DEVICE INDEPENDENT PERSPECTIVE VOLUME RENDERING USING OCTREES

by

Timothy Lee Ryan

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science and Applications

APPROVED:

[illegible]

Clifford A. Shaffer, Chairman

[illegible]

Roger W. Ehrich

[illegible]

Deborah S. Hix

February, 1992

Blacksburg, Virginia

c.2

LD
5655
V855
1992
R924
C.2

DEVICE INDEPENDENT PERSPECTIVE VOLUME RENDERING USING OCTREES

by

Timothy Lee Ryan

Committee Chairman: Clifford A. Shaffer

Computer Science and Applications

(ABSTRACT)

Volume rendering, the direct display of data from 3D scalar fields, is an area of computer graphics still in its infancy. Only recently has graphics hardware advanced to a state where volume rendering became feasible. Volume rendering requires the analysis of large amounts of data, typically tens of megabytes. As hardware speeds increase, we can only expect the datasets to get larger. This thesis describes a reasonably fast, space efficient algorithm for volume rendering. The algorithm is device independent since it is written as an X Windows client. It makes no graphics calls to dedicated graphics hardware, but allows the X server to take advantage of such hardware when it exists. It can be run on any machine that supports X Windows, from an IBM-PC to a high-end graphics workstation. It produces a perspective projection of the volume, since perspective projections are generally easier to interpret than parallel projections.

The algorithm uses progressive refinement to give the user a quick view of the dataset and how it is oriented. If a different orientation or dataset is desired, the user may interrupt the rendering process. Once the desired dataset and position have been determined, the progressive refinement process continues and the image improves in quality until the greatest level of detail is displayed.

While this algorithm may not be as fast as algorithms written specifically for dedicated graphics hardware, its overall rendering time is acceptable. Hardware vendors who develop

X servers that take advantage of their graphics capabilities will only enhance the performance of our algorithm. The device independence this algorithm provides is a major benefit for people who work in an environment of mixed hardware platforms.

ACKNOWLEDGEMENTS

I would like to thank my wife Vyneta G. Ryan for her support (both monetary and emotional) for the last year and a half. I could not have done it without her. I would like to thank my parents for the examples they set for me to achieve this degree. I would like to express my appreciation to the professors of VPI for the knowledge that they have imparted. I would like to especially thank my advisor Cliff Shaffer for his advice and encouragement. Finally, I thank my friends who kept me sane through this entire thing, Loretta, Kim, Bill, Sarah, Dale, Carrie, and my best friend, Vyneta.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

1.1 Background

Volume rendering is defined by Foley et al [Fole90], as "the direct modeling of data represented as 3D scalar fields". Direct volume rendering creates an image by converting the data values of the volume directly into pixel contributions. This is generally a slow process, yet is amenable to several forms of optimization. To speed image generation times, some rendering algorithms produce a polygon mesh, or a collection of polygons that share common edges, as an intermediate step. The polygon mesh can then be displayed quickly by dedicated graphics hardware. Volume rendering can be used in many applications including the following.

- Visualization of fluid dynamics, such as airflow over a wing.
- Thermodynamic systems, such as heat transfer.
- Representing three dimensional volumetric functions.
- Medical imaging, such as Magnetic Resonance Images (MRI) or Computer Aided Tomography (CAT) scans.

Such a variety of uses has made volume rendering a rapidly growing area of computer graphics.

Three major approaches have been presented in the literature for volume rendering. The first is ray tracing, or ray casting [Levo90]. In ray tracing, a ray is cast from the viewer's eye position through each screen pixel and then through the volume. Points or regions that a ray passes through contribute to the color and opacity of the pixel corresponding to that ray. This approach can be optimized by performing calculations for a ray only until the ray reaches some opacity threshold (making it opaque) or the ray exits the volume. The second approach to volume rendering is to draw the voxels (three dimensional pixels) as clouds of points, or small gausian dots [Max90]. Areas of the volume that have more substance, receive more points or dots thus making them more opaque. Areas that are empty receive few or no points or dots and therefore appear transparent. The third approach to volume rendering is to transform the volume into the viewing coordinate system and project the data onto the screen [Wilh91]. A region when projected onto the screen contributes to all pixels that it covers. All regions are processed in turn, either from back to front or from front to back.

Combinations of these three approaches are common. For example, ray tracing can be used to render semi-transparent density clouds. In another example, surfaces can be projected and then rays cast only through the pixels covered by a surface.

Volume rendering algorithms can also be categorized by how they traverse the volume dataset: image order or object order. Image order traversal depends on the screen image. Data access is done in whatever order is required to produce a pixel at screen location x, y. Ray tracing algorithms are generally image order traversal algorithms. Object order traversal is directed by the data values in the dataset. Traversal is done so that contiguous regions of the dataset are accessed together, regardless of what screen locations they will map to. Projection algorithms are generally object order traversal algorithms.

Specialized graphics hardware to speed image generation times is becoming more common on graphics workstations. This new hardware is very often a parallel architecture. There are two different approaches to parallelization, image-parallel and object-parallel. These two approaches are analogous to image order and object order traversals. Image parallel architectures assign multiple processors to each pixel or group of pixels, in order to derive pixel values faster. Each processor accesses those sections of the data necessary to compute the resultant pixel value(s). Object parallel architectures divide the dataset into 3D regions and assign a processor to each region. This approach derives multiple pixels simultaneously. Each architecture has its advantages and disadvantages. Object parallel architectures divide the dataset among processors so no one processor needs the entire dataset in memory. Their disadvantage is that pixels may be computed by one processor that will be totally obscured by the pixels computed by another processor. Thus, computational power may be wasted. Image parallelism only computes pixels that will be visible to the viewer. However, each processor must have access to the entire dataset.

1.2 Problem Statement

Some of the problems associated with volume rendering are as follows.

1) The algorithms that are space efficient generally have slow image generation times.
2) The algorithms that have fast image generation times are generally space inefficient.
3) Most attempts to fix problem 1 entail using specific graphics hardware to draw high level graphics primatives such as Gouroud-shaded polygons.
4) Many algorithms produce a parallel projection instead of a perspective projection.

This thesis describes a reasonably fast, space efficient method of volume rendering. The algorithm produces a perspective projection of the volume on any hardware platform that runs X Windows. While not requiring any special graphics hardware, the algorithm, since it outputs polygons, can take advantage of it if the X server was written to do so.

1.3 Goal and Approach

The goal of this research is a fast method for volume rendering to enable real-time user interaction. The type of interaction we facilitate is interactive change of user view position. By interactive change we mean, allowing the user to move their view position in real time.

State of the art 3D input devices today are the space-ball and the data glove. Since these are not yet in common use, changes to the user's view position are accomplished via the mouse and on-screen controls. Since the user is stationary, object(s) represented by the data will appear to move relative to the viewer. In this way the user can make the object(s) rotate in any dimension, grow, or shrink by adjusting the view position appropriately. In order to support real-time manipulation of the volume, the data must be rendered very quickly.

To obtain the desired speed we exploit the technique of *progressive refinement* [Cohe88]. Progressive refinement begins by displaying a crude representation of the final image as quickly as possible. This representation is then improved upon over successive (typically slower) iterations, until an acceptable image has been rendered. This technique is used in many computer graphics applications, most notably in radiosity algorithms. Progressive refinement gives the user some indication of the final output early in the rendering process. We use progressive refinement to allow the user to intuitively identify their location in the image space by their orientation to the object(s) being rendered. Even a crude representation should be enough to determine whether the right dataset is being displayed, and whether the object(s) are right side up or upside down, for example. Once the user has reached the desired view position, the progressive refinement process continues and the image improves in quality until the greatest level of detail is displayed or the user moves to another view position.

The purpose of producing a space efficient method of volume rendering is to allow volume rendering of large data sets, possibly on computers with memory restrictions. Fast algorithms have been developed that require the entire dataset to be loaded into memory. This may be acceptable for small datasets. Large datasets do exist however, and are likely to become more prevalent as the use of volume rendering increases. The dataset used in this thesis is a Magnetic Resonance scan of a human head. Its dimensions are 256x256x109 at 16 bits per data value. This is equivalent to approximately 14Mb of data. 16 to 32Mb of RAM are commonly available for workstations at this time. Problems will arise, however, when datasets grow to 256x256x256 at 16 bits per data value. That is 32Mb just for the data. Our implementation only requires part of the dataset to be loaded into memory at any one time. To keep the rendering time as low as possible our method makes one sequential pass through the dataset when reading it from disk.

1.4 Data Structure

The octree, a three dimensional extension of the quadtree, is a hierarchical data structure based on recursive subdivision of space into eight equal subvolumes [Same90]. A complete octree is defined as follows. A three dimensional space is represented by a cube whose dimensions are $2^n \times 2^n \times 2^n$. The array is subdivided into eight partitions, each of dimension $2^{n-1} \times 2^{n-1} \times 2^{n-1}$. Each of these partitions is in turn subdivided. The partitioning continues as necessary until each octant meets some decomposition criteria. One way of representing this is by a tree of out degree eight. The root represents the entire object. Each of its eight children represent one eighth, or octant, of the object. Normally, this would require eight pointers for each non-leaf node to locate the children. Since our original dataset is at a voxel resolution we will produce a complete octree. The location of children can therefore be calculated as in a heap [Bent85]. This allows us to eliminate the pointers. This implementation is traditionally called the *pyramid* [Tani75]. Tanimoto and Pavlidis developed the idea of using several lower resolution levels of a 2D image to speed image processing time. Their levels were two dimensional matrices, each one quarter the size of the previous. All internal nodes of the pyramid contain some combination of the values of their children. This combination could be, for example, an average, the maximum value, or the minimum value. In our application we store a pyramid of 3D volumes. Parent nodes store the minimum and maximum of their child values. The reason for this choice will be explained in Chapter 3. Moving from the root down to the lowest level of the pyramid, we encounter successively higher resolution representations of the data. The lowest level of the pyramid is the actual data. The total storage requirement for the pyramid is 8/7 times that required for the base dataset if the internal nodes each require the same storage as one of the lowest level nodes. All levels of the pyramid are stored in octree order, described in Chapter 3, to facilitate access. This necessitates some preprocessing to convert from raster order to octree order, and then to construct the upper portions of the pyramid.

