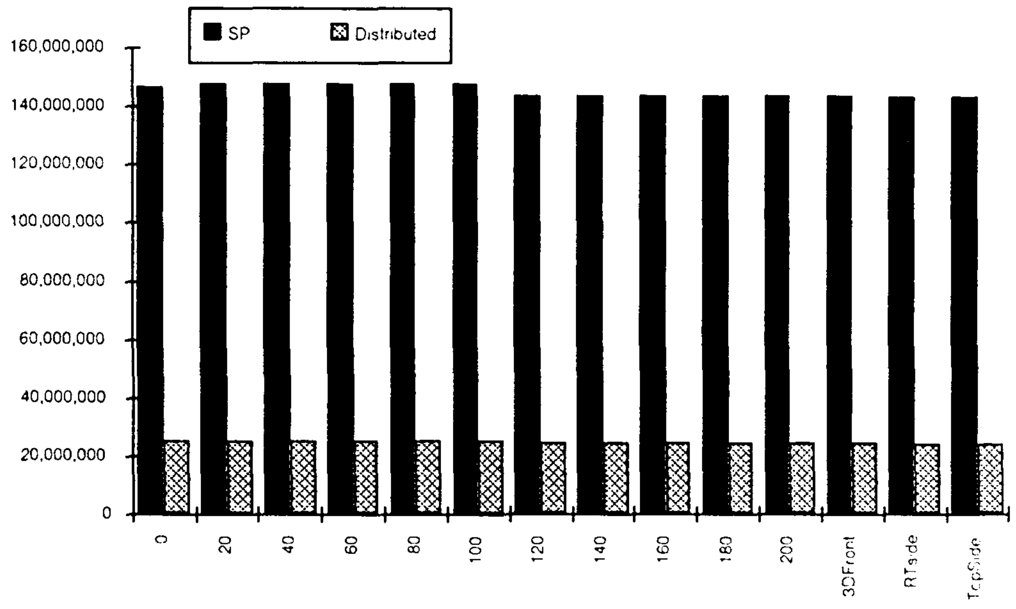

Figure 4.13 Comparison of CPU Clock Times (in Microseconds) for the Preprocessing Step of Volume Renderer Between Single Processor (Black) and Distributed (Hashed) Implementations on Artificial Box Data.

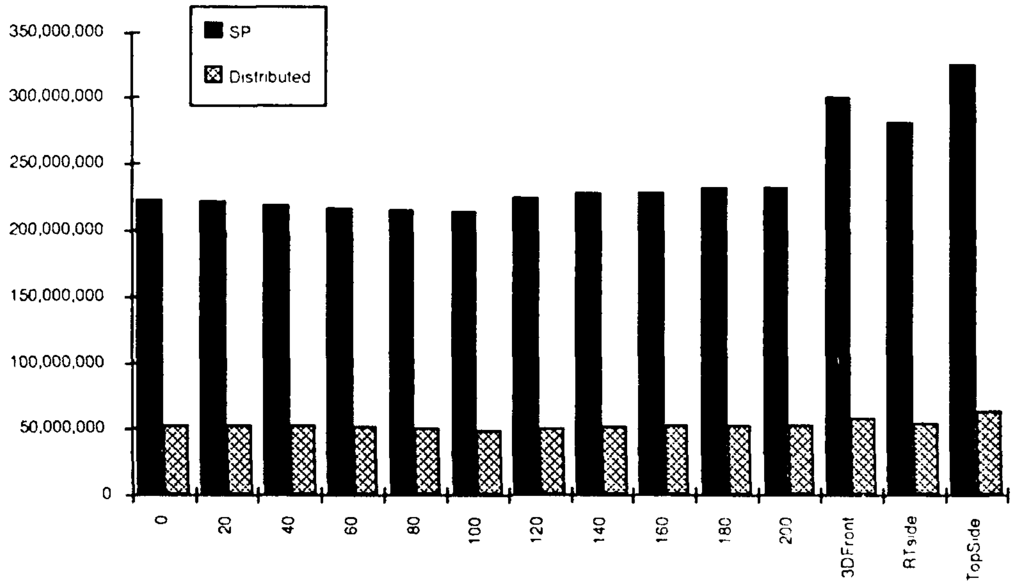

Figure 4.14 Comparison of CPU Clock Times (in Microseconds) for the Ray Casting Step of Volume Renderer Between Single Processor (Black) and Distributed (Hashed) Implementations on Artificial Box Data.

