Mississippi State University
Scholars Junction

Theses and Dissertations | Theses and Dissertations

5-10-2003

Rzsweep: A New Volume-Rendering Technique for Uniform Rectilinear Datasets

Gautam Chaudhary

Follow this and additional works at: https://scholarsjunction.msstate.edu/td

Recommended Citation

Chaudhary, Gautam, "Rzsweep: A New Volume-Rendering Technique for Uniform Rectilinear Datasets" (2003). *Theses and Dissertations*. 3845.
https://scholarsjunction.msstate.edu/td/3845

This Graduate Thesis - Open Access is brought to you for free and open access by the Theses and Dissertations at Scholars Junction. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Scholars Junction. For more information, please contact scholcomm@msstate.libanswers.com.

RZSWEEP: A NEW VOLUME-RENDERING TECHNIQUE FOR UNIFORM RECTILINEAR DATASETS

By

Gautam Chaudhary

A Thesis
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computer Science
in the Department of Computer Science and Engineering

Mississippi State, Mississippi

May 2003

RZSWEEP: A NEW VOLUME-RENDERING TECHNIQUE FOR

UNIFORM RECTILINEAR DATASETS

By

Gautam Chaudhary

Approved:

Hasan Jamil
Assistant Professor of Computer Science and Engineering
(Major Professor)

Ricardo Farias
Adjunct Assistant Professor of Computer Science and Engineering
(Thesis Director)

Edward B. Allen
Assistant Professor of Computer Science and Engineering
(Committee Member)

David A. Dampier
Assistant Professor of Computer Science and Engineering
(Committee Member)

Susan M. Bridges
Professor of Computer Science and Engineering
Graduate Coordinator
Department of Computer Science and Engineering

A. Wayne Bennett
Dean of the College of Engineering

Name: Gautam Chaudhary

Date of Degree: May 10, 2003

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Dr. Hasan Jamil

Director of Thesis: Dr. Ricardo Farias

Title of Study: RZSWEEP: A NEW VOLUME-RENDERING TECHNIQUE FOR UNIFORM RECTILINEAR DATASETS

Pages in Study: 84

Candidate for Degree of Master of Science

A great challenge in the volume-rendering field is to achieve high-quality images in an acceptable amount of time. In the area of volume rendering, there is always a trade-off between speed and quality. Applications where only high-quality images are acceptable often use the ray-casting algorithm, but this method is computationally expensive and typically achieves low frame rates. The work presented here is RZSweep, a new volume-rendering algorithm for uniform rectilinear datasets, that gives high-quality images in a reasonable amount of time. In this algorithm a plane sweeps the vertices of the implicit grid of regular datasets in depth order, projecting all the implicit faces incident on each vertex. This algorithm uses the inherent properties of a rectilinear datasets. RZSweep is an object-order, back-to-front, direct volume rendering, face projection algorithm for rectilinear datasets using the cell approach. It is a single processor serial algorithm. The simplicity of the

algorithm allows the use of the graphics pipeline for hardware-assisted projection, and also, with minimum modification, a version of the algorithm that is graphics-hardware independent. Lighting, color and various opacity transfer functions are implemented for giving realism to the final resulting images. Finally, an image comparison is done between RZSweep and a 3D texture-based method for volume rendering using standard image metrics like Euclidian and geometric differences.

DEDICATION

To my grandfather, grandmother, parents, my younger brother and all others in my family.

ACKNOWLEDGMENTS

The findings and opinions in this thesis belong solely to the author and do not necessarily represent those of Mississippi State University.

I would like to take this opportunity to first extend my sincere gratitude to Dr. Ricardo Farias for directing this research. The implementation of the plane-sweep paradigm for volume rendering of uniform rectilinear datasets, implementation of lighting, color and opacity to the algorithm was carried out under his guidance. I would like to thank Lakshmy Ramaswamy for her help in this research. I thank Dr. Robert Moorhead for his valuable suggestions and opinions from time to time, Dr. Joerg Meyer and my committee members for their comments and corrections on this thesis. I am also thankful to Sagar Saladi and Pujita Pinnamaneni for providing some of the results of their work and the entire graphics and visualization group of the Department of Computer Science and Engineering Research Center (ERC) of Mississippi State University. Finally, I thank the research community for providing the datasets that have been used in this work.

TABLE OF CONTENTS

Page

DEDICATION ii

ACKNOWLEDGMENTS iii

LIST OF TABLES vi

LIST OF FIGURES vii

CHAPTER

I. INTRODUCTION 1

1.1 **Volume Rendering** 1
1.1.1 *What is it ?* 1
1.1.2 *Uses of volume rendering* 2
1.1.3 *Main steps in volume rendering* 3
1.1.4 *Methods of data acquisition* 4
1.1.5 *Classification of volume rendering methods* 4
1.1.6 *Bottlenecks faced in volume rendering* 8
1.2 **Techniques Used for Making Results More Realistic** 9
1.3 **Classification of Datasets** 11
1.4 **Sweep Paradigm** 12
1.5 **Research Proposal** 12

II. LITERATURE REVIEW 14

2.1 **Volume Rendering Algorithms for Regular Datasets** 14
2.1.1 *Ray-casting algorithms* 15
2.1.2 *Splatting algorithms* 16
2.1.3 *Shear-warp factorization algorithms* 17
2.1.4 *Texture mapping algorithms* 18
2.2 **Sweep Based Volume Rendering Algorithms** 18
2.3 **Review on Transfer Functions** 19

CHAPTER Page

III. RZSWEEP 21

3.1 **Significance of RZSweep** 22
3.2 **The Basic RZSweep Algorithm** 22
3.3 **Research Issues** 26
3.4 **Implementation Details** 27
3.4.1 *Preprocessing and basic structures* 27
3.4.2 *Sweep* 28
3.4.3 *Projection* 29
3.4.3.1 Hardware 29
3.4.3.2 Software 30
3.4.4 *Optimizations* 30
3.4.4.1 Projecting faces within threshold 31
3.4.4.2 Volume reduction 31
3.4.4.3 Faces reduction 31
3.4.4.4 Connected datasets 32
3.5 **Slice Viewer** 33
3.6 **Lighting, Color and Opacity Transfer Functions** 33
3.6.1 *Lighting* 35
3.6.2 *Color* 37
3.6.3 *Opacity* 37

IV. RESULTS 44

4.1 **RZSweep Timings** 45
4.2 **RZSweep Images** 47
4.3 **RZSweep Image Comparison** 60
4.4 **SGI Machine's System Hardware** 75