The remainder of this thesis is organized as follows. Chapter 2 describes previous work in the area of volume rendering, common problems encountered by volume rendering algorithms and the approaches others have taken to solve them. Chapter 3 describes our preprocessing and viewing algorithms. It lists problems we encountered and our solutions to those problems. Chapter 4 details the rendering routines used in the viewing algorithm. It also lists enhancements that were added to speed volume rendering time. Chapter 5 provides our results and analysis for different hardware platforms. Chapter 6 is the conclusion and includes thoughts about future work in the area of volume rendering.

Chapter 2

PREVIOUS WORK

Volume rendering is a young area of computer graphics. Most of the previous volume rendering work has been done since 1987. Some of the first papers on volume rendering were presented at SIGGRAPH 88. Paolo Sabella presented a paper describing a ray tracing algorithm entitled "A Rendering Algorithm for Visualizing 3D Scalar Fields" [Sabe88]. Craig Upson and Michael Keeler preseneted a paper entitled "V-BUFFER: Visible Volume Rendering" in which he compares ray casting to a simple projection method [Upso88]. Robert Drebin, Loren Carpenter, and Pat Hanrahan also presented a paper at SIGGRAPH 88. Its title was simply "Volume Rendering" [Dreb88]. It is discussed in greater detail in a following paragraph. Many problems are still being discovered in volume rendering. Some of the problems that others have addressed are image quality, image generation time, perspective versus parallel projections, rendering volumes not based on a rectilinear grid, and producing isosurface generations from volume datasets. These problems have been addressed by the volume rendering community in a variety of ways, some of which are presented in this chapter.

Donald Meagher was a pioneer in the applications of octrees. In 1982 he developed a geometric modeling scheme called *Octree Encoding* [Meag82]. Octree encoding allows for the representation of objects of arbitrary complexity (within memory limits) at a specified precision, or resolution. In octree encoding the data is stored in a hierarchical tree structure. The leaf nodes in an octree encoding represent data that are either entirely inside an object or entirely outside an object. Meagher's octree encoding supports the boolean operations of union, intersection, and difference, as well as the geometric transformations of translation, rotation, and scaling. Meagher's algorithm provides an orthographic projection, but describes a deformation capability to give the appearance of a perspective projection, though this was not implemented at the time this paper was written. Octree encoding provides display of the objects with hidden surfaces removed by the simple process of visiting the children nodes in the proper sequence. However, Meagher's algorithm did not allow arbitrary viewing positions. Viewing was only allowed from one of the eight corners of the volume at an infinite distance. Aref and Samet [Aref91] continue Meagher's work by discussing the problems involved in perspective projections from an arbitrary view position. In particular, they address perspective viewing of objects represented by octrees. They present an algorithm that correctly displays data stored in an octree in time linearly proportional to the total surface area of the objects being viewed (since it processes each octree node once). By correctly displays, it is meant that using their algorithm will generate no false artifacts such as can be introduced by other perspective projection approaches. Based in view position, the algorithm determines, at each level of the octree, what order to process the children octants. They implement this ordering by processing the octree in a depth-first, recursive, fashion. While depth first processing will generate high quality final images, progressive refinement requires a breadth first order of processing.

Drebin et al [Dreb88] address the problem of image quality. Their algorithm consists of several stages. Each stage accepts an input volume and produces an output volume. Each input volume is interpreted as a sampled continuous signal. To avoid aliasing problems, this interpretation requires the original dataset to be sampled above the Nyquist frequency, or that the original continuous signal is low-pass filtered. The original volume is converted to a set of material percentage volumes. The material percentage volumes describe the percentage of a particular material, like bone, present in a region. Additionally, a composite color volume, opacity volume, density volume, surface strength volume, surface normal volume, shaded color volume, and transformed volume are produced. One of the problems with their algorithm is that it requires a different probabilistic classifier for each type of dataset to be viewed. The probabilistic classifier estimates the percentages of each material found in a given region. The classifier for a CAT scan dataset, for example, would be based on x-ray radiation absorption. The algorithm applys rotations and a perspective transform to the volume and projects it onto the viewing plane in a back to front traversal. All of this is performed on a PIXAR Image Computer. The resultant images are of very high quality, but require large amounts of computation for the various stages. The specialized nature of the PIXAR Image Computer may make this algorithm feasible on that platform, and not feasible on more common workstations.

Marc Levoy has been working with volume rendering since at least 1988. In a paper entitled "Efficient Ray Tracing of Volume Data" [Levo90], he addressed the problem of long image generation times. He presented a front-to-back image-order volume rendering algorithm using a pyramid data structure. Levoy assigned a color and opacity to each voxel and a 2 dimensional orthographic projection of the resulting semitransparent volume was created by a ray tracing algorithm. He discusses two strategies for improving the performance of a standard ray tracing algorithm for volume data. The first makes use of coherent regions of empty voxels. By employing the pyramid data structure, it is possible to determine the largest region defined by a pyramid node that contains all empty voxels. It is then possible to stop the ray calculations upon entering such a region, calculate the exit point of that ray from the region, and restart the ray calculations from that point. The second improvement strategy relies on the accumulated opacity of a ray. Once a ray has reached a specified opacity threshold, voxels farther along the ray do not contribute to the color of the resultant pixel. It is therefore possible to terminate ray calculations before the ray has passed through all voxels in the dataset. Levoy's algorithm gains time efficiency at the expense of space efficiency. He requires that the entire pyramid be stored in memory due to the random data access of his algorithm. For large datasets this may not be possible.

Novins et al [Novi90] address the problem of parallel projection rendering algorithms by presenting an efficient ray tracing approach that provides perspective projection. Their algorithm, unlike Levoy's, does not require that the entire dataset be resident in memory. This also addresses the problem of space efficiency. They divide the dataset into slabs. A slab is two adjacent x-y planes of data. They determine a viewing order before processing starts. Slabs are read in one at a time and all rays which pass through the current slab are processed. Once a ray has been processed, the next slab it will encounter is calculated. The ray is queued to be processed again when that slab becomes the current slab. This approach would tend to give the algorithm a progressive refinement look since many rays are completed partially as the data is read.

Wilhelms and Van Gelder [Wilh91] also address the problem of image generation time, but use an object-space projection algorithm instead of ray tracing. Projection differs from ray tracing by processing the volume region by region instead of ray by ray. They use a coherent projection approach to direct volume rendering. The coherence of the sample data, or degree to which large areas have equal or nearly equal data values, can be used by projection algorithms to display large flat surfaces faster than general ray tracing algorithms. Their paper discusses a projection approach for directly rendering rectilinear, parallel-projected sample volumes. It also looks at the issues involved in integration and interpolation of the data. They discuss the advantages and disadvantages of using hardware Gouraud-shading for volume rendering. The problems with their approach are the parallel projections and the dependence on hardware Gouraud-shading. Parallel projections are faster to compute, but less realistic in appearance than perspective projections. The speed of their algorithm seems to come from the fact that they can recognize large coherent surfaces and let the hardware draw it as a Gouraud-shaded polygon. This implies that their algorithm may not be suited to machines which do not have such hardware graphics primitives.

Max et al [Max90] work with density clouds (volumes represented as clouds of points). Their goal was to render surfaces and density clouds in the same image. They first scan convert and composite the density cloud into a convex polyhedron. Compositing consists of adding the colors and opacities of voxels together. They then slice the convex polyhedron into other convex polyhedra by the planes that define a surface. These are depth sorted and passed to a volume compositor. They take advantage of area coherence by producing polygons. If a polygon projects to a large area of the screen, then this can be quite efficient. For data where each polygon projects to only a few pixels however, the overhead in generating the polygon outweighs the savings. Another problem is the fact that they have to sort the polyhedra. If a cycle exists, such that for three polyhedra A, B, and C, A obscures B, B obscures C, and C obscures A, then one of the polyhedra must be subdivided. They had not developed a method for doing this at the time their paper was published. They hoped that such cycles would not occur in real datasets.

Wilhelms et al [Wilh90a] describe direct volume rendering of curvilinear volumes. They address the problems of volume rendering a dataset that is not sampled in a rectilinear grid. In particular, they explore the use of direct volume rendering for data from computational fluid dynamics. The sample points for such a dataset lie on a *curvilinear grid.* A curvilinear grid is defined to consist of cells, each of which have eight vertices, but whose faces are not necessarily planar. In addition, the grid points do not lie on orthogonal axes. Their conclusions are that reinterpolation to a rectilinear volume provides a faster method of rendering curvilinear volumes. This may introduce errors into the resultant image, however. Their results indicate that direct volume rendering of volumes that are not in a rectilinear grid is not a time efficient process.

Wilhelms and Van Gelder [Wilh90b] address the problem of generating isosurfaces from volume data and image generation time. Isosurface generation is the generation of a surface of uniform value. The dataset is scanned for a particular data value. The locations where those values are found are used to form a surface, which is projected onto the screen. Isosurface generation is useful for displaying elements of the volume that are in common. On an MR scan for example, if the value corresponding to soft tissue is known, an image can be displayed that shows the surface of the soft tissue. On the same dataset, a value corresponding to bone can be used to produce an image of just the skeleton. Wilhelms and Van Gelder introduce a space-efficient design for octree representations that are not of dimensions that are a power of two: branch-on-need octrees (BONOs). BONOs are basically octrees that are created from the leaves to the root, instead of the other way around. Eight children are combined to form one parent. This produces the greatest branching at the lowest levels, and hence the smallest tree.