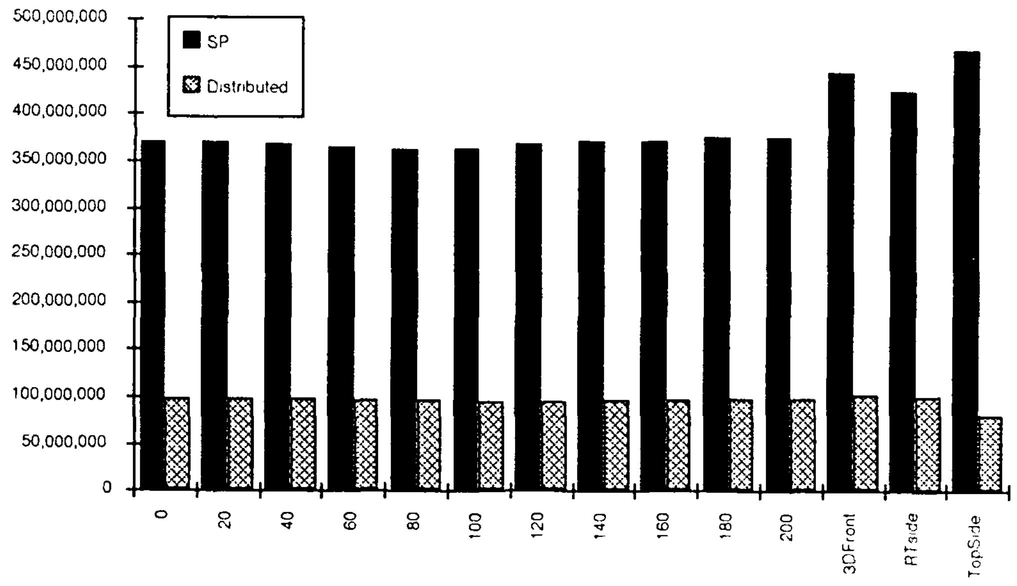

Figure 4.15 Comparison of CPU Clock Times (in Microseconds) for the Total of Preprocessing and Ray Casting Steps of Volume Renderer Between Single Processor (Black) and distributed (Hashed) Implementations on Artificial Box Data.

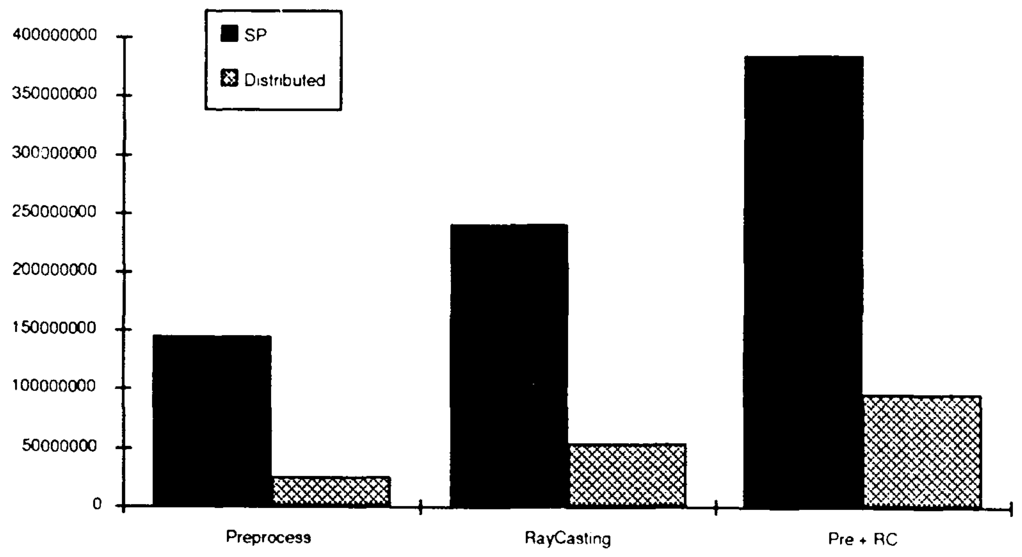

Figure 4.16 Comparison of Average CPU Clock Times (in Microseconds) for the Steps and Total on Artificial Box Data.