V. CONCLUSIONS 77

5.1 **Contributions** 77
5.2 **Further Research** 79

REFERENCES 80

LIST OF TABLES

TABLE Page

4.1 Dataset details . . . 45

4.2 Hardware and software rendering times of RZSweep . . . 47

4.3 Hardware rendering times of RZSweep . . . 47

4.4 Rendering times of RZSweep for different image sizes of Engine dataset . . . 48

4.5 Preprocessing times of RZSweep for different datasets . . . 49

4.6 RZSweep timing without any optimizations . . . 49

4.7 RZSweep timing with volume reduction . . . 49

4.8 RZSweep with eight and four projected faces . . . 50

4.9 RZSweep with connected data optimization . . . 50

4.10 RZSweep after implementation of light, color and opacity . . . 50

4.11 Difference image list . . . 64

4.12 Euclidian difference between RZSweep and 3D texture-based rendering . . . 64

4.13 Geometric difference between RZSweep and 3D texture-based rendering . . . 64

LIST OF FIGURES

FIGURE Page

1.1 Classification of volume-rendering algorithms 5

1.2 Cell representation of volume data 5

1.3 Voxel representation of volume data 6

1.4 Classification of grid types 11

3.1 Twelve incident faces on an internal vertex 23

3.2 Faces that lie ahead of the sweep plane 24

3.3 The schematic representation of RZSweep algorithm 25

3.4 Best cases for face projection 32

3.5 MRI-Head dataset slice with RZSweep slice viewer 34

3.6 Slice of CT-Brain dataset using RZSweep slice viewer 34

3.7 Engine dataset slice with RZSweep slice viewer 35

3.8 A vertex (x_i, y_i, z_i) with its immediate neighbors 36

3.9 Linear transfer function for opacity 39

3.10 Exponential transfer function for opacity 39

3.11 Logarithmic transfer function for opacity 40

3.12 Box transfer function for opacity 40

3.13 Triangle transfer function for opacity 41

FIGURE Page

3.14 Opacity transfer function for bone 42

3.15 Opacity transfer function for soft tissues 42

3.16 Opacity transfer function for bone and soft tissue 43

4.1 Graph showing rendering time for increasing number of data elements 48

4.2 Graph showing the rendering times for different image sizes of Engine dataset 48

4.3 CT-Brain dataset 52

4.4 CT-Skull dataset with $90\,\mathrm{deg}$ rotation about the axis 52

4.5 Engine dataset with $0\,\mathrm{deg}$rotation about the Y axis 53

4.6 Foot dataset with $90\,\mathrm{deg}$ rotation about the Y axis 53

4.7 Fuel dataset 54

4.8 Lobster dataset 54

4.9 MRI-Head dataset 55

4.10 Statue-Leg dataset with $180\,\mathrm{deg}$ rotation about the Y axis 55

4.11 CT-Brain dataset with threshold range of $100 - 255$ 56

4.12 CT-Brain dataset with uniform opacity of 0.005 56

4.13 CT-Brain dataset with lighting 57

4.14 Engine dataset after implementation of a lighting model 57

4.15 CT-Brain dataset with threshold range of $100 - 255$ 58

4.16 CT-Brain dataset after implementing light, color and opacity 58

4.17 Engine dataset with lighting, color and opacity 59

FIGURE Page

4.18 Weight matrix of $\times$ and 5×5 pixels 60

4.19 Image(1): Image generated by RZSweep 65

4.20 Image(2): Image generated by 3D texture-based rendering method 65

4.21 Absolute difference image between Image(1) and Image(2) 66

4.22 Signed difference image between Image(1) and Image(2) 66

4.23 Image(1) $-$ Image(2) 67

4.24 Image(2) $-$ Image(1) 67

4.25 Image(1)$_{3\times3}$: Image after averaging $\times$ pixels of RZSweep 68

4.26 Image(2)$_{3\times3}$: Image after averaging $\times$ pixels of 3D texture-based rendering 68

4.27 Absolute difference image between Image(1)$_{3\times3}$ and Image(2)$_{3\times3}$ 69

4.28 Signed difference image between Image(1)$_{3\times3}$ and Image(2)$_{3\times3}$ 69

4.29 Image(1)$_{3\times3}$ $-$ Image(2)$_{3\times3}$ 70

4.30 Image(2)$_{3\times3}$ $-$ Image(1)$_{3\times3}$ 70

4.31 Image(1)$_{5\times5}$: Image after averaging 5×5 pixels of RZSweep 71

4.32 Image(2)$_{5\times5}$: Image after averaging 5×5 pixels of 3D texture-based rendering 71

4.33 Absolute difference image between Image(1)$_{5\times5}$ and Image(2)$_{5\times5}$ 72

4.34 Signed difference image between Image(1)$_{5\times5}$ and Image(2)$_{5\times5}$ 73

4.35 Image(1)$_{5\times5}$ $-$ Image(2)$_{5\times5}$ 73

4.36 Image(2)$_{5\times5}$ $-$ Image(1)$_{5\times5}$ 74

4.37 Clean image generated by RZSweep with a threshold range of $100 - 255$ . . . 74

CHAPTER I

INTRODUCTION

"... in 10 years, all rendering will be volume rendering."

- Jim Kajiya at SIGGRAPH '91

At the time when Kajiya made the above statement [8], volume rendering used to take several minutes to give some desired resulting images. But today computer capabilities make it possible to produce images of comparable quality within a few seconds.

1.1 Volume Rendering

In order to understand the task of volume rendering, an insight into certain basic concepts and aspects relevant to the field are given here. Most of the explaination, classification and ideas presented in this chapter are drawn from the literature review in Chapter II. The two main survey publications referred here to explain various terms and classifications related to volume rendering are [8, 31].

1.1.1 *What is it ?*

Volume rendering is a sub-area of scientific visualization that has received a lot of attention of the research society due to its importance in the analysis of medical and scien-

tific 3D data. Volume rendering has been defined from many perspectives. The idea that is reflected by most of the definitions is that volume rendering is a method for displaying a sampled 3D scalar field directly, without first fitting intermediate geometric primitives to the samples [21, 22]. Algorithms of this category treat volumetric data as semitransparent material and create compresensive images, which show information about its interior, giving scientists better insight of the phenomenon represented by the data. It can be stated that given a 3D object, the goal of volume rendering algorithms is to produce an image that provides a compresensive, 3D representation of the object. Researchers have been trying to optimize currently known algorithms as well as look for new rendering algorithms that would bring down the rendering times without compromising the quality of the final images.

1.1.2 *Uses of volume rendering*

In volume rendering the interior information is not thrown away, thus enabling one to look at the 3D dataset as a whole. Weak or fuzzy surfaces can be displayed using volume rendering techniques. By this method there is no need to make a decision whether a surface is present or not [7, 24].

Volume rendering in scientific visualization has found extensive usage in a plethora of applications. Some of the common scientific and engineering areas are paleontology, archeology, computational fluid dynamics, geosciences, astrophysics, meteorology, chemistry, oil and gas exploration, aerospace enginering, mechanical enginering, non-

destructive testing, industrial and security applications, microscopy, biomedicine, medical imaging, surgical planning and simulations, and of course entertainment. Volume visualization is sometimes used to compare (numerical) results derived from real empirical experiments with results from simulations of the same experiments.

1.1.3 *Main steps in volume rendering*

Many steps are common to all the algorithms. Most of the algorithms have a subset of the steps mentioned below, usually in the given order. Sometimes it might be difficult to order steps 3 and 4 below. The steps are as follows:

1. Data acquisition: The first step in every volume rendering method is data acquisition. This is done by different methods that are discussed in the next section either by empirical measurement or by computer simulation.

2. Data conversion: Convert the data into a format that can be easily manipulated and worked with. The data may need to be scaled or process each slice so that it covers a good distribution of values and has high contrast in the values. Out-of-range data should be removed. It is often tried to filter out the noise but care must be taken so that valuable information is not excluded in the process. Of course, the same set of operations must be applied to all the data slices.

3. Data classification or applying a threshold: It means choosing a threshold value for a surface-fitting (SF) algorithm or color and opacity values for each possible range of data values for a direct volume rendering (DVR) algorithm (Section 1.1.5). This is a difficult task to perform as it requires some previous information and knowledge about the dataset. The threshold values also depend on the dataset and its type of modality.

4. Store, manipulate and display primitives: In this step the elements are mapped onto geometric or display primitives using some mapping operation. Here, the primitives are stored, manipulated, and if required, mixed with other primitives that have been defined externally. Shading and transformation to screen space is also done here and the primitives are displayed. This step has the maximum variation for different algorithms.

1.1.4 *Methods of data acquisition*

The 3D datasets can be either acquired by some scanning process or by simulations in super computers. Among the various acquisition methods in the medical field we mention magnetic resonance imaging (MRI), computed tomography (CT), positron emission tomography (PET) and ultrasound sonography of the area-of-interest. High-resolution microscopes like confocal laser-scanning microscopes and others are also used to acquire data [8, 17, 40]. The simulation of vector and scalar fields in industrial applications, time-varying ocean and weather models and computational fluid dynamics are a few other non-medical datasets.

> Finite element analysis or computational fluid dynamics programs are often used to simulate events from nature. If an event is too big, too small, too fast, or too slow to record in nature, then only the simulated event data volumes can be studied [8].

All the above types of datasets are considered similar in structure although they are generated by diverse methods.

1.1.5 *Classification of volume rendering methods*

Volume Rendering Algorithms can be classified into several categories based on various criteria as can be seen in Figure 1.1. The classification mentioned here is based on the literature survey done in [8]. They are as follows:

- Data Elements: The volume data is given as a scalar array of sampled points. There are basically two ways in which a volume rendering algorithm can treat this data: an array of volume elements (voxels) or an array of cells.