Laur and Hanrahan [Laur91] address the problem of image generation time by using a progressive refinement technique they call *hierarchical splatting* to perform volume rendering. They use a pyramidal volume representation where the upper levels of the pyramid store average data values and an estimated error. Splats are footprints scaled to match the size of a projected cell. A footprint is a two dimensional projection of a three dimensional function, such as a Gaussian. A splat will typically be brighter in the center, indicating more volume of the 3D object projecting to these pixels than at its edges. The splats are approximated by RGBA (color and opacity) Gouraud-shaded polygons. The splats are then composited on top of each other in back to front order. They have developed a real-time rendering algorithm with a few limitations. According to their conclusions, even at full resolution, the resulting image does not equal other high-quality rendering techniques in picture quality. Thus the approximations that were introduced to attain the interactive speed sacrifice the quality of the final image. They also appear to rely on hardware Gouraud-shading of polygons to achieve their speed.

Previous work in the area of volume rendering address the problems of image generation times, or space efficiency, or perspective viewing of the dataset. This thesis attempts to address all of these problems as well as provide device independence. Our work attempts to give the user more information quicker, by using progressive refinement similar to hierarchical splatting. It provides space efficiency that is scalable to the memory available on the host machine, like the slab approach. It provides perspective projections of the volume, like Aref and Samet's approach. In addition, it provides device independence by being written as an X Windows client.

Chapter 3

GENERAL DESCRIPTION OF THE ALGORITHM

This chapter provides a general description of the preprocessing and viewing algorithms. There are two steps to preprocessing. The data must first be transformed to octree order and then the pyramid must be built from this octree ordered dataset. After these preprocessing steps have been done the dataset can be viewed. The preprocessing steps only need to be done once for each dataset. They produce a view independent pyramid data structure.

3.1 Preprocessing

Most 3-D volume datasets that exist today are arranged as a three dimensional raster array. The first n elements represent the first row, m rows of n elements each represent the first xy -plane, and p xy -planes make up the volume. The first preprocessing step is to change this order. Since we want to access this data in octree order, the most efficient way to have it stored on disk is in octree order. This allows sequential disk access by octants. In other words, octants are stored sequentially on disk, so reading one octant requires one sequential disk access. If an octree is described as a tree of degree eight, then octree order is the order in which the voxels would be accessed during a traversal of the octree. Figure 3.1

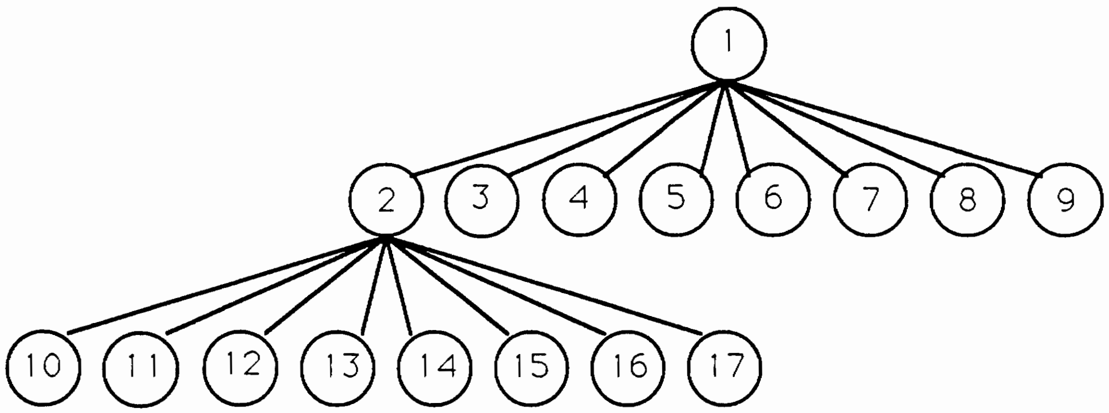

Figure 3.1: Octree represented as a tree of degree eight

shows a tree of degree eight with the nodes labeled. Figure 3.2 shows the same octree as a cubic volume with the nodes labeled the same as Figure 3.1. A complete octree requires that each dimension of the dataset be equal, and a power of two. It may be necessary to add empty data elements to the dataset in order to achieve this. Alternatively, there are ways to implement the rendering algorithm to deal with non-complete octrees. Such an implementation will be discussed later. A recursive routine was created to read the data elements from a raster ordered dataset and write them as an octree ordered dataset. This routine calculates the x, y, z coordinate for the next voxel in octree order. It then reads the element at position (x, y, z) in the original dataset and writes that element to a new dataset.

The second preprocessing step builds the pyramid. The units of subdivision within our pyramid are levels and nodes. The octree ordered dataset obtained from step one becomes the lowest level, level 0, of our pyramid. Each element of the dataset is a node in level 0. The total number of nodes in level 0 is then the total number of elements in the dataset. Each node of level 1 in our pyramid is called a *parent* and is constructed by combining the values

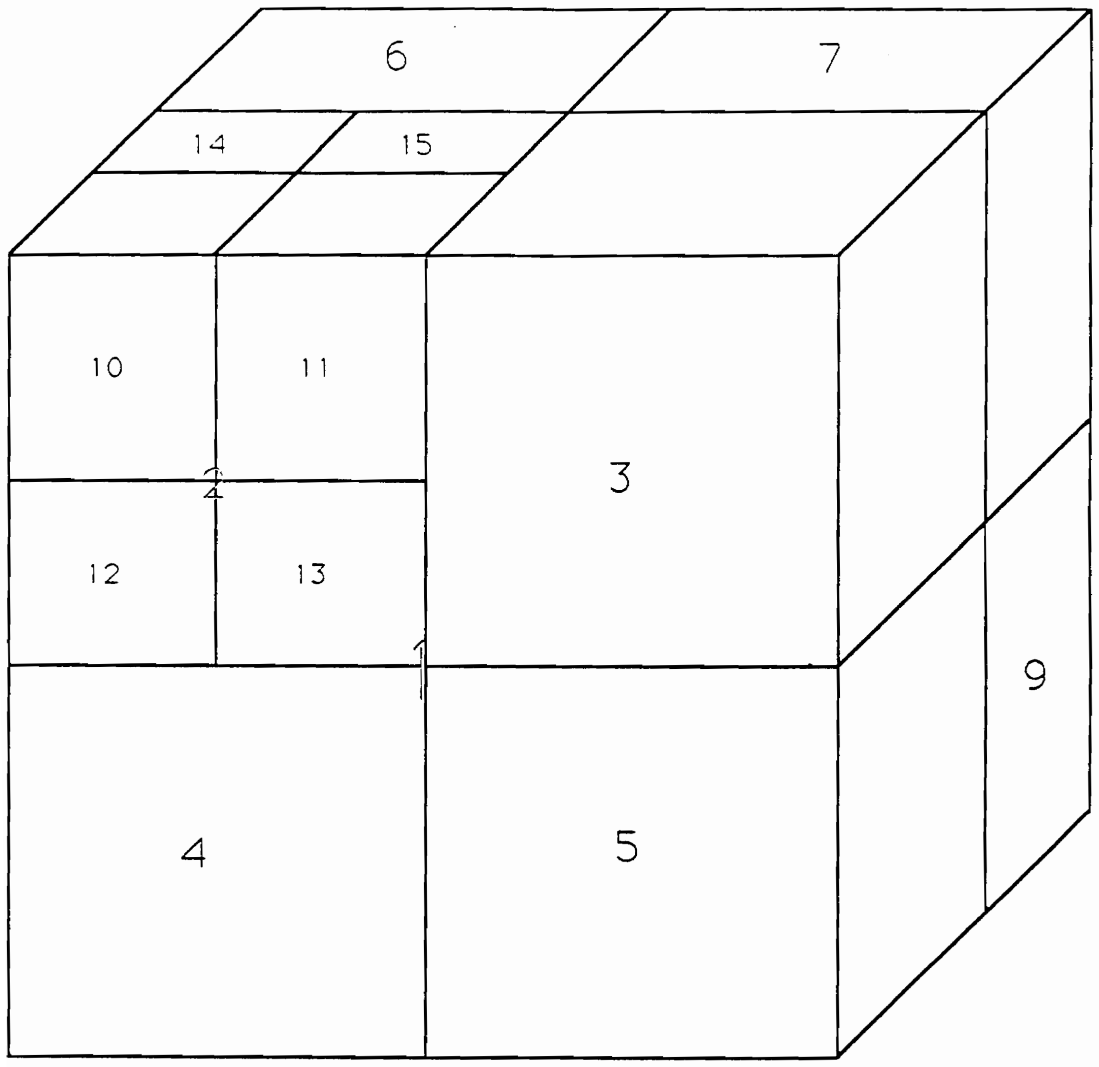

Figure 3.2: Spatial representation of the octree

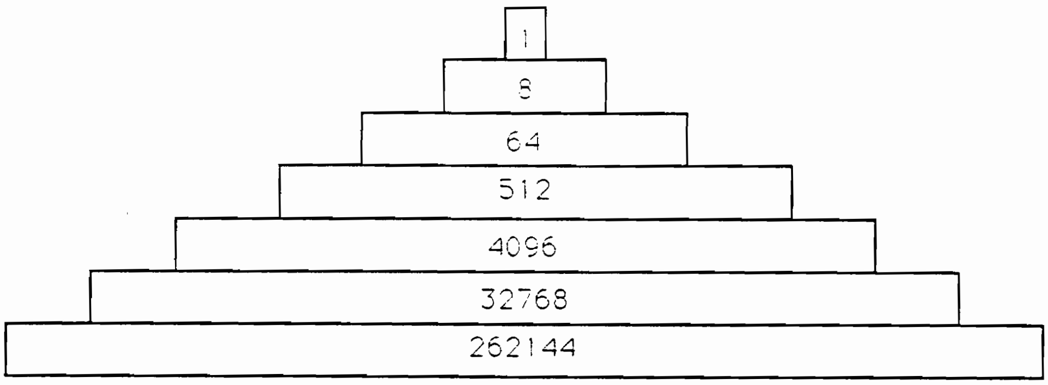

Figure 3.3: Conceptual picture of the pyramid data structure showing number of nodes at each level for a dataset of resolution 64 × 64 × 64

of the eight nodes from level 0 that would be its *children* in a pointer based representation of the octree. We retain the use of the terms *children* and *parent* even though we do not store explicit links between them. Since the lowest level is already in octree order, the first eight children are the first eight nodes of that level. The procedure continues combining eight children of level *i* and forming one parent on level *i+1*, until all nodes on level *i* have been processed. When it has completed, level *i+1* contains one eighth the number of nodes at level *i*. These nodes are also in octree order. The procedure then repeats itself, reading level *i+1* as children and writing level *i+2* as parents. This continues until we reach the root of the pyramid, where the number of nodes is one. The final result is a complete pyramid with each level in octree order. An illustration of a pyramid and the number of nodes that are stored at each level is given in Figure 3.3.