it does not disprove that the subvolumes can be rendered independently. I discussed where I thought the problem might be and what I concluded about the problem thus far. The next chapter will discuss overall conclusions and what I recommend for future work in this area.

V. Conclusions and Recommendations

The preceding chapters presented the development of a volume renderer using distributed processing. Results showed a 75 percent savings in rendering times. This chapter presents concluding remarks about my work and discusses future work in this area.

5.1 *Conclusions*

The distributed volume renderer implemented in this research effort meets the objectives set out at the start. The first objective met was successfully parallelizing the volume renderer at the subvolume level. Volumes were rendered by subdividing the volume, casting rays through each subvolume concurrently, using a front-to-back algorithm, and compositing the results into a final image. The medical images rendered in this way were of high quality and did not visually differ from single processor rendered images. However, as was discussed in Chapter IV, uniform artificial volumes were rendered with errors depending on the view. This indicates a problem remaining in the code. I surmised that the error probably exists within the sampling of the ray; particularly at octant borders.

A second objective met was intermediate viewing of the volume. This was accomplished by generating subvolume images and storing them as RLE files. These images provide the additional benefit of cut-away views of the volume. The limitation on these cut-away views is that they can only show cut planes along the octant divisions, which are midway along all axes in object space. Even still, these intermediate views can provide useful information to the clinician. If a specific cut-away view is desired, it can be setup with the appropriate location of the eyepoint and then rendered with DVR.

The other objective met was speedup of the volume rendering processing time. Substantial reductions in processing time were realized through distributed processing. A 75 percent speedup in total rendering time and a 58 percent speedup in total program execution times were the average savings. Although the distributed volume renderer did not achieve interactive timing, it still made great improvement over the single processor implementation. So, in that regard, it becomes more acceptable which means more usable to clinicians.

The success of this distributed volume renderer is significant for several reasons. First it proves that the ray casting method of volume rendering also can benefit from concurrent programming. Since ray casting's major disadvantage was its linear growth in rendering time with respect to the size of the data set, the other benefits of ray casting can now outweigh this cost. Second, the speedup achieved by this implementation of a distributed volume renderer approaches interactive timing—the end goal of volume visualization. Also, the distributed programming offers a parallel algorithmic solution that is not tied to specific hardware or architectures. In this way, a cost savings is realized because existing general-purpose workstations can be taken advantage of for their often idle computing power. In addition, the transparency of the distributed programming, except for the use of control files, makes the application more acceptable to the user. This combined with the speedup factor can stimulate the practical use of volume rendering in medical analysis of patients.

5.1.1 *Recommendations.* I make several recommendations for future work to improve the current state of the distributed volume renderer. Some of these recommendations stem from planned further iterations of my development, had time been available. Others stem from observations of the implementation by myself and others. I separate this future work into suggested improvements to the volume renderer itself and those to the distributed computing aspects.

5.1.2 *Volume Renderer.* First and foremost as a recommendation to improve the volume renderer side of my work, is to identify the error within my implementation in rendering uniform artificial data. I feel that the answer is within reach, but requires further concentration into the logic of the distributed implementation of the ray casting algorithm.

Furthermore, since the volume renderer implemented did not take advantage of existing speedup techniques within its design, these could be incorporated into the object oriented design of my work. Examples of these techniques are the use of octree data structures and dynamic adjustment of number of rays traced into subvolumes based on complexity of the subvolume (Levoy, 1988, 1989, 1990).