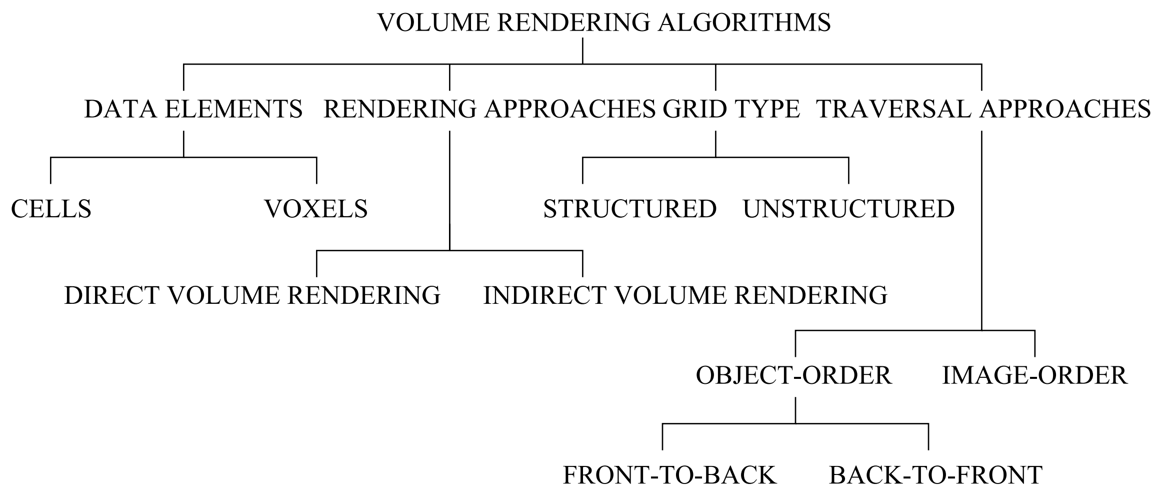

Figure 1.1 Classification of volume-rendering algorithms

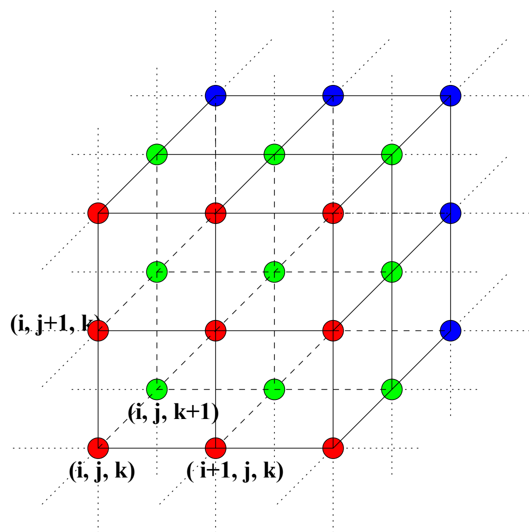

Figure 1.2 Cell representation of volume data

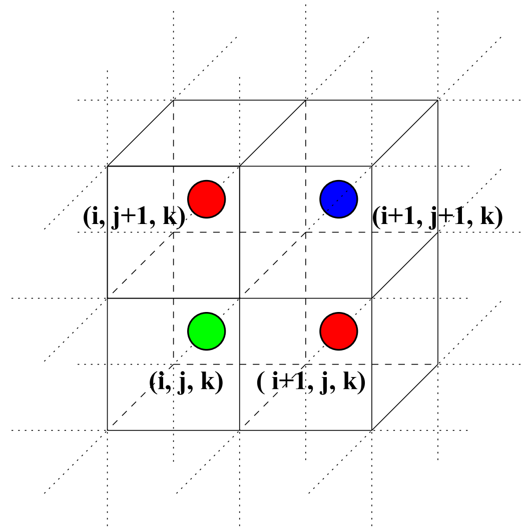

Figure 1.3 Voxel representation of volume data

1. Cell - Some algorithms consider the data points as the corners of a hexahedral cell as shown in Figure 1.2. This data is most often visualized as a lattice where each vertex of the grid is associated with a scalar value. Trilinear and tricubic interpolation between the values at the corners of the cell are commonly used to estimate the values inside the cell. It has been observed that the cell approach gives smoother images than the voxel approach.
2. Voxel - Here the data points are considered as the centers of regions of influence, for instance, the center of a regular hexahedral as in Figure 1.3. It may be visualized as if the small volume around a data point has the same value as the point. In some approaches this value is inversely proportional to the distance from the center of the hexahedral. Another small variation is the implemention of a filter function with each data point to represent the contribution of it in the region of influence. The voxel approach has the advantage that no assumptions are made about the behavior of data between grid points, i.e., only known data values are used for generating an image.

Grids and Lattices: Another classification is based on the type of grid format in which the processed data (scalar or vector field) is represented. Details of this classification are mentioned in Section 1.3.

1. Structured/Regular grids algorithms - They handle data that are distributed regularly on the vertices of a rectilinear grid. The grid is implicit and no structure is necessary to allow adjacency information to be retrieved.
2. Unstructured/Irregular grids algorithms - They handle data unevenly distributed in space with no implicit connectivity. These algorithms require auxiliary structures to compute adjacency information of the cells to be able to render the data correctly and efficiently.

Rendering Approaches: Volume visualization algorithms can also be divided into two categories, Direct Volume-Rendering (DVR) algorithms and Indirect Volume-Rendering (IVR) or Surface-fitting (SF) algorithms.

1. Direct Volume Rendering (DVR): In the DVR methods data elements are directly mapped onto the screen space without using any intermediate geometric primitive. The scalar values are directly rendered without any intermediate conversion step [6, 7, 31, 38, 42, 43]. The DVR approach gives very high-quality images. One disadvantage of using DVR methods is that the entire dataset must be traversed each time an image is rendered. Sometimes a low resolution pass or random sampling of the data is used. This creates low-quality images fast and can be used for parameter checking. The process of successively increasing the resolution and quality of a DVR image over time is called progressive refinement. It can be either an Image-order or an Object-order traversal method. Examples of DVR algorithms are ray-casting and splatting.
2. Indirect Volume Rendering (IVR) or Surface-Fitting (SF): IVR/SF algorithms are sometimes called feature-extraction or iso-surfacing. In this approach an intermediate representation of the data is created, which is rendered afterwards [31]. Intermediate representations are typically planar surface primitives that have constant-value contour surfaces in volumetric datasets. IVR/SF methods are typically faster than DVR methods since IVR/SF methods only traverse the volume once to extract surfaces. After extracting the surfaces, rendering hardware and other existing fast rendering methods can be used to quickly render the surface primitives each time the user changes a viewing or lighting parameter. There are a few disadvantages associated with this method. Typically, a huge number of surface primitives are generated for volumetric datasets. Also, changing the IVR/SF threshold value takes a lot of time because it needs to revisit every cell again to extract a new set of surface primitives. Contour-connecting, marching cubes, marching tetrahedra and dividing cubes are some examples of the IVR/SF approach.

Traversal Approaches: The final image is created in either of the following two ways: image-order traversal or object-order traversal

1. Object-order traversal: In this type of image generation all the elements of the volumetric dataset are traversed in an orderly fashion. For every element the projection and contribution to the pixels in the image plane is calculated. Two further sub-classifications are front-to-back and back-to-front order of traversal.
   (a) Front-to-back: This method is usually the faster among the two. The reason is that the elements in front may have already created a completely opaque image and the ones in the back do not need to be rendered anymore since they will not effect the final image anyhow. This reduces the number of elements that need to be traversed and rendered. This advantage does not hold good if the data is highly transparent, and in such cases both methods should take the same amount of time.
   (b) Back-to-front: An advantage of this method is that the user can see all the internal structures of the volume as the final image progressively builds up. Most structures might eventually get hidden but in some applications this method of image generation is preferred.
2. Image-order traversal: In this type of image generation, as the name suggests, the pixels in the image plane are usually traversed in scanline order. Sometimes pixels are also traversed in a random order.

Usually, any algorithm uses only one of the above mentioned traversal approaches, but there are some algorithms that use a combination of both types of traversal [8].

1.1.6 *Bottlenecks faced in volume rendering*

A great challenge in the field of volume rendering is to have a balanced trade-off between rendering speed and quality of the resulting images. A typical dataset can contain 256^3 (about 16.7 million) data elements or more. Treating such large datasets makes volume rendering computationally expensive and gives low frame rates. High quality images are a necessity for some applications like biomedicine and surgery. There is a lot

of ongoing research on optimizations on existing algorithms as well as for new rendering algorithms that would bring down the rendering times for accurate images.

Data classification is another difficult task that has to be carried out in volume visualization. Since getting images that clearly visualize the features of interest is only possible with a good transfer function, the transfer function specification is very important for the resulting images. It has been mentioned in earlier literature that transfer function specification is a difficult task and is application and dataset dependent.

There are datasets that have multiple values (scalar or vector) for every data point. Visualizing such a volume of data is more difficult for the above stated reasons as it requires a multi-dimensional transfer function.

1.2 Techniques Used for Making Results More Realistic

Computer-assisted biomedical imaging, pre-surgery planning, surgery and post-surgery diagnosis essentially needs volume rendering with correct transfer functions to convey accurate information of anatomic structures and pathology [2, 16, 18]. The data elements in the volume array needs to be mapped to visual parameters, like color and opacity, for the final image. Transfer functions are used for this purpose in volume rendering. Without transfer functions a volume rendered image would simply look like an opaque flat image. The power of volume rendering is the ability to specify a transparency for each voxel within the volume [19, 20, 34]. This results in images where you can look inside the objects, or patients, and see their internal structure [17]. Opacity transfer functions are

also used to pre-select the volume of interest. For instance, applying a transfer function that blends off everything except high-density materials in a given CT dataset would result in a final image in which only bone structures are visible. Other tissues are simply clipped off. Another issue in bio-medical volume visualization is that the density function often depends on the type of tissue. Hence, color transfer functions are used to color different types of tissues differently along with opacity [3, 40].