The combination of data values that are stored in the upper levels of the pyramid affect the performance and possible applications of the algorithm. We chose to store a quantized minimum and maximum value of the eight children. The original dataset contained sixteen bit integers, of which the lowest twelve bits are significant. We quantized, or reduced the resolution, to eight bits for the upper levels in order to pack both the maximum and minimum into one sixteen bit integer. The lowest level still contains one sixteen bit integer for each data value. We have found that storing the minimum and maximum gives the most flexibility to our algorithm. The minimum value allows us to safely draw the upper levels to the screen without having to erase parts of the picture later. The maximum value allows us to terminate processing parts of the octree that contain only empty space.

Handeling non-complete octrees is accomplished as follows. The octree is built in a manner similar to the BONOs described by Wilhelms and Van Gelder [Wilh90b]. The octree is constructed from the bottom up instead of from the top down. If eight children are not available to be combined into one parent, by the preprocessing, then the necessary number of nodes are created. These created nodes contain a special "empty" value that the algorithm will handle by not drawing and not breaking down into its children. If a dataset is of dimensions 256 × 256 × 128, for example, then building up from the bottom results in levels ranging in size from 8,388,608 nodes on the bottom level to 4 nodes at the top level. To these are added 4 more "empty" nodes to fill out that level to 8. Finally, 1 more node is added as the parent of those 8. This adds only 4 extra nodes to the pyramid.

3.2 Viewing Algorithm

We construct the pyramid to support progressive refinement. Progressive refinement allows the user to interact with the dataset in real-time. The time required to completely render the dataset using progressive refinement is typically greater than that required by using direct volume rendering. The advantage comes from being able to change the viewer's position with respect to the object(s) in the image after partial rendering. By quickly giving the user a rough approximation of the objects being rendered, the user can determine the orientation of the image and change positions if necessary. Ray tracing and direct volume rendering without progressive refinement give the user no indication of the entire image until rendering is complete.

The collection of pyramid levels above the lowest level, or voxel level, will be referred to as the internal levels. The internal levels of the pyramid together take up approximately one seventh the amount of space required by the lowest level. For datasets common today, one seventh is sufficiently small that it can be stored in RAM. The internal levels can be read in once at the beginning of the viewing process. This allows us to repeatedly display the internal levels and change view positions as many times as we wish without additional disk access. The lowest level of the pyramid need not be read until its data is needed. The entire lowest level need not be in memory at once, unlike the ray tracing algorithms of Levoy, because we know in advance the order in which the nodes will be accessed. The amount of lowest level data that is read into memory can easily be changed to accommodate the memory availability of the host computer. For our experiments, we chose to read one octant of the lower data into memory at a time. The internal levels of the pyramid and one eighth of the lowest level then occupy just over one quarter of the memory required by the entire volume. With our dataset this amount is approximately 6MB.

Our viewing algorithm starts with the root of the pyramid. The root level contains only one node. It then traverses the pyramid in breadth first order. In other words, all of the nodes on one level are processed before moving down to the next level. Our viewing algorithm is additive, i.e. it does not draw anything that will have to be erased later. This is accomplished by storing the minimum and maximum values as described in the preprocessing section. Nodes further down the pyramid can only expand, never decrease the image.

The dataset is a three dimensional volume. There are two traditional ways to interpret the data in volume rendering. The data could represent totally opaque material. In this case the easiest processing order would be back to front (often referred to as the painter's algorithm). Objects which obscure other parts of the image would be drawn later. Thus no hidden surface removal is necessary. The other way to interpret the volume data is as a semi-transparent solid. This allows us to see, at least part way, through an object. In this case, each voxel contains a color and an opacity. This opacity is a value between zero and one and represents the degree to which it obscures voxels that lie behind it. The processing order for semi-transparent solids can be back to front, or front to back. There are advantages to each processing order. We have chosen front to back since that will allow us to stop processing for a screen pixel that is already opaque. When dealing with semi-transparent solids, a compositing function is needed to add colors and opacities together. Our algorithm allows for a variety of compositing functions to be inserted. The particular function we implement is discussed in Chapter 4.

Our viewing algorithm approach can be viewed as the inverse of ray tracing. Ray tracing casts a ray through one pixel of the screen and follows that ray through the dataset. Color and opacity are accumulated for that pixel until its final value is determined. Our algorithm traverses the dataset and accumulates partial values for many pixels. The final color of any pixel is not known until the last data value is processed. In ray tracing, a ray can access any data value in the dataset depending on the view position. Any time the view position is changed, the order in which the data is accessed is changed. Thus, ray tracing algorithms require random access to the pyramid. For this reason, ray tracing algorithms are most efficient when the entire dataset is stored in memory. Our algorithm allows sequential access irrespective of the view position. According to timings we have performed, sequential disk access accounts for approximately 3% of our total processing time, when the volume is completely rendered. More often, the user will stop the rendering process before the bottom level must be read in.

Chapter 4

DESCRIPTION OF RENDERING ROUTINES

The steps we have devised to render the volume are as follows.

1) An octant from the pyramid is identified as the next to be rendered.
2) If its minimum value is greater than the "empty" intensity threshold, then the eight corners of the octant are transformed and projected to the screen by means of a perspective projection.
3) Faces are constructed from the projected corners. Either one, two, or three faces of the octant are visible from the view position.
4) The visible faces are filled as polygons using the minimum value.
5) The rectangular region containing the visible polygons is composited with the corresponding pixels in the image.

This chapter describes four sets of routines used in the volume rendering process. The first allows the user to change their view position. The second set transforms the x,y,z coordinate system of the volume to the coordinate system defined by the user's view position. The third set of routines projects the data represented by our pyramid onto the screen in this new coordinate system. The last set of routines is the compositing process.

4.1 View Positioning Routines

The user interface for the system contains six view positioning buttons. They are labeled “Spin +”, “Spin −”, “Direction +”, “Direction −”, “Move In”, and “Move Out”. An indicator on the screen shows in which direction the object will rotate. The screen is illustrated in Figure 4.1. Clicking the “Spin +” button appears to rotate the volume in the direction indicated on the screen. Similarly, clicking “Spin −” will appear to rotate the volume in a direction opposite to what is indicated on the screen. The buttons “Direction +” and “Direction −” change the direction in which the volume appears to rotate, as well as the direction of the indicator. The use of the direction buttons will be described further below. The “Move In” and “Move Out” buttons make the image grow and shrink respectively. We have used the term “appears to rotate”, because in fact, it is the viewer's position that changes, not the volume. The x,y,z coordinates of the data always remain constant.

Figure 4.2 illustrates the actions of the buttons. When a “Spin” button is clicked, the view position is translated along a great circle of a sphere that surrounds the volume, illustrated by the ring in the figure. The “Direction” buttons change the great circle by rotating the axis labeled “Direction” in Figure 4.2. The view position is defined by two angles, ϕ and θ. θ is the angle of rotation of the “Direction” axis. ϕ determines where the viewer is on the ring. The “Spin” and “Direction” routines need only increase or decrease ϕ and θ to change the view position. Each click of the button changes ϕ or θ by 0.1743 radians, or approximately ten degrees. The “Move In” and “Move Out” buttons change the radius of the great circle.

The view position is then converted to Cartesian coordinates so that the dataset can be projected to two dimensional screen coordinates correctly. The center of the volume and r, the radius from the center to the view position, must be known. The radius is controlled by the “Move” buttons. “Move In” decreases the radius and similarly “Move Out” increases the radius. The spherical to Cartesian coordinate conversion uses the following equations