Additionally, adaptive subdivision and recursive subdivision of the volume can further enhance the speed of the process. Since my implementation always divides the volume into eight subvolumes, an ability to adaptively decide the initial division factor based on size of the input would be ideal. This would entail further algorithm changes to the Ray and Volume object, specifically when tracing a volume into octant boundaries. The implementation currently is tied to eight octants, the location of the pads associated with those eight octants, and other specifics to octant subdivision. Recursive and adaptive subdivision would go hand-in-hand with future work in the distributed aspect. Since recursive subdivision means the servers themselves would become clients, this could imply sophisticated distributed programming. Adaptive subdivision would require adaptive selection of servers.

Another modification to the volume renderer could allow for other shading techniques in the preprocessing step, such as Z-depth shading and gray level gradient shading.

5.1.3 *Distributed.*

Although my distributed programming of the DVR is useful, it has room for improvement. Several features could improve its usefulness, such as increased fault tolerance, recoverability, improved transparency, and optimization. Of course, many of these features are dependent on the operating systems and network computing environment available. But with the advent of transparent Distributed Computing Environments (DCE), many of these features will be incorporated within them and/or tools provided to allow the programmer to achieve such goals. Fault tolerance can prevent the DVR from failing due to a server that does not respond to a connection request. Recoverability can allow the DVR to continue in spite of a server's failure to render an image of its octant. An ideal recoverable design would provide the capability of the client to detect a problem and re-assign that *worker's* task to another server. Improved transparency would incorporate dynamic selection of servers into the DVR based on current load status.

Another desirable distributed feature is dynamic load balancing across the nodes. This is the ultimate goal to achieve peak performance. Dynamic load balancing would keep the program running at peak efficiency based on the local network computers available. The idea is to offload a server's process to another server with low usage when the initial server's load reaches some threshold. This requires that the client become more active during the server renderings of their volumes. During that time the client must monitor the active servers, as well as potential servers, frequently to asses load balancing information.

Looking toward long term goals for improvement, the advent of distributed application environments that allow transparency for both local area as well as wide area networks promises distribution across longer distances. This design could take advantage of general purpose computers with the highest MIP rates which are generally limited in quantity and possibly located remotely. Of course, the dependence on network speed becomes more of an issue in order to assess benefits gained by such wide area distribution, that is, unless network speed technology keeps pace with the software capabilities.

Further research should explore designing current DCE into compute-intensive applications, such as 3D medical imaging. I feel this offers the optimal solution toward interactive volume visualization.

Appendix DVR Operations Manual

DVR depends on several files for operation. One of these files provides the names of the servers on which processing is to be distributed. The format for this file is shown in Table A.1.

Table A.1 DVR Server Control File Format

Line	Keywords	Comments
1	servers <# of servers>	currently must use 8
2	<server 1 name>	machine names
	●	
	●	
	●	
	<servr n name>	

Another file is used to specify parameters, like those in GPR's control file. In fact, the format for this file was copied from GPR and modified slightly to reflect parallel orthographic viewing parameters and additional Phong shading variables. The complete format of this control file is shown in Table A.2 (see page A-2) with the changes from the GPR syntax shown in italics. The last control file required provides specific information about the volume size, location, etc., see Table A.3 (see page A-3). The syntax for all these tables is such that all words not delineated by special characters, must be duplicated in the file. Angle brackets represent variables and curly brackets mean a choice of options. Square brackets mean the term is optional.

The way to use these files on the command line when executing DVR is

```
dvr [-d] -s <server control file>
   -c <(gpr) control file>
   -v <volume control file>
   -o <rle output file>
```

The optional "d" is for debugging. The rle output file is the desired file name for the rendered image.

Before DVR can be run, several environment variables must be set. These environment variables set up directory paths so the user need not precede control file or output file names with directory paths on the command line. The environment variables should be named CNTRLDIR and OUTPUTDIR, and set to user specification.