Every element in a volume potentially contributes to the color and opacity of the final image. Also important is the use of lighting for the final image as it adds an additional depth cue, features look sharper, and the human visual system responds better in the presence of light [49]. Care must be taken that over-use of light may introduce errors due to change in final colors, or some information may get occluded or misinterpreted by shadows. To get a good image, the transfer functions should be implemented carefully as slight changes in color and opacity values may have drastic impact on the final image. Since getting images that clearly visualize the features of interest is only possible with a good transfer function, transfer function specification is very important for understandibility of the final images. The fact that this understandibility is subjective to the field of application, and to the datasets being rendered, and that it is, to some extent, also subjective to the user, makes this task difficult.

1.3 Classification of Datasets

There are many types of grids, and many of the grid structures depend on the application. Many new types of grids have come into existance over the years. Here only the basic and most commonly used ones have been mentioned (Figure 1.4). Definitions and some nomenclatures are from [8].

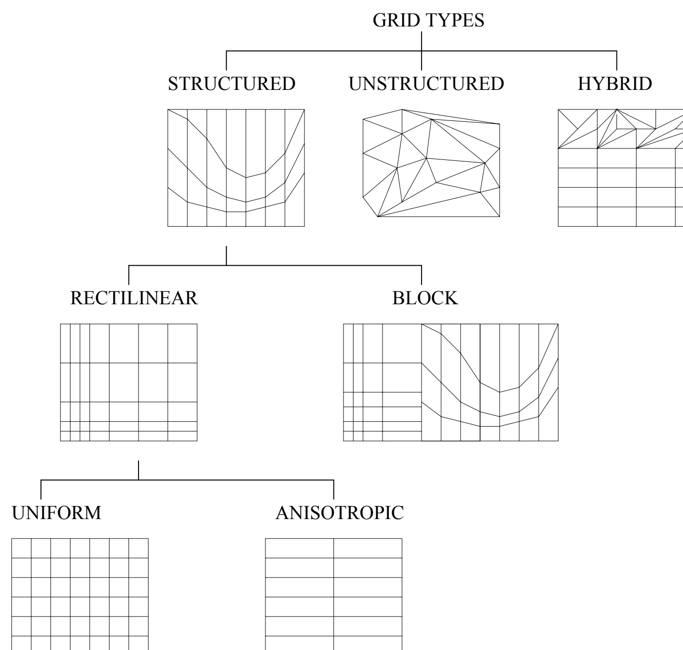

Figure 1.4 Classification of grid types

Structured: These are non-axis-aligned hexahedra (warped bricks) in general. Spherical and curvi-linear lattices are also examples of structured grids. Specific types are Rectilinear and Block.

1. Rectilinear: All the elements are identical axis-aligned rectangular prisms, but the elements are not cubes. Elements on a rectilinear grid are not necessarily identical but they are still hexahedra and axis-aligned. Two further subclassification are Uniform Rectilinear and Anisotropic Rectilinear (Regular).

   (a) Uniform (Cartesian grid/isotropic/cubic): All of the cells are identical axis-aligned cubes.
   (b) Anisotropic (Regular): All of the cells are identical axis-aligned rectangular bricks (not cubes).
2. Block: It is a collection of various structured grids sewn together to fill a space.

Unstructured: Here cells can be tetrahedra, hexahedra, prisms, pyramids, etc. There is no implicit connectivity. It is not necessary that the cells have planar faces. Tetrahedra is a popular type of cell because a tetrahedron has planar faces and any volume can be decomposed into tetrahedra.

Hybrid: A combination of any of the grids mentioned above to fill a volume space.

1.4 Sweep Paradigm

Processing geometric entities in an order determined by passing a sweep-line across a plane or a sweep-plane across 3D space is the Sweep paradigm. It is used in computational geometry as a standard algorithmic paradigm [36].

Till now the sweep paradigm has been used in some volume rendering algorithms applied to irregular grid datasets only. Girsten [14] was the first to use the concept in volume rendering. Yagel [53] and Silva [39] furthered the work on sweeping algorithms. The most recent work based on the sweep paradigm for volume rendering was the ZSweep algorithm [10].

1.5 Research Proposal

RZSWEEP is a new volume-rendering technique for uniform rectilinear datasets. The research work that is proposed here is that the RZSweep algorithm can be developed following the basic idea of the ZSweep presented by Farias et al. in [10]. The ZSweep

algorithm was designed for irregular grids. The new method, RZSweep, is designed for uniform, rectilinear grids. It would be based on the sweep paradigm (section 1.4), where a plane sweeps the vertices of the grid (implicit for regular datasets) in depth order, projecting all the implicit faces incident on each vertex. This algorithm would use the inherent properties of a rectilinear datasets. RZSweep would be an object-order, back-to-front, direct volume rendering, face projection algorithm for rectilinear datasets using the cell approach. It would be a single processor serial algorithm.

Only a volume rendering algorithm in itself does not produce comprehensible results. Various transfer functions need to be implemented in order to obtain images that clearly visualize the features of interest. Hence, it is also proposed that implementation of the following in RZSweep would make the results more realistic:

Lighting model

Color transfer functions

Opacity transfer functions

Finally, an image comparison would be done between RZSweep and a 3D texture-based method for volume rendering using standard image metrics like Euclidian and geometric differences.

The goal is to generate realistic volume-rendered results of uniform rectilinear datasets used in scientific, engineering, and medical fields.

CHAPTER II

LITERATURE REVIEW

This chapter outlines the most popular approaches of various volume rendering techniques for rectilinear datasets, the sweep-plane method and its implementation for volume rendering of irregular datasets and finally a review on implementations of transfer functions to improve understanding of the final image.

2.1 Volume Rendering Algorithms for Regular Datasets

Classification of volume rendering algorithms have been widely discussed in the literature in the past. Present day volume rendering algorithms for regular datasets can be divided into four main classes:

1. Ray-casting algorithms
2. Splatting algorithms
3. Shear-warp factorization algorithms
4. Texture mapping algorithms

A few distinguishing features of these algorithms are the traversal order of the entire data volume and projection methods of the data elements for the final image [21]. Pre-Direct and Post-Direct Volume Rendering Integral are also used to categorize volume

rendering approaches. Pre-Direct Rendering Integral is first changing the color and opacity of the data [24, 25]. In Post-Direct Rendering Integral, for every point the density and gradient attributes are computed [1, 17, 40, 41]. Another nomenclature introduced in [21] is based on "*loop ordering*". The ordering of the loops are based on the rendering approach and thus, distinguish one algorithm from the other. Space Leaping [52, 54] is the method of avoiding unrequired computations for empty space in datasets and is commonly used by most of the algorithms mentioned above to improve overall rendering process [6]. Every type of algorithm has its own share of performance advantages and disadvantages with respect to final result quality and speed. There is always a trade-off between quality and speed in the field of volume rendering and sometimes this trade-off may depend on the application requirement. It would be very difficult to review the volume rendering literature on classification in great detail here. Hence, this chapter would focus on outlining a general algorithm of every class mentioned above.

2.1.1 *Ray-casting algorithms*

There are many ray-casting implementations available today. The basic algorithm [24, 25, 38, 42, 43] is:

Shoot a ray into the data volume for each image pixel

Compute the color and opacity at regular intervals along the ray. This is usually done by trilinear interpolation with the values at eight closest neighbors.

The final color for every pixel is computed by compositing using a back-to-front linear interpolation method.

Trilinear interpolation of floating point numbers makes ray casting computationally intensive and time consuming. Many computer graphics techniques like adaptive early ray termination [21, 25, 49], space leaping by using octree decomposition [12, 29, 52, 54] and adaptive sampling [24, 25, 26] have been implemented to the basic algorithm to achieve speed without compromising on image quality. Adaptive early ray termination is to stop computation along a ray if the final color or opacity has reached a particular threshold for that pixel. It is useful in front-to-back ray traversals and simply means that all objects behind a pixel get occluded once the pixel becomes opaque. Empty space is skipped using octree decomposition. Adaptive sampling is used for reducing the computations by exploiting the homogeneous parts of the volume. The reader is advised to refer to [6, 21, 25, 52] for a discussion on optimizations on ray casting in great detail. Ray casting is a computationally expensive image-order backward projection algorithm but it produces high-quality images.