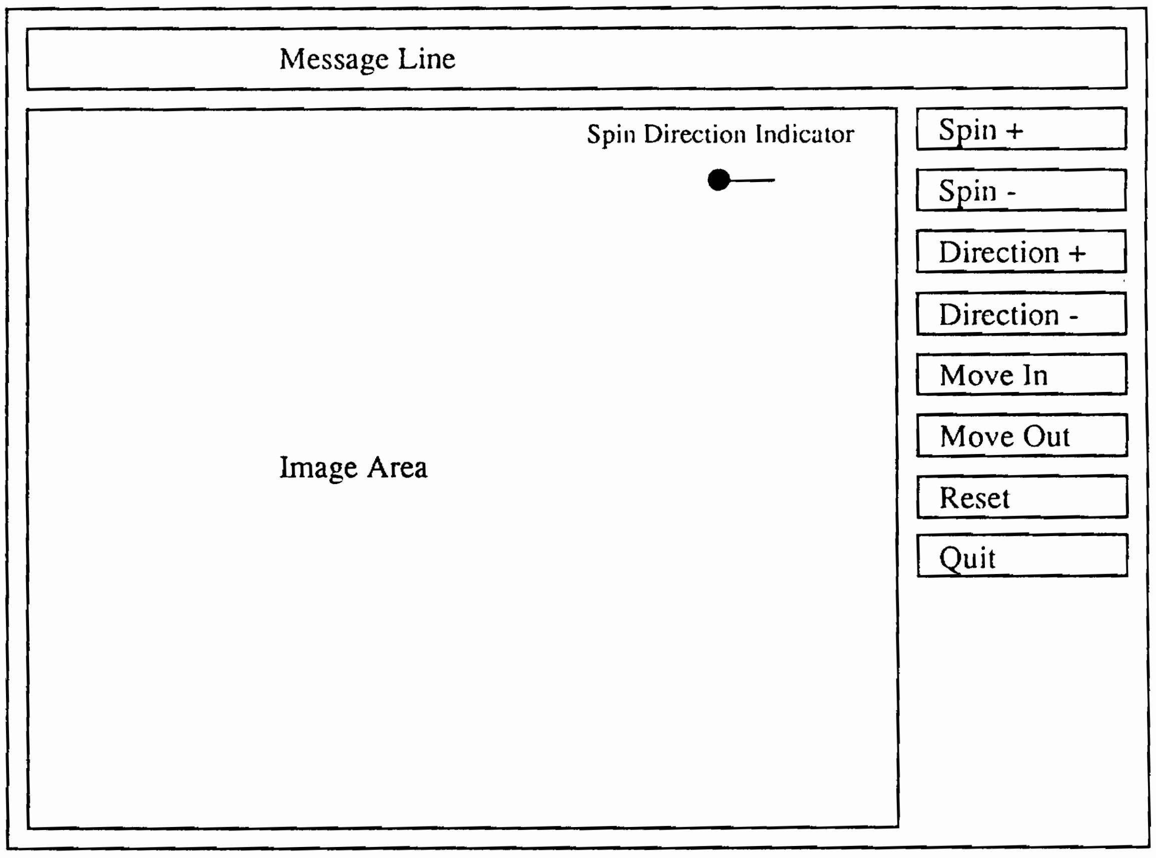

Figure 4.1: User interface controls

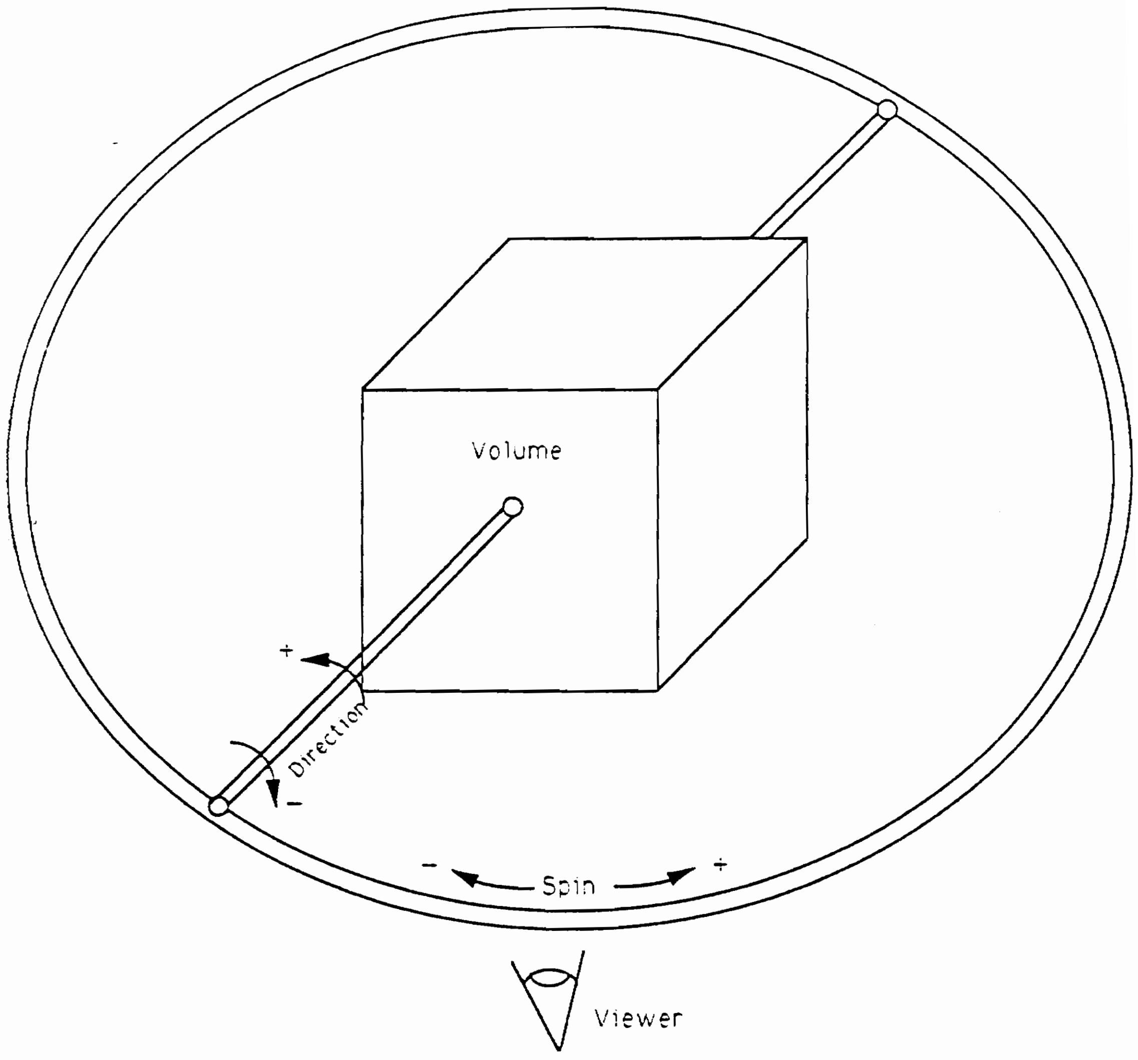

Figure 4.2: How the "Spin" and "Direction" buttons affect the viewer's position

[Berk84].

$$\begin{aligned}
viewposition_x &= (r * sin\phi * cos\theta) + center_x \\
viewposition_y &= (r * sin\phi * sin\theta) + center_y \\
viewposition_z &= (r * cos\phi) + center_z
\end{aligned}$$

4.2 Determining the Position of a Node

The viewing algorithm processes the pyramid in breadth first order to provide the progressive refinement. To process the next octant, the algorithm must compute the position and size of this octant within the volume. Our first approach to achieve a breadth first ordering was to use a queue. For each octant we calculated the position and size of its eight children. For those children whose maximum value was greater than the "empty" value, this information was added to the queue in the proper order. This allowed us to skip areas that contained only empty values and ensure a view consistent ordering of the octants. A view consistent ordering means the data is accessed in the same order with respect to the viewer from any view position. This ordering could be front to back, for example. The maximum length of the queue was the number of octants at the lowest level of the pyramid. Each element contained an *x,y,z, data location, and size.* The *x, y, and z* defined one corner. The size allowed us to computer the other corners. The data location was the location in the heap of the intensity value. Alternatively, we could have stored the data value itself in the queue. This required at least five bytes per element. The resultant queue size was two and a half times larger than the original dataset. That was certainly not space efficient.

The next approach was to calculate the *x,y,z* location from the position in the heap, which was readily available if the pyramid was processed in the same order in which it was stored. We developed a routine that would allow us to skip "empty" portions of the pyramid by using the minimum and maximum values. If the current node's minimum value was less than the "empty" value, then the parent of that node was calculated and its maximum value was compared to the "empty" value. If its maximum value was less than the "empty" value then none of its children would need to be rendered. We continued to move up the pyramid until we found a parent with a maximum value greater than "empty". We then chose the next child of this parent, and moved back down to the level at which we started. In this fashion we could move up and down the pyramid as we moved across one level. This allowed us to skip portions of the data that would not contribute to the image. This worked well, until we started compositing colors. Because we allow the user to change view positions, we can not insure a front to back or back to front processing order as required by the compositing routine.

The solution we have implemented is a modified recursive approach that ensures a view consistent processing order. A normal recursive algorithm, such as Aref and Samet's [Aref91], can ensure a view consistent processing order, but processes the data in a depth first manner. Our approach enables breadth first processing, and hence progressive refinement. Our algorithm consists of a loop that calls a routine named *ProcessLevel* for each level of the pyramid in turn. *ProcessLevel* is recursive. It accepts three parameters, *CurrentLevel, DrawingLevel, and Octant. ProcessLevel* always starts at the root of the pyramid. If *CurrentLevel* is not equal to *DrawingLevel*, then the routine subdivides *Octant* into its children and recursively calls itself in the correct order. The correct order is determined as follows. The view position is compared to the center of *Octant.* The view position lies in one of eight regions with respect to the center of the node as illustrated in Figure 4.3. The view position can be in the Left Upper Front (LUF) region, the Right Upper Front (RUF), the Left Down Front (LDF), the Right Down Front (RDF), the Left Upper Back (LUB), the Right Upper Back (RUB), the Left Down Back (LDB), or the Right Down Back (RDB). The region that the view position is in is used as an index into the Octant Order Table, see Table 4.1. The Octant Order Table indicates a view consistent ordering of the children. When *DrawingLevel* is reached, the octants are drawn and composited as described in the following sections. This approach achieves two goals. First "empty" areas of the pyramid are never processed. If the maximum value of an *Octant* is less than the "empty" value, then that *Octant* is not subdivided. None of its children are processed. The second goal is