Another environment variable is required until DVR is set up as a system-wide application. This environment variable, DISTRDIR, should reflect the current path to the DVR client and

Table A.2 DVR Modified GPR Control File Format

Line	Keywords	Comments
1		***Buffer Spec***
	buffer *ALPHASHADEBUFFER*	only 1 buffer type supported
	size <xsize><ysize>	integers
	background <r><g><b>	$0 \leq r,g,b \leq 1$
2		***Global Environment Parameters***
	viewport<xmin><xmax><ymin><ymax>	floats
	Ka <a>	$0 \leq r,g,b \leq 1$ ambient coefficient
	color <r><g><b>	$0 \leq r,g,b \leq 1$ ambient color
3		***Global Attribute Parameters***
	<{PHONG, FLAT, NONE}>	
	reflectance <{PHONG, FLAT, NONE}>	
	Kd <d>	$0 \leq d \leq 1$ diffuse coefficient
	Ks <s>	$0 \leq s \leq 1$ specular coefficient
	n <e>	float exponent
	dc1 <d>	$0 \leq d \leq 1$ depth cue 1
	dc2 <d>	$0 \leq d \leq 1$ depth cue 2
4		***Counts***
	num cameras <n>	integer camera count
	num lights <n>	integer light count
5+		***Camera Lines***
	position <x><y><z>	floats eyepoint
	coi <x><y><z>	floats center of interest
	twist <t>	float angle of twist
	near <n>	float near clipping plane
	far <f>	float far clipping plane
	left <l>	float left clipping plane
	right <r>	float right clipping plane
	top <t>	float top clipping plane
	bottom <b>	float bottom clipping plane
	fovx <x>	float field of view x
	fovy <y>	float field of view y
6+		***Light Lines***
	position <x><y><z>	floats
	direction <i><j><k>	floats generally=(eyepoint-coi)
	color <r><g><b>	$0 \leq r,g,b \leq 1$
	status <{ON, OFF}>	
	type <{INFINITE, POINT, SPOT}>	
	exponent <n>	float
	intensity <n>	float
	solidangle <n>	float

Table A.3 DVR Volume Control File Format

Line	Keywords	Comments
1	<{VOXELLAB, SUNRASTER, TEST}>	volume data type
2	<directory path to volume slice files>	no terminating "/"
3	<first slice number>	integer
4	<last slice number>	integer
5	<x size of volume>	integer
6	<y size of volume>	integer
7	<interslice distance>	float
8	<opacity region count>	integer
9+		region lines
	<d>	$0 \leq d \leq 255$ density
	<o>	$0 \leq o \leq 1$ opacity
	<w>	integer

server executables, namely **dvr** and **dvrsrvr**. This environment variable is critical for the client to start server processes correctly, since the client does so with an **rsh** call.

All servers generate a RLE file. By default, these RLE files will be located in the same directory as specified by the OUTPUTDIR environment variable. If the debug flag is on, each server generates a debug file which is also located in OUTPUTDIR.

Bibliography

Andrews, Gregory R. "Paradigms for Process Interaction in Distributed Programs," *ACM Computing Surveys 23 (1):*, 49-90 (March 1981).

apE. apE Version 2.0 Reference Manual. The Ohio Supercomputer Center. 1990.

Baum, Daniel R. and James M. Winget. "Parallel Graphics Applications," *Unix Review, 8(8)*: 50-61 (August 1990).

Booch, Grady. *Object Oriented Design With Applications.* Redwood City: Benjammin/Cummings Publishing Co, Inc. 1991.

Clay, Bruce A, Systems Engineer. AFIT/ENG (SRL). Personal Interviews. Wright-Patterson AFB, OH. September-October 1991.

Drebin, Robert A. et al. "Volume Rendering," *ACM Computer Graphics, 22(4)*: 65-74, (August 1988).

Deguchi, Hiroshi et al. "A Tree-Structured Parallel Processing System for Image Generation by Ray Tracing," *Systems and Computers in Japan, 17(12):* 51-61 (1986).

Dippe', Mark and John Swensen. "An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis," *ACM Computer Graphics, 18(3):* 149-158 (July 1984).

Fuchs, Henry et al. "Interactive Visualization of 3D Medical Data," *IEEE Computer, 22(8):* 46-51 (August 1989).