2.1.2 *Splatting algorithms*

Splatting typically achieved higher speed in volume rendering at the cost of reduced image quality. It is an approximate algorithm in which every data element contributes to the final image, and this contribution is called “footprint”. Several methods have been developed on this concept [7, 46, 47, 48, 50]. Splatting differs from ray casting in way the final image is generated. In this method the data elements are projected onto the image plane. This type of projection is sometimes called *“splatting”* [46]. This technique

was developed by Lee Westover at the University of North Carolina at Chapel Hill [48]. The efficiency of this algorithm depends on the chosen complexity of the footprint. Initial implementations projected all data elements but later on out-of-range values were not considered. The concept of early ray termination was adapted as early splat termination in which splatting was not carried out in regions of the image that had already become opaque [33]. Later, animation was possible after image-aligned splatting was developed by Mueller et al. [32]. Splatting is an object-order algorithm that can be implemented in either back-to-front or front-to-back order. Theoretically, splatting can produce results of the same quality as ray casting, but it is very difficult to compute the perfectly balanced filter weights for splatting. Westover [46, 47] mentions that splatting can be implemented to achieve either speed or quality but not both at the same time.

2.1.3 *Shear-warp factorization algorithms*

Lacroute and Levoy [21, 22] presented the Shear-warp factorization algorithm for volume rendering. The major disadvantage of this image-order method is that every ray has to travel through the entire dataset. This results in huge computation overheads. On the other hand, object-order methods cannot exploit the effective optimization technique of early ray termination. Shear-warp exploits both these advantages and is "based on a factorization of the viewing matrix into a 3D shear parallel to the slices of the volume data, a projection to form distorted intermediate image, and a 2D warp to produce the final image" [22]. It is a very fast method giving good quality images but the quality may reduce due to multiple

resampling [22]. “The second potential problem is that the shear-warp algorithm uses a 2D rather than a 3D reconstruction filter to resample the volume data” [22].

2.1.4 *Texture mapping algorithms*

Texture mapping is a hardware-dependent approach for volume rendering. Significant initial research and development was done by Cabral [4] and a lot of work of implementing lighting and shading has been done on this technique since then [5, 13, 30, 45]. There are two main methods of using texture mapping for volume rendering. The first method is to have 2D textures of the given slices and then blending these slices from back-to-front using transparency [7]. In this method, appropriate interpolation needs to be done to compensate for the material between the slices. The second method is more recent and uses a 3D texture mapping [15]. It has been possible only after achieving improved capabilities of implementing 3D textures in the hardware [35]. This method has the advantage that the entire 3D texture needs to be loaded into the texture memory only once and all the interpolations are done in the hardware. Rendering is fast but this method entirely depends on, and is limited to, available texture memory and swap memory for swapping textures.

2.2 Sweep Based Volume Rendering Algorithms

As mentioned before, the sweep paradigm has been implemented in computational geometry [36]. It is the technique of processing geometric entities in an order determined by passing a sweep-line across a plane or a sweep-plane across 3D space. When applied to

volume rendering [14], the sweep method has the advantage that it becomes a 2D sorting problem, and this reduces computational overheads [14, 39]. Spatial coherence is maintained, and connectivity between cells/elements does not need to be maintained [14, 39]. Although till now this approach has been implemented in volume rendering algorithms for irregular datasets only, it is relevant to briefly discuss this method, as the work presented here is an implementation of the sweep paradigm for volume rendering of regular datasets. Giertsen [14] was the first to use this method for volume rendering in which the viewing direction was the -direction and the sweeping plane was parallel to this viewing direction, that is parallel to the x -plane [9, 39]. It was the same approach in the Lazy Sweep Ray Casting Algorithm [39]. A modification in the sweep direction was put forth by Yagel et al. in [51, 53], where the sweep-plane was perpendicular to the -direction. ZSweep was the most recent work that was done using the plane sweep paradigm [9, 10, 11]. It is a fast and memory-efficient algorithm for volume rendering of irregular datasets. In the ZSweep algorithm also the sweep-plane was perpendicular to the -direction and parallel to the viewing plane. The sweep-plane passes over the entire dataset in the increasing order of , and all the faces of cells that the plane touches are projected onto the image plane [9, 10].

2.3 Review on Transfer Functions

The importance of transfer functions in scientific visualization has been well documented in scientific literature. Transfer functions are most commonly used for opacity, color, and emittance [27]. Various strategies have been developed for the creation of color

maps on the basis of data types, application dependencies, and user studies in [3, 37, 44]. Opacity transfer functions play an important role as they allow the data to be either seen or not seen in the final image. The Design Gallery [28] has a very comprehensible interface for transfer functions. Contour Spectrum [2] uses isosurfaces, and He et al. [16] use a genetic algorithm in their approach for generating transfer functions. All these works have been primarily in 1D transfer functions. Recent focus has been on implementing 2D and multi-dimensional transfer functions [19, 20]. Laidlaw [23] and Levoy [24] have been the pioneers in the use of 2D transfer functions. Semi-automatic generation of 1D and 2D transfer functions have been presented in [18, 34].

CHAPTER III

RZSWEEP

RZSWEEP is a new volume-rendering technique for uniform rectilinear datasets that is presented in this work.

There are two main parts of the research that is described here. The first part is implementing the sweeping plane paradigm in the direction [10] for volume rendering of uniform rectilinear datasets. It is a single processor serial algorithm that uses the inherent properties of a rectilinear datasets. Other aspects are that it is an object-order, back-to-front, direct volume rendering, face projection algorithm for rectilinear datasets using the cell approach.

Only a volume rendering algorithm in itself does not produce comprehensible results. Various transfer functions need to be implemented in order to obtain images that clearly visualize the features of interest. Hence, implementation of a lighting model, color and opacity transfer functions in RZSweep make the results more realistic.

Finally, an image comparison is done between RZSweep and a 3D texture-based method for volume rendering using standard image metrics like Euclidian and geometric differences. The volume-rendered image using 3D texture mapping method has been provided by [35].

The goal is to generate realistic volume-rendered results of uniform rectilinear datasets that are useful in scientific, engineering and medical fields.

3.1 Significance of RZSweep

As mentioned earlier, the sweeping-plane concept has also been applied to accomplish volume-rendering tasks. The basic intention is to localize the task. Thus, computations are performed on the element only when the sweep plane passes over it. There are several advantages of this approach:

1. The task at hand is reduced from a 3D to a 2D sorting problem.
2. The task is localized, thereby becoming computationally less complex and less expensive.
3. Spatial coherence of the data is maintained as this approach ensures orderly sweeping of the space.
4. There is no requirement to maintain information about the connectivity between cells/elements.

As mentioned in the introduction earlier, the sweep paradigm has been very successful for volume rendering algorithms applied to irregular grid datasets only. It is a novel idea to implement this approach for regular datasets also.

3.2 The Basic RZSweep Algorithm

RZSweep algorithm sweeps rectilinear data in depth order using an imaginary plane perpendicular to the viewing direction called the sweep plane, touching one vertex at a time. Points already touched by the plane sweep and projected are flagged as *swept*. When

the sweep plane touches a vertex, the algorithm projects all faces incident on the vertex, which are defined only by non-swept vertices, on to the screen. Now follows a detailed explanation of the algorithm.

The first concern is how to create a real representation for the implicit grid in which the rectilinear datasets are represented. A world coordinate system is built that is represented initially by its three unit vectors *u0*, *u1* and *u2* correponding to the *x*, *y* and *z* directions. Then, an affine transformation matrix containing all requested rotations and translations is created and applied to the world coordinate system's basis vectors. From this point on, any vertex *(i, j, k)* of the implicit grid can be represented in the world space by projecting (using a dot product) the vertex's implicit coordinates with each of the unit vectors of the world.