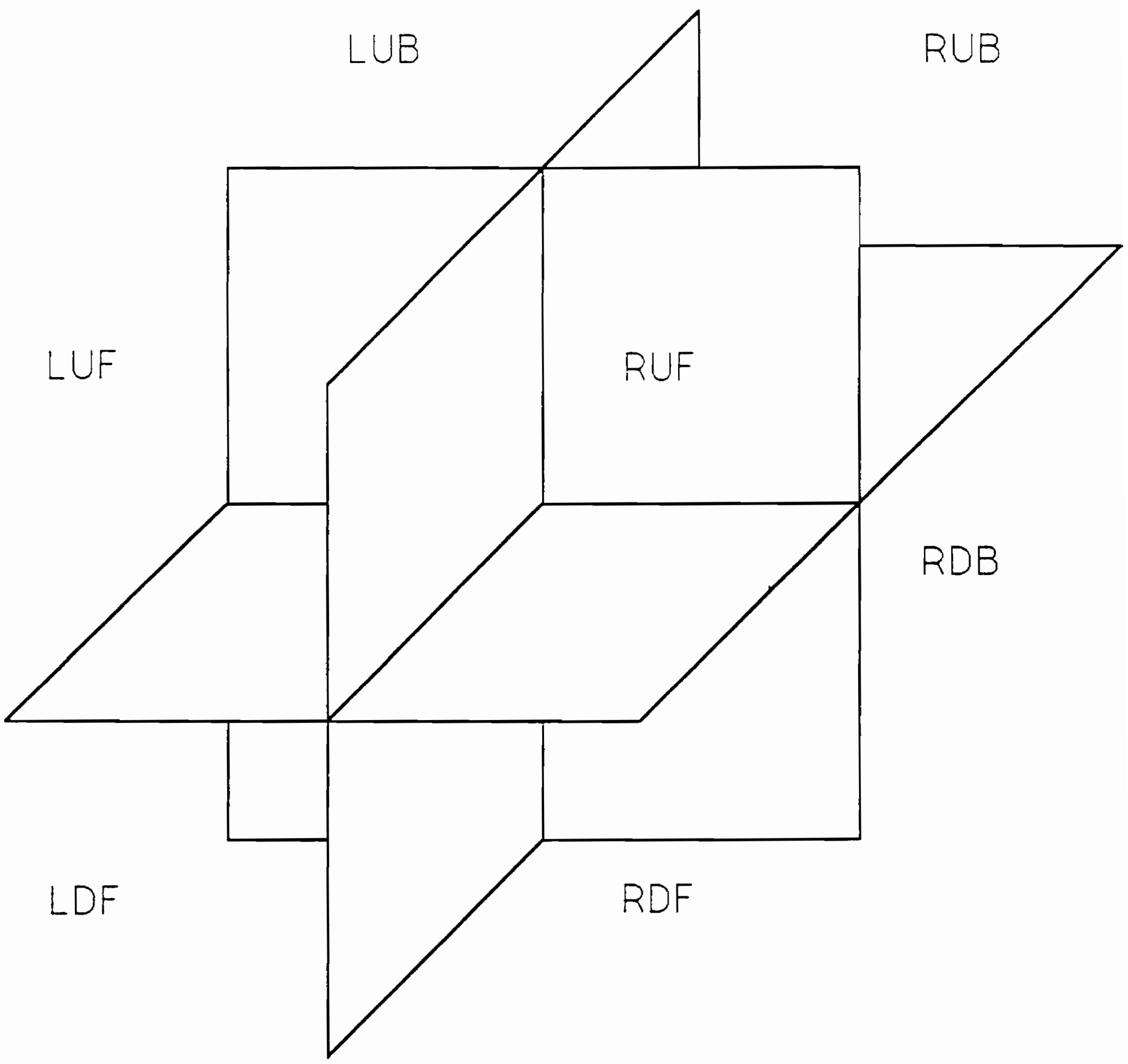

Figure 4.3: Eight regions that the view position can be in

Table 4.1: Octant Order Table used to determine child processing order

region	*Octant Order*							
LUF	LUF	RUF	LDF	RDF	LUB	RUB	LDB	RDB
RUF	RUF	LUF	RDF	LDF	RUB	LUB	RDB	LDB
LDF	LDF	RDF	LUF	RUF	LDB	RDB	LUB	RUB
RDF	RDF	LDF	RUF	LUF	RDB	LDB	RUB	LUB
LUB	LUB	RUB	LDB	RDB	LUF	RUF	LDF	RDF
RUB	RUB	LUB	RDB	LDB	RUF	LUF	RDF	LDF
LDB	LDB	RDB	LUB	RUB	LDF	RDF	LUF	RUF
RDB	RDB	LDB	RUB	LUB	RDF	LDF	RUF	LUF

the proper ordering of octants from any view position. Table 4.1 provides a front to back octant ordering. A back to front processing could be accomplished by simply replacing Octant Order Table with a back to front version.

4.3 Projection Of Coordinates

Several transformations must be done in order to project an octant correctly onto the screen. The viewing reference coordinate (VRC) system is defined by three axis. The projection plane (or view-plane) is the 2D plane to which the 3 dimensional coordinates of the volume are projected to form the image. One axis is the view-plane normal (VPN), a vector that is normal to the view-plane. A second axis is found from the view up vector (VUP). The projection of the VUP onto the view plane is called the v-axis. The third axis is called the u-axis. Its direction is defined such that u,v, and VPN form a right handed coordinate system. To a viewer at the view position, the volume is defined by this coordinate system. Since the view-plane is defined by the view position, this entire coordinate system changes when the view position does. The VRC may or may not coincide with the coordinate system of the dataset, or the device coordinates of the computer screen. Thus we must transform the VRC to the device coordinates, such that the screen becomes the view plane, and the view position is in front of the screen at some distance. To accomplish this, the VRC must be rotated so the VPN becomes the z-axis, the u-axis becomes the x-axis, and the v-axis becomes the y-axis. The geometric transformations required for this are consolidated into a single matrix. By multiplying each x,y,z coordinate from the volume by this matrix, the coordinates are transformed such that a projection routine can project it onto the plane $z = 0$ (equivalent to the screen). The routines to generate this matrix are based on the traditional techniques described in Foley et al [Fole90] and Watt [Watt89]. They are given in Figure 4.4 and Figure 4.5.

Each point is multiplied by the resultant matrix to produce a transformed point. This transformed point is then projected onto the view-plane. The correct projection point is the intersection of a line between the transformed point and the view position, and the view-plane. The view-plane is now the $z = 0$ plane and the view position lies at a distance dz from the view-plane on a line that is parallel to the z-axis. The distance dz is equal to the center of the object minus the radius. We specify this parametrically as

$$\begin{aligned} point_x &= viewposition_x + t(TransformedPoint_x - viewposition_x) \\ point_y &= viewposition_y + t(TransformedPoint_y - viewposition_y) \end{aligned}$$

where t is defined over the range $0 <= t <= 1$ as

$$-TransformedPoint_z/(dz - TransformedPoint_z)$$

4.4 Compositing

Compositing is required when semi-transparent solids are rendered. Semi-transparent rendering requires an opacity as well as a color for each voxel. In a semi-transparent solid, the color that is mapped to a screen location depends not only on the object that is closest to the viewer at that location, but also to a lesser extent on the colors of objects farther away also projecting to that location, due to the opacity. Medical imaging devices do not record multiple values for each sample position. Most three dimensional datasets contain a

```
void compute_matrices(float matrix3[4][4], Point3D offset,
                      int size, Point3D viewposition)
{
  FPoint3D Rx,Ry,Rz,VPN,VUP,VRP,DOP;
  float matrix1[4][4],matrix2[4][4];
  int i,j;

  /* view reference point is the center of the object */
  VRP.x = offset.x + (size/2);
  VRP.y = offset.y + (size/2);
  VRP.z = offset.z + (size/2);
  /* compute the view plane normal from the viewposition
     and the view ref pt */
  VPN.x = viewposition.x - VRP.x;
  VPN.y = viewposition.y - VRP.y;
  VPN.z = viewposition.z - VRP.z;

  /* set the view up vector to be a unit vector on the y axis */
  VUP.x = 0;
  VUP.y = 1;
  VUP.z = 0;

  /* retranslate VRP to original coordinates */
  translate(VRP, matrix2);

  /* figure rotations */
  rotateVRC(VPN, VUP, &Rx, &Ry, &Rz);
  matrix1[0][0] = Rx.x;
  matrix1[0][1] = Rx.y;
  matrix1[0][2] = Rx.z;
  matrix1[1][0] = Ry.x;
  matrix1[1][1] = Ry.y;
  matrix1[1][2] = Ry.z;
  matrix1[2][0] = Rz.x;
  matrix1[2][1] = Rz.y;
  matrix1[2][2] = Rz.z;
```

Figure 4.4: Transformation matrix computation routine, continued on next figure

```
  for(i=0;i<3;i++){
    matrix1[3][i] = 0.0;
    matrix1[i][3] = 0.0;
  }
  matrix1[3][3] = 1.0;

  /* multiply translation and rotation matrices */
  matrix_mult(matrix2, matrix1, matrix3);

  /* translate VRP to origin */
  translate(VRP, matrix1);

  /* multiply trans*rotate and 2nd trans matrices */
  matrix_mult(matrix3, matrix1, matrix2);
```

```
void rotateVRC(FPoint3D VPN, FPoint3D VUP, FPoint3D *Rx,
               FPoint3D *Ry, FPoint3D *Rz)
{
  FPoint3D VUPXRz;
  float lenVPN,lenVUPXRz;

  lenVPN = vector_length(VPN);  /* calculate the length of VPN */

  /* the VPN is rotated onto the z-axis */
  Rz->x = VPN.x / lenVPN;
  Rz->y = VPN.y / lenVPN;
  Rz->z = VPN.z / lenVPN;

  /* the u-axis is rotated onto the x-axis */
  VUPXRz = cross_prod(VUP,*Rz);  /* u is perpendicular to VUP and VPN */
  lenVUPXRz = vector_length(VUPXRz);/* which is the cross product     */
  Rx->x = VUPXRz.x / lenVUPXRz;
  Rx->y = VUPXRz.y / lenVUPXRz;
  Rx->z = VUPXRz.z / lenVUPXRz;

   /* the v-axis is perpendicular to Rx and Rz so take cross product */
  *Ry = cross_prod(*Rz,*Rx);
}
```

Figure 4.5: Rotation matrix computation routine

single value for each discrete voxel location. That value may represent density, temperature, color, intensity, or other measurement, depending on the application generating the data. It is up to the algorithm designer to decide how to map these values into color and opacity. We chose to regard the data values in our test dataset as gray-scale intensities. We assign opacities to be proportional to the intensity. Empty space has a low intensity and therefore a low opacity. Bone has a high intensity and therefore a high opacity. For this particular dataset, an opacity equal to 1/4 of the intensity works well.