Fujimoto, Akira et al. "ARTS: Accelerated Ray-Tracing System," *IEEE CG&A, 6(4):* 16-26 (April 1986).

Glassner, Andrew S., editor. *An Introduction to Ray Tracing*. San Diego: Academic Press, 1990.

Kaufman, Arie. "Voxel-based Architectures for Three-Dimensional Graphics," *Proceedings of the International Federation of Information Processing, 10th World Computer Congress*. 361-366. Holland: Elsevier Science Publishers B. V., IFIP, 1986.

Levoy, Marc. "Display of Surfaces from Volume Data," *IEEE CG&A, 8(3):* 29-37 (May 1988).

__________. *Display of Surfaces from Volume Data.* Phd Dissertation, University of North Carolina, Chapel Hill, N.C., 1989.

__________. "Efficient Ray Tracing of Volume Data," *ACM Transactions on Graphics, 19(3):* 245-261 (July 1990).

Lorin Harold. Aspects of Distributed Computer Systems, New York: John Wiley & Sons, Inc., 1988.

Schlusselberg, Daniel s. et al. "Three-Dimensional Display of Medical Image Volumes," *Proceed of NCGA 1986*, vol.III 114-123. Anahiem, Calif. May, 1986.

Singhal, Mukesh and Thomas L. Cassavant. "Distributed Computing Systems," *IEEE Computer, 24(8):* 12-15, (August 1991).

Stytz, Martin R. *Three-Dimensional Medical Image Analysis Using Local Dynamic Algorithm Selection on a Multiple-Instruction, Multiple-Data Architecture*, PhD Dissertation, University of Michigan, 1989.

Sun Microsystems. Three Intense Days of Sun Distributed Computing Conference, April 1991. Conference notes.

Talton, D.A. et al. "Volume Rendering Algorithms for the Presentation of 3D Medical Data," *NCGA '87 Technical Session Proceedings*, 119-128, 1987.

Udupa, Jayaram K. and Gabor T. Herman, editors. *3D Imaging in Medicine*. Boca Raton: CRC Press, 1991.

Upson, Craig and Michael Keeler. "VBUFFER: Visible Volume Rendering," *ACM Computer Graphics, 22(4):* 59-64 (August 1988).

Wayner, Peter. "Distributed Applications With Vision," *Unix Review, 8(6)*: 58-62, (June 1990).

Westover, Lee. "Interactive Volume Rendering," *Proceedings Chapel Hill Workshop on Volume Visualization*, C. Upson, editor., Chapel Hill, NC, Dept. Computer Science, University of North Carolina, May 1989.

____________. "Footprint Evaluation for Volume Rendering," *ACM Computer Graphics, 24(4)*: 367-376 (August 1990).

Vita

Captain Patricia L. Brightbill was born on 20 November 1961 in Reading, Pennsylvania. She graduated there from Wilson High School in 1979 and immediately enlisted in the Air Force. Her first assignment as an airman was to the 81st Supply Squadron at RAF Bentwaters, England. In October 1981 she returned stateside to the 9th SRW at Beale AFB, Calif. As SSgt Brightbill, she was selected for the Airmen's Education and Commissioning Program (AECP). Under AECP she attended the University of California at Los Angeles (UCLA) to obtain her BS in Math/Computer Science. Following graduation from UCLA in 1986, Capt Brightbill was commissioned into the Air Force at Officer Training School. She then proceeded to gain technical training in communications and computers at Keesler AFB, Miss. Her first tour as an officer was to the 7th Communications Group at the Pentagon. She began as a programmer and communications officer in support of bringing the HQ USAF Local Area Network (LAN) on-line. Throughout her tour she assisted in the operations and management of the HQ USAF LAN, bringing about its expansion to service most of SAF, Air Staff, 7th Comm Group and other Air Force offices in the Pentagon and D.C. area. Within three years she became branch chief of the Network Operations Branch. In May 1990 she entered AFIT.

Permanent Address:
RD 6 Box 297
Sinking Spring, PA 19608