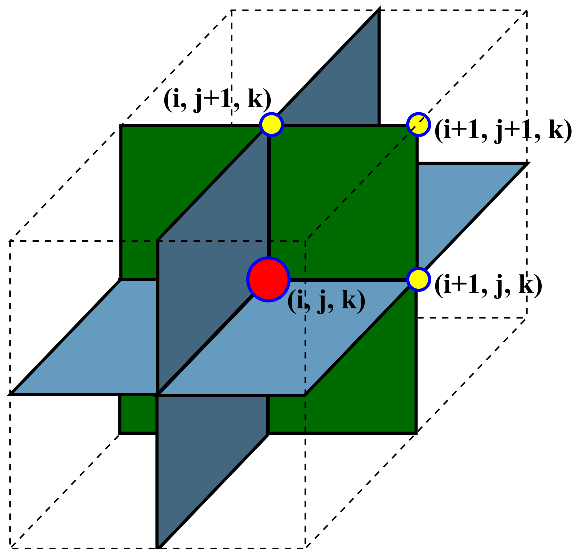

Figure 3.1 Twelve incident faces on an internal vertex

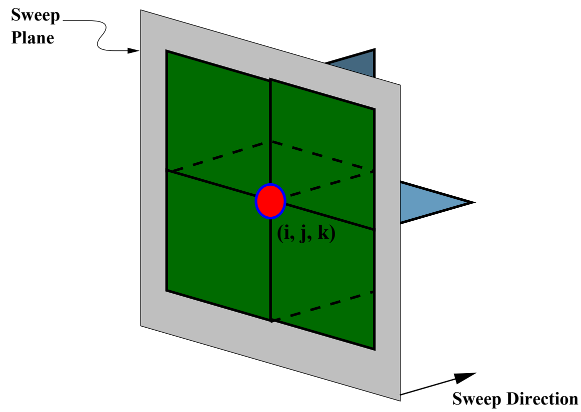

Figure 3.2 Faces that lie ahead of the sweep plane

In this preprocessing phase, the algorithm also determines the projectable faces out of all incident faces on a vertex. Every internal vertex has twelve faces incident on it, see Figure 3.1. Among those faces only the ones lying ahead of the sweep plane with respect to the sweeping direction must be projected, and are called *new faces* throughout the text. Compare Figure 3.1 with Figure 3.2.

The sweeping is performed by ordering the vertices by their -coordinate using a heap sort (from now on referred to simply as *heap*) and retrieving one by one in order. The sweeping process continues until the heap is empty. The algorithm starts by inserting only the nearest of the eight vertices of the bounding box of the data into the heap. The other vertices are inserted on-the-fly, minimizing the necessary size of the heap. Figure 3.3 shows sweep plane touching a vertex (current vertex). The two non-swept vertices would be sent to the heap and marked accordingly. This allows the sweep to continue. The two

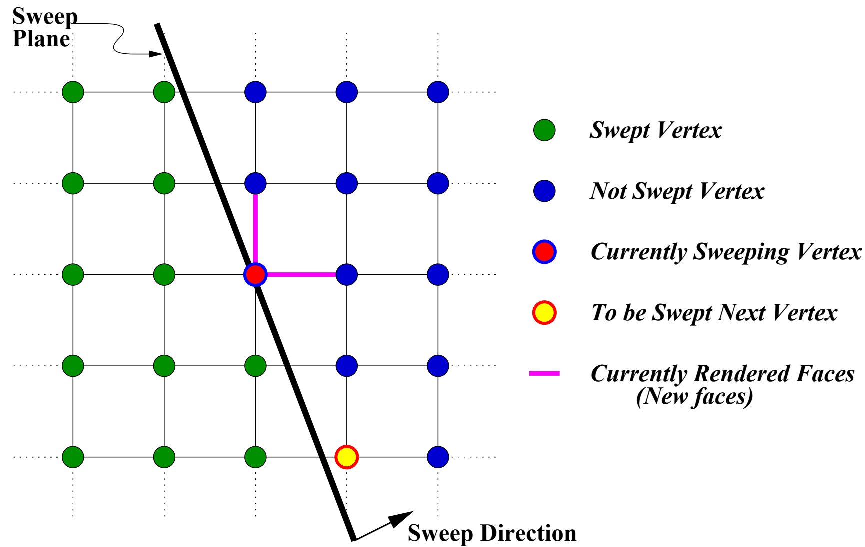

Figure 3.3 The schematic representation of RZSweep algorithm

new faces are incident on the current vertex and are lying ahead of the sweep plane. The algorithm goes into the main loop, which is explained below in pseudo code of RZSweep.

In order to avoid multiple insertions into the heap, the algorithm sends only those neighboring vertices to the heap that have not been sent before. Also, to avoid multiple projection of a face, it is checked if any other vertex that defines the face (except the current vertex) has already been projected, i.e. swept.

The on-the-fly classification is performed by sending only those vertices that are inside the required range of scalar values, to the heap. By flagging the vertices in the preprocessing step, the classification could be done efficiently.

Pseudo code of RZSweep

The outline of the basic RZSweep algorithm is shown here in the form of a pseudo code.

1. Read data from slices
2. Convert data into single binary file, in which each scalar value is of one byte
3. Compute the faces that are visible in the given orientation
4. Start rendering by inserting the closest (minimum value) corner data element of the new bounding volume, into the heap
5. While heap not empty {

    Retrieve next vertex from the heap *current vetex*

    Set it as *swept*

    Determine the neighboring vertices which define the *new faces* incident on the current vertex

    If and only if the *sent* flag is *not* set

    - Send the neighboring vertices to the heap
    - and set them as *sent*

    Project the *new faces* only if the *swept* flag of all the vertices of the faces are *not* set.

   }

3.3 Research Issues

Some of the research issues that were identified to require particular attention are:

Representation of the implicit grid of uniform rectilinear datasets in the real-world coordinate system. This was achieved by representing the (x, y, z) of the real-world coordinate system as three unit vectors $(u0, u1, u2)$ in our coordinate system.

Application of transformations to the data points. For this we applied one affine transformation matrix to the implicit data point and then projecting onto the world coordinate system.

Redundant and unnecessary multiple projection of faces should be avoided. For this, all the faces that need to be projected for a particular projection are pre-computed in the pre-processing.

The correct depth order must be ensured. Heap is used to sort the data points according to their depths, i.e. -values in the real world coordinate system. Heap sort is used since it has the best performance of $O(nlogn)$ for the worst case.

3.4 Implementation Details

The RZSweep was implemented completely in C++. Two versions of projection, one using the graphics hardware and a second one using software compositing, have been developed.

3.4.1 *Preprocessing and basic structures*

Besides the data itself, the algorithm requires memory space to keep the heap and flags to control the sweeping process. Since the size of the heap can be predicted a priori and is of the order of the largest diagonal plane of the data volume, it can be considered as one more slice of data, which is not significant. That comes from the fact that at any time only the points from a thin slab of the data ahead of the sweep plane will be in the heap. The worst case can occur for certain orientations of the dataset in which the sweep plane is parallel to the largest diagonal plane of the data volume. Care has been taken to use only bits for the flags so that memory is not wasted by unnecessary allocation.

Before starting the rendering process three steps are performed. First, the transformation matrix containing the desired orientation is computed and applied to the unit vectors

that will be used to transform the implicit data grid to the world coordinate system. Second, the data array is scanned, and vertices which lie within the desired threshold are flagged to speed up on-the-fly classification. The third step determines which faces incident on a vertex are projectable. A detailed description for this step is provided in the following paragraph.

Internal vertices have twelve faces incident on them, as can be observed in Figure 3.1. Consider that the data has dimensions Dim_x, Dim_y and Dim_z and the the internal vertex v has indices (i, j, k). By adding or subtracting 1 to or from the indices of v, we are able to determine all its twenty six neighboring vertices, which along with v will define the twelve implicit faces incident on it (for instance, the face (i, j, k), $(i + 1, j, k)$, $(i + 1, j + 1, k)$ and $(i, j + 1, k)$ in Figure 3.1). Projectable faces are the ones that lie ahead of the sweep plane with respect to the sweeping direction (see Figure 3.2). As shown in this figure, in the worst case, there will be eight faces to project. In the best case there will be only three. We consider the value of the coordinate of vertex v as reference. Then the projectable faces will be the ones whose all four vertices have coordinate values greater than or equal to this reference. A list of such faces is built, and this allows fast checking during the sweeping process.

3.4.2 *Sweep*

A heap sorting data structure was chosen to order the vertices in the sweeping process because of its $(n \cdot log(n))$ worst case complexity. Initially, only the nearest of the eight

corner vertices of the data volume are inserted into the heap. The algorithm obtains the next vertex from the heap, labels it as swept, and proceeds by computing the projectable faces using the list mentioned in Section 3.4.1. Among the vertices that define the projectable faces, the vertices that have not been marked as *sent* will be inserted into the heap and labeled as *sent*. The new faces are then projected based on the following criteria:

1. Out of the four vertices that define a face, the current vertex is the *only* swept vertex.
2. The scalar value of at least one of the four vertices must lie within the given threshold range.

3.4.3 *Projection*

Two types of projection have been implemented. In the hardware-dependent implementation, each face is sent to the graphics hardware which takes care of the rasterization and compositing. The software-dependent version is slower, as expected, but enables the parallelization of the code for shared and distributed-shared memory architectures.

The simplicity of the algorithm in projecting the faces enabled us to write the code such that creating a hardware-dependent or independent executable is only a matter of changing the type of final projection (hardware or software) of the faces.