As the internal levels of the pyramid are processed, the minimum data value is read for a node. If this value is above the empty threshold, the octant is transformed as described in the previous section, and projected onto the screen. The cube projects as either one, two, or three polygons. Depending on the view position, the user sees either one, two, or three faces of the cube. The polygons are "drawn" to a pixmap, which is an X Windows term for a chunk of memory that can be accessed with screen coordinates, but does not exist in the frame-buffer and so does not appear on screen. A rectangular section of the pixmap denoted by x,y coordinates is compared to a rectangular section, at the same x,y coordinates of the image that has been rendered so far. The new pixmap section is composited with the existing image. The processing order determines which compositing equations to use. Our processing order is front to back so we use:

$$\begin{aligned} C_{acc} &= ((1 - O_{acc}) * (C_{new} - C_{acc})) + C_{acc} \\ O_{acc} &= ((1 - O_{acc}) * O_{new}) + O_{acc} \end{aligned}$$

where C_{acc} is the accumulated color in the existing image, C_{new} is the new color from our pixmap, O_{acc} is the accumulated opacity in the existing image, and O_{new} is the new opacity from our pixmap. These equations are modified versions of the compositing equations given by Wilhelm [Wilh91]. Each pixel in the rectangular section must go through these equations, since a cube may only partially overlap its predecessor. Current graphics workstations often contain a Z buffer which could be used to handle the accumulated color, but hardware opacity buffers are not common at this time.

When the lowest level of the pyramid is reached, the eight adjoining data values are averaged to obtain a cube that can be composited just like those in the upper levels.

4.5 Modifications to Speed the Rendering Time

Since we store in each node the maximum data value of that node's children, we can determine if a particular section of the octree can be ignored. If the maximum value in a node is below some threshold, then it is assumed that the data in that node and all its children represent empty voxels. Thus everything below it need not be processed. The threshold value has to be determined for each dataset individually. Medical image datasets such as the one we used can contain large numbers of empty voxels. In our dataset voxels range in value from 2 to 4095. We have found that voxel values below 480 generally represent empty space. This constitutes 59% of the voxels in the original dataset. Skipping these voxels through our modified recursion process reduces both the internal level and total image generation times significantly. Internal level rendering using this approach takes only 2.1% of the time it takes to render every internal node of the pyramid. Total rendering time using this approach takes only 12.4% of the time to render every node in the pyramid.

In order to facilitate changing the view position while the rendering process is executing, an interrupt routine had to be written. Under X Windows, this is done by polling the X event queue to respond to events during the process of rendering the image. If the X event queue were not polled by the algorithm, the button clicks that should rotate the volume would be queued up and not handled until the entire volume was processed. By polling the X event queue, waiting events are dispatched every time the polling routine is executed. Initially the X event queue was polled for every data value read from the pyramid. This provided excellent response time for changing view position, but increased the rendering time of the upper and lower levels. By reducing the polling to one time for every 100 data values read, the rendering time for the upper levels was cut by 66%. The total rendering time was reduced by 72%.

Chapter 5

RESULTS AND ANALYSIS

5.1 Hardware Used

The viewing algorithm was developed in the C language, running under UNIX. The development hardware was an IBM PC compatible 80386 based machine running at 40MHz, with a 40MHz math co-processor. The display hardware was an Orchid ProDesigner IIs VGA graphics adapter and a Magnavox color VGA display. The program was then ported to a DECstation 5000 PXG Turbo. The display hardware on the DECstation is a 24-bit Truecolor graphics processor and color monitor. The port was accomplished in less than one day, and consisted mainly of modifications relating to the differences between 8-bit and 24-bit displays.

5.2 Dataset Used

The program was used to view a Magnetic Resonance (MR) scan of a human head. The original dataset dimensions were $x = 256$, $y = 256$, $z = 109$. Each data element consisted of one sixteen bit integer. Total file size was 14,286,848 bytes. To this file were added nineteen more x-y planes to make the resultant z dimension equal to 128. This was done to simplify the pyramid building process. Since 128 is a power of 2, it will divide completely and evenly into octants. This will require only four additional nodes to be added by the pyramid building process. These added data elements were all set to a predetermined value that did not exist in the original dataset. They were termed "empty", and as such, could be bypassed by our viewing algorithm. The resulting file size was 16,777,216 bytes. This represented the lowest level of our pyramid. The internal levels of the pyramid together occupied 2,396,754 bytes. This made a total pyramid file size of 19,173,970 bytes. One quarter was chosen as the amount of lowest level data to be stored in memory. Since this dataset was a half-cube, with dimensions of 256 × 256 × 128, this would be equivalent to one octant of a full-cube of dimensions 256 × 256 × 256. The amount of lowest level data in memory at one time was then 4,194,304 bytes.

5.3 Performance

Image generation times using our algorithm are not greatly affected by the viewer's position. They are affected by the combination of hardware and X Windows server, however. Performance on the PC was not outstanding, but reasonable considering the task. The program could display the internal levels of the pyramid in approximately 2.5 minutes. Display of the entire pyramid took approximately 50 minutes.

Performance on the DEC Station is currently comparable to the algorithms described in Chapter 2. Complete display of the internal levels of the pyramid takes approximately 40 seconds. The total image takes approximately 11.5 minutes. More detailed timings are given below. The DEC station is not a high end graphics workstation. Unfortunately, we did not have access to such a workstation to perform timings on.

Special graphics hardware will only help our algorithm when the X server is written to take advantage of it. Following are some hardware/server abilities that would speed image generation times.

- Hardware polygon fills would speed image generation times since we do generate a significant number of polygons.

Table 5.1: Processing statistics per level of the pyramid

Level	*Total Elapsed Time*	*No of Polygons*	*No of Points*
1	0 secs	0	0
2	0 secs	0	0
3	0 secs	0	0
4	0 secs	0	0
5	0 secs	0	0
6	2 secs	833	19520
7	38 secs	28,961	176,176
8	684 secs	650,914	668,196

- Hardware projection from three dimensions to two dimensions could be used since we project each octant in the pyramid before it is rendered.
- Hardware compositing would speed image times. Compositing accounts for 5.5% of our total processing time.
- A hardware opacity buffer, and multiple frame-buffers that are directly accessible from X Windows would eliminate several time consuming steps.
- Hardware polygon compositing would save even more time.

Table 5.1 shows the time required to draw the pyramid. The first column indicates the level of the pyramid completed. The second column shows the total elapsed time. Entries of 0 seconds indicate times less than the resolution of our timer. Column three shows the total number of polygons drawn. Column four shows the total number of points that have been composited. The time in Table 5.1 for level 8 includes disk access time. Total disk access time for the lowest level is 16 seconds. Table 5.2 shows the distribution of time spent in the various rendering routines. The vast majority of the time, 61.7%, is spent copying the rectangular portion of the pixel array (pixmap) that stores the projected polygons into an X Windows structure called an Ximage so that it can be composited. This is accomplished by one call to an X Windows library function. This copy operation is required because of the following two facts about current X Windows image handling routines.

- Polygons can only be drawn to a window or a pixmap.

Table 5.2: Percentage of time spent in rendering routines

Routine	*Percentage of Time*
Selecting Octants	0.3%
Performing Transformations	11.1%
Projecting Coordinates	2.8%
Calculating Polygonal Faces	1.5%
Drawing Polygons to Pixmap	8.8%
Copying Pixmap to Ximage	61.7%
Compositing Pixels	5.5%
Displaying Pixels	6.5%

- Pixels can only be accessed from an Ximage.

Somehow, the pixmap has to be copied to an Ximage for access by the compositing routine. The routine that X Windows provides for this purpose, XGetSubImage, is responsible for 61.7% of our processing time. A significant time reduction could be achieved if some other way were found to copy the information, or direct access to the pixmap were allowed.

The levels of the pyramid are drawn as projected cubes. From the center front view position, which should require the fewest polygons, 650,914 polygons were drawn. Each one of those polygons was then composited on a pixel-by-pixel basis. The total number of points that were composited was 668,196. This amounts to an average rate of 962 pixels composited per second.

Table 5.3 is a comparison of our new rendering algorithm to some of the rendering algorithms discussed in Chapter 2. Levoy's algorithm is a ray tracing algorithm and Wilhelm and Van Gelder's is an isosurface projection algorithm. These two papers were the only ones discussed in Chapter 2 that provided timings for datasets of equivalent sizes. Our rendered dataset is shown in Figure 5.1.