3.4.3.1 Hardware

OpenGL is a standard application interface for graphics hardware. The hardware-dependent implementation sends each face that is supposed to be projected to the graphics pipeline using the OpenGL library. When a valid face with at least one vertex within the

user-defined threshold is computed, it is sent to OpenGL using the quad primitive. After this, the graphics hardware is responsible for rendering and rasterizing of the face. The code was tested on SGI's Irix and PC Linux. Gouraud shading is implemented as it gives a smooth shading and generates more photo-realistic images.

3.4.3.2 Software

The hardware independent implementation inherited the same lighting model as the one used on the original ZSweep algorithm [9, 10]. Instead of sending the faces to the graphics hardware, each face is scan converted. This creates intersections for all pixels belonging to the area of the screen on which the face is projected. Notice the scan conversion of all faces of a cubical cell will project two intersections on a given region of the screen. For pixels in this region, the lighting integral is applied adding the contribution from this cell to the final color of such pixels. After compositing between two intersections for a pixel, the algorithm will keep the intersection with greater value (if the sweep is performed toward the positive direction), which will be used when the next intersection is projected on the pixel, and so on.

3.4.4 *Optimizations*

Several optimizations were introduced and implemented in the algorithm that reduced memory requirement and the number of faces to be considered in the projection phase of the algorithm.

3.4.4.1 Projecting faces within threshold

This is the basic optimization done before the projection stage. A face consists of four vertices that define it. Only those faces are projected that have at least one out of four vertices within the threshold range. This optimization helps to avoid unnecessary projections of faces that are outside the given threshold range.

3.4.4.2 Volume reduction

In the preprocessing, we reduce the volume of the data. This is done by shrinking the bounding box towards the center of the dataset, until at least one vertex with a scalar value that lies within the threshold range is found. Defining the range of threshold for which relevant information is preserved varies for each dataset. Using their respective threshold range, all datasets had their original volume reduced. This optimization results in speeding up of the algorithm as the sweep process does not need to be performed over empty space and parts of the datasets that are outside the threshold range. Section 4.1 gives some numerical results of the speed-up. A lot of unwanted noise can be eliminated from the final result by choosing the threshold range appropriately as shown in the images of Figure 4.19 and Figure 4.20 in Section 4.2.

3.4.4.3 Faces reduction

As mentioned in Section 3.2, interior points have twelve faces incident on them, out of which only eight must be considered for projection. Out of these eight faces, certain

faces will not contribute significantly to the final image as they lie parallel to the viewing direction. Those faces are identified in the preprocessing step and eliminated from the list of projectable faces.

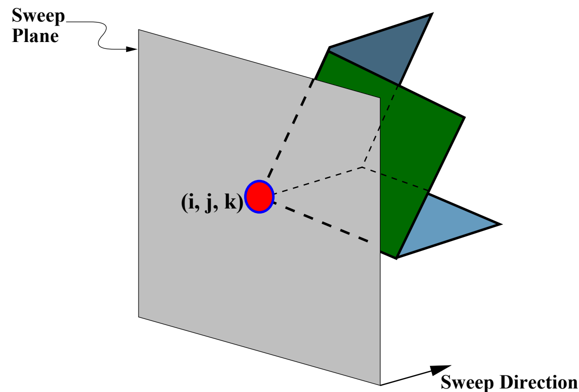

Figure 3.4 Best cases for face projection

It has been observed that in all orientations, a minimum of three and a maximum of four faces out of eight would contribute significantly. As shown in Figure 3.2, in the worst case there will be only four faces (faces shown in solid lines in Figure 3.2), and in the best case there will be three faces to be projected (see Figure 3.4). Viewing from any data corner, only three faces will be projected by each vertex.

3.4.4.4 Connected datasets

During the sweep process, once a data element is found within the given threshold range, on-the-fly classification is performed. Only those vertices out of the projectable faces that are inside the required range of scalar values are considered and sent into to the

heap. This further reduces the number of faces projected. By flagging the vertices in the preprocessing step, the classification could be done efficiently. As mentioned earlier, this optimization holds valid for connected datasets only. The efficiency brought about by this optimization can be seen in Section 4.1. A lot of disconnected objects outside the chosen threshold range (typically artifacts) in the dataset are also avoided by this optimization.

3.5 Slice Viewer

Most of the datasets come in the form of 2D slices where each data element is usually represented in 2 bytes (16 bits). First a converter has been developed that takes 2D slices as the input and converts them to a single binary dataset of unsigned bytes.

A slice viewer has been developed. This helps to see the contents of inner slices that make up the inner material of the dataset. Also this facilitates to compare the result of the new algorithm with a standard image editor. Figure 3.5 shows a slice of a MRI-Head dataset with the slice viewer. Figure 3.6 and Figure 3.7 show a slice of a CT-Skull and an Engine (an industrial, non-medical) dataset respectively using the RZSweep slice viewer.

3.6 Lighting, Color and Opacity Transfer Functions

Lighting, color and different opacity values bring realism to the final result. This section describes the approaches taken to implement lighting, color and opacity transfer functions. The results of these can be found in Sections 4.1 and 4.2.

Figure 3.5 MRI-Head dataset slice with RZSweep slice viewer

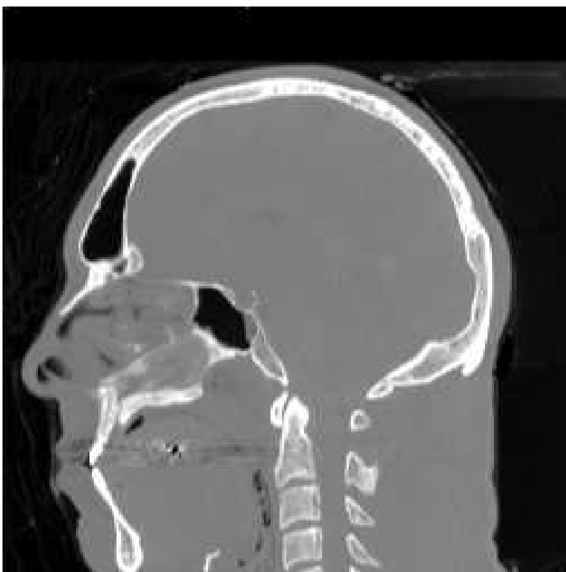

Figure 3.6 Slice of CT-Brain dataset using RZSweep slice viewer

Figure 3.7 Engine dataset slice with RZSweep slice viewer

3.6.1 *Lighting*

In order to implement lighting, it is necessary to have information about the normals of every vertex. There is no such information provided in a uniform rectilinear dataset. For this reason virtual normals are computed for every vertex by using first-order differentiation.

$$N = (N_x, N_y, N_z) \tag{3.1}$$

where

$$N_x = (x_{i+1} - x_i) + (x_{i-1} - x_i) \tag{3.2}$$

$$N_y = (y_{i+1} - y_i) + (y_{i-1} - y_i) \tag{3.3}$$

$$N_z = (\;_{i+1} - \;_i) + (z_{i-1} - \;_i) \tag{3.4}$$

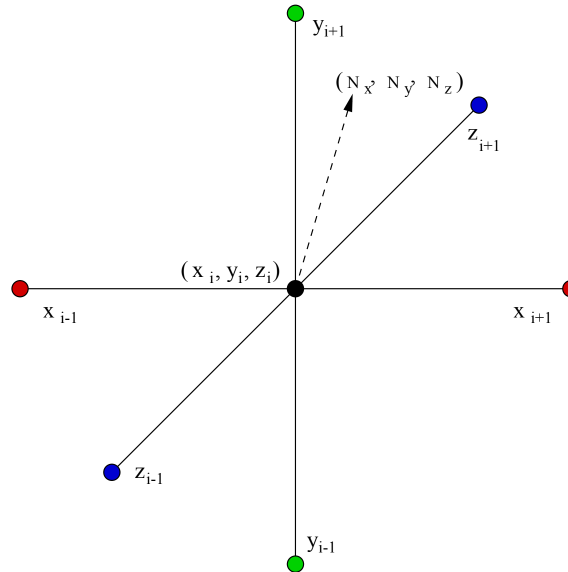

Figure 3.8 A vertex (x_i, y_i, z_i) with its immediate neighbors

$$|N| = \sqrt{N_x^2 + N_y^2 + N_z^2} \tag{3.5}$$

and then

$$N := \frac{N}{|N|} \tag{3.6}$$

From Figure 3.8, let N be the normal of a vertex whose components are (N_x, N_y, N_z) as shown in Equation (3.1). These components in the x, y and directions are computed using Equations (3.2), (3.3) and (3.4). Then Equation (3.5) is calculated in order to normalize the normal N using Equation (3.6).

All these computations are carried out for every vertex, as a preprocessing step and are passed to the graphics pipeline in the last stages of projection of faces. This helps to save rendering time, otherwise, computing normals at every vertex would make the algo-

rithm computionally very expensive. This normal information is provided to the graphics hardware for implementing OpenGL's Phong lighting model (Figure 4.13).

3.6.2 *Color*

The data elements in the datasets are just single scalar values. They have no color information. Color has to be put in explicitly, based on the scalar values. This is implemented using OpenGL's Gouraud shading and a lookup color table. The minimum and maximum of colors is decided with respect to the minimum and maximum threshold values that the user defined. This range of colors in the given RGB color model is linearly interpolated over 0 to 255 discrete intervals.

The colors are generated as a preprocessing step and stored as color lookup tables. Doing all the computations and creating the color lookup tables in the preprocessing step saves a significant amount of rendering time during the creation of the final image.

3.6.3 *Opacity*

Computation of opacity of every data element is part of the preprocessing step. It is computed by using different transfer functions. The results are within the range of $0 \ldots 1$ and are stored as a opacity lookup table with 256 discrete values corresponding to each scalar value. Some of the most common opacity transfer functions that have been implemented are:

1. **Linear** - opacity of every data element is linear and is directly proportional to the scalar value. Figure 3.9 is an example.

2. **Exponential** - opacity of every data element has some exponential relation with the scalar value. This exponential function is given by the user. One of the common exponential functions may look like Figure 3.10.

3. **Logarithmic** - opacity of every data element has some logarithmic relation with the scalar value. The user provides this logarithmic function. A usual logarithmic function may look like Figure 3.11.

4. **Box** - opacity of every data element is either transparent or a positive opacity value. This positive opacity value and the scalar range are given by the user. In Figure 3.12, two out of many types of box functions are shown that have been implemented.

5. **Triangle** - opacity of increases linearly till a certain value and then falls linearly, thus forming a triangle function. The peak value and the slope can be controlled by the user. Figure 3.13 shows two such cases.

6. **Bone** - If the data value is within the bone threshold range, then it has a positive opacity value otherwise it is transparent. The positive opacity value is a function of the scalar value that can be defined by the user. Also, the bone threshold range is also defined by the user. This type of transfer function is particulary useful only for bio-medical CT datasets. The opacity can be uniform for all bone data or can depend on the scalar values. Figure 3.14 shows a linearly increasing as well as constant opacity value function for the bone scalar values.

7. **Soft tissue** - If the data value is within the soft tissue threshold range, then it has a positive opacity value otherwise it is transparent. The positive opacity value is a function of the scalar value that can be defined by the user. The soft tissue threshold range is also defined by the user. This type of transfer function is particulary useful only for bio-medical CT datasets. Figure 3.15 shows a linearly increasing and a constant opacity for the range of scalar data values for soft tissues.

8. **Bone and Soft tissue** - If the data value is within the soft tissue threshold range, then it has a positive opacity value that is a function of the scalar value. If the data value is within the bone threshold range, then it has a positive opacity value that is a different function of the scalar value. Both these functions and the ranges can be defined by the user. This type of transfer function is particulary useful for bio-medical CT datasets. There are many functions possible. Figure 3.16 shows two of such cases, and combinations of the two functions is also possible. See Figure 4.15 for the final image.

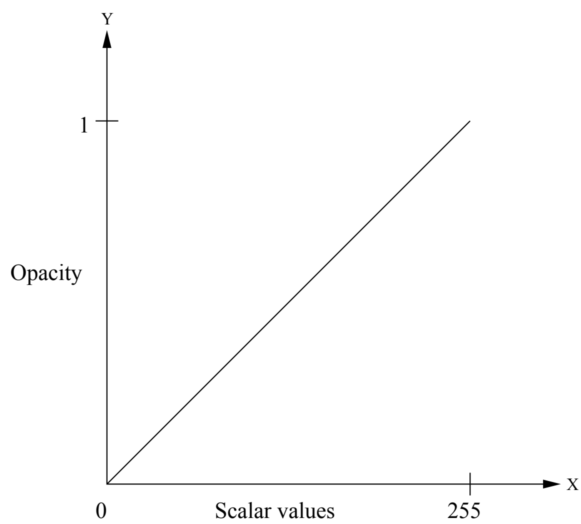

Figure 3.9 Linear transfer function for opacity

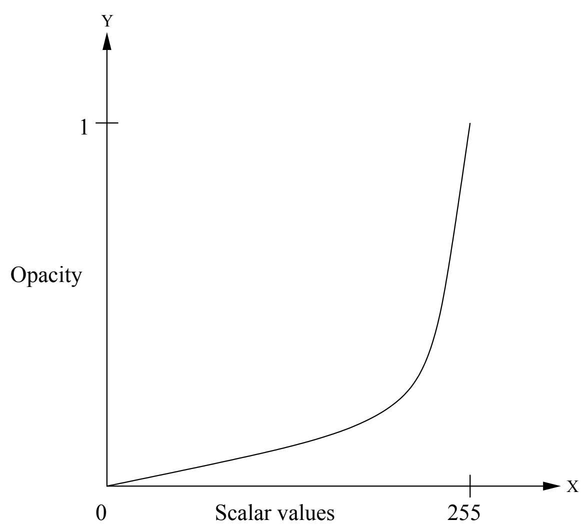

Figure 3.10 Exponential transfer function for opacity

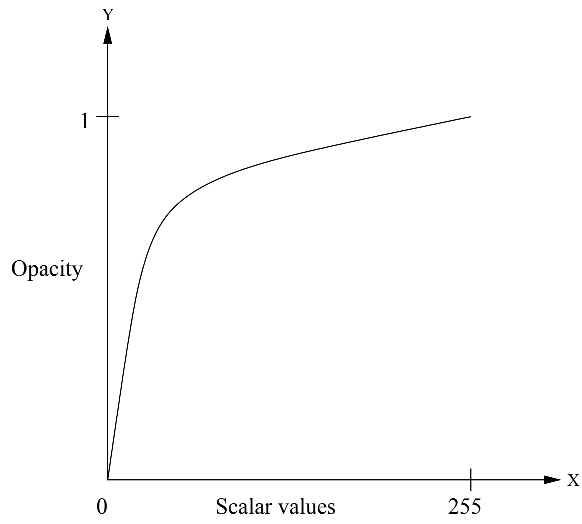

Figure 3.11 Logarithmic transfer function for opacity

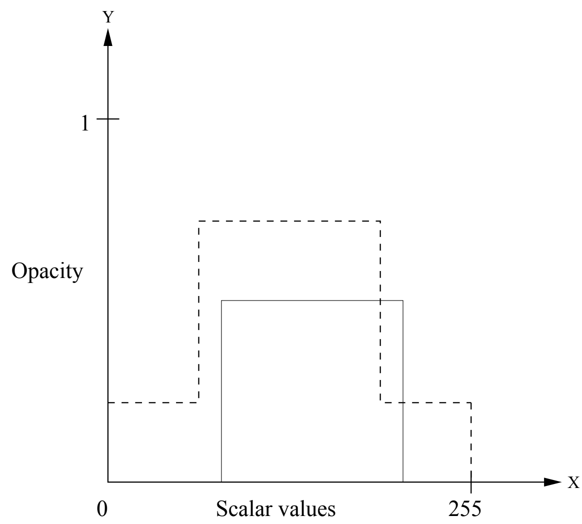

Figure 3.12 Box transfer function for opacity

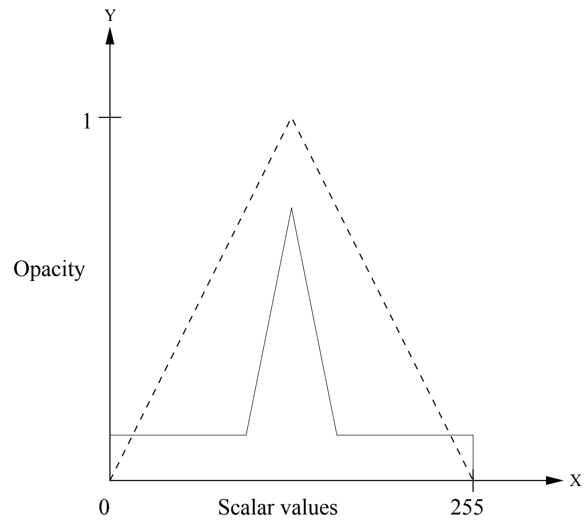

Figure 3.13 Triangle transfer function for opacity