Table 5.3: Comparison of rendering times

Algorithm	*Hardware Platform*	*Dataset Size*	*Total Rendering Time*
New Algorithm	DECstation 5000/200	256 x 256 x 109	684 secs
[Wilh90b]	Sun Sparcstation 1	256 x 256 x 109	391.8 secs
[Levo90]	Sun 4/280	256 x 256 x 113	105 secs

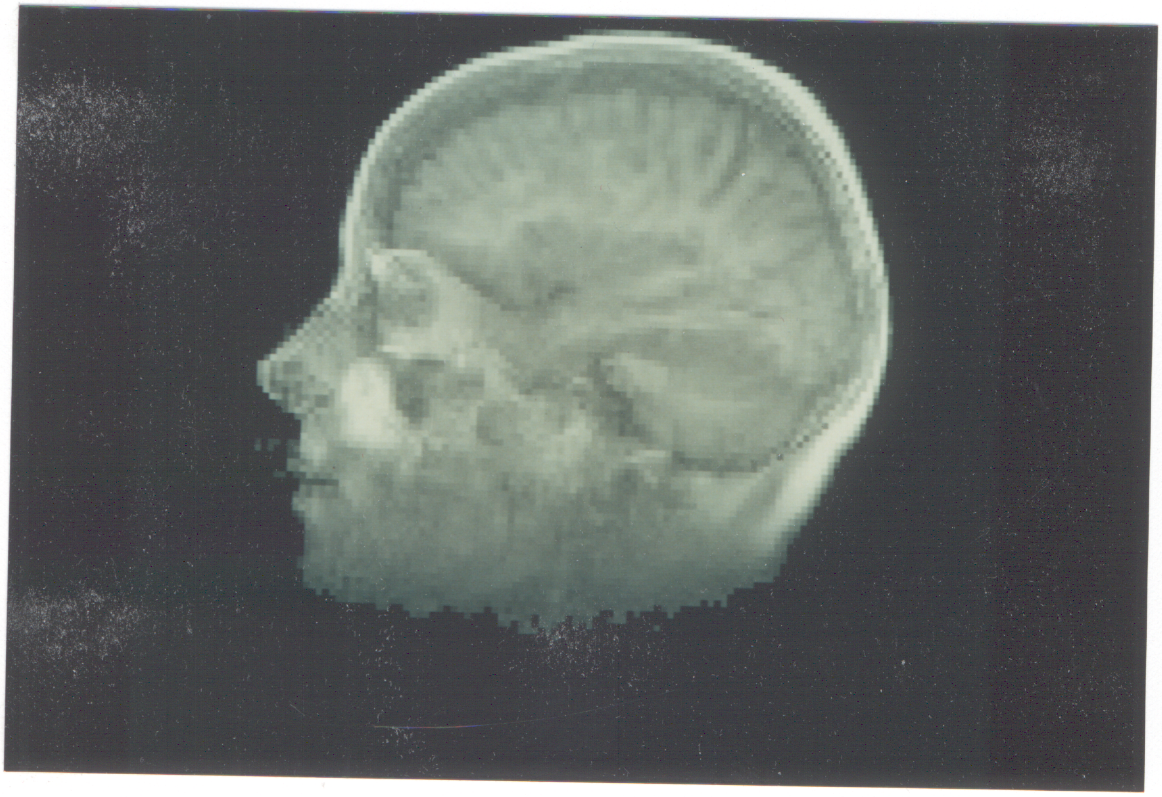

Figure 5.1: MR scan of human head

Chapter 6

CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

Volume rendering continues to be a fast changing area of computer graphics. Timing results on our current hardware do not show our approach to be a faster overall algorithm for volume rendering. However, the algorithm could benefit substantially from faster hardware and a better X server. At this time, there is not an X Windows standard for accessing the pixmap directly. Hardware vendors such as Hewlett Packard are developing routines to allow this however. When a standard is implemented, this algorithm could see substantial reductions in rendering time by taking advantage of pixmap accessibility.

The algorithm does currently provide several advantages. These advantages include the following.

- It provides some indication of object orientation quickly. Within 40 seconds, a user familiar with the dataset can usually determine the general orientation of the object(s).
- It can handle large datasets with minimal memory requirements. The algorithm can be tuned to take advantage of the memory available. The minimum memory requirement is determined by the size of the internal levels of the pyramid. The amount of data

from the lower level that is stored in memory can be any size from one page to the entire level.

- Our algorithm provides a perspective view of the volume. A perspective image is easier for the viewer to interpret than a parallel projection, since this is what our visual system is accustomed to. Objects of a given size that are farther from the viewer appear smaller than those of the same size that are nearer the viewer.
- It provides device independence. Since the algorithm is written to operate under X Windows, it can be ported from one platform to another with little or no changes. Device independence makes this algorithm especially attractive to many different prospective users. A small town doctor can render medical images just like a city hospital, but without the expense of a dedicated graphics workstation. University departments that can not or will not decide on a single hardware vendor can run the same application on different platforms. Businesses that upgrade to different hardware platforms can port their application with little or no changes. Porting the application from an IBM PC to a DEC Station took less than one day.

6.2 Future Work

There are several ways in which this work could be advanced. In the area of compositing routines, the user could be allowed to select between several different compositing routines, or even write his own. The algorithm was deliberately written to separate the compositing routine so that it could be changed if needed. A menu of prewritten compositing routines could be supplied. One of the menu options could be "other", which opens a dialogue box to specify a user written routine. The user could be allowed to adjust the opacity. This could be handled by an on-screen slider that adjusts the proportionality constant for the opacity. Image processing techniques could be incorporated. These could include providing a way to adjust the contrast and intensity of the image. This could be accomplished via on-screen brightness and contrast "knobs". The user could be allowed to interactively change the color thresholds. A dialogue box would probably be best for this, since the data value range is dataset-specific.

Other areas of future work might entail exploration of parallel processing. Our approach should work well as an object-parallel algorithm as described in Chapter 1. The algorithm is readily scalable for different numbers of processors. If only one processor is available, it starts at the root and handles the entire pyramid. If eight processors are available, each starts at one level below the root and handles one eighth of the pyramid. Adding a factor of eight processors allows the processing to start one level lower in the pyramid. A processor is assigned to each octant. Each processor produces a two dimensional section of the image along with an opacity array for that section. These sections can then be composited together to form the complete image.

REFERENCES

[1] [Aref91] W.G. Aref and H. Samet, Perspective Viewing of Objects Represented by Octrees, Center for Automation Research CS-TR-2757, University of Maryland, College Park, Maryland, September 1991.

[2] [Bent85] J. Bentley, Thanks Heaps, *Communications of the ACM*, 28(3), March 1985, 245-250.

[3] [Berk84] D. Berkey, *Calculus*, Saunders College Publishing, Philadelphia, PA, 1984, 804-806.

[4] [Cohe88] M. Cohen, S. Chen, J. Wallace, and D. Greenberg, A Progressive Refinement Approach to Fast Radiosity Image Generation, *Computer Graphics*, 22(4), August 1988, 75-84.

[5] [Dreb88] R. Drebin, L.Carpenter, and P. Hanrahan, Volume Rendering, *Computer Graphics*, 22(4), August 1988, 65-74.

[6] [Fole90] J. Foley, A. vanDam, S.K. Feiner, and J.F. Hughes, *Computer Graphics Principles and Practice*, Addison Wesley Publishing Company, Reading, MA, 1990.

[7] [Laur91] D. Laur and P. Hanrahan, Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering, *Computer Graphics*, 25(4), July 1991, 285-288.

[8] [Levo90] M. Levoy, Efficient Ray Tracing of Volume Data, *ACM Transactions on Graphics*, July 1990, 245-261.

[9] [Max90] N. Max, P. Hanrahan, and R. Crawfis, Area and Volume Coherence for Efficient Visualization of 3D Scalar Functions, *Computer Graphics*, 24(5), November 1990, 27-33.

[10] [Meag82] D. Meagher, Geometric Modeling Using Octree Encoding, *Computer Graphics and Image Processing*, 19(2), June 1982, 129-147.

[11] [Novi90] K.L. Novins, F.X. Sillion, and D.P. Greenberg, An Efficient Method for Volume Rendering Using Perspective Projection, *Computer Graphics*, 24(5), November 1990, 95-102.

[12] [Sabe88] P. Sabella, A rendering Algorithm for Visualizing 3D Scalar Fields, *Computer Graphics*, 22(4), August 1988, 51-58.

[13] [Same90] H. Samet, *Applications of Spatial Data Structures*, Addison-Wesley Publishing Company Inc., 1990.

[14] [Same90] H. Samet, *The Design and Analysis of Spatial Data Structures* , Addison-Wesley Publishing Company Inc., 1990.

[15] [Tani75] S. Tanimoto and T. Pavlidis, A Hierarchical Data Structure for Picture Processing, *Computer Graphics and Image Processing*, 4(2), June 1975, 104-119.

[16] [Upso88] C. Upson, and Michael Keeler, V-BUFFER: Visible Volume Rendering, *Computer Graphics*, 22(4), August 1988, 59-64.

[17] [Watt89] A.Watt, *Fundamentals of Three-Dimensional Computer Graphics*, Addison Wesley Publishers Ltd., Wokingham, England, 1989.

[18] [Wilh90a] J. Wilhelms, J. Challinger, N. Alper, and S. Raamoorthy, Direct Volume Rendering of Curvilinear Volumes, *Computer Graphics*, 24(5), November 1990, 41-47.

[19] [Wilh90b] J. Wilhelms and A. Van Gelder, Octrees for Faster Isosurface Generation (Extended Abstract), *Computer Graphics*, 24(5), November 1990, 57-60.

[20] [Wilh91] J. Wilhelms and A. Van Gelder, A Coherent Projection Approach for Direct Volume Rendering, *Computer Graphics*, 25(4), July 1991, 275-284.

VITA

Tim Ryan was born in Cedar Falls, Iowa on November 9, 1963. He attended Clemson University where he graduated with a BS in Computer Science in December 1985. After graduation he was employed by Information Systems Development, a division of the Clemson University Computing Center. In 1988 Mr. Ryan went to work for the Naval Surface Warfare Center in Dahlgren, Virginia. Through NAVSWC, he began his graduate degree in Computer Science at Virginia Polytechnic Institute and State University. In August of 1990, Mr. Ryan left the Naval Surface Warfare Center to pursue his degree full-time. After graduating from Virginia Tech, Mr. Ryan plans to pursue a career with Texas Instruments in Dallas, Texas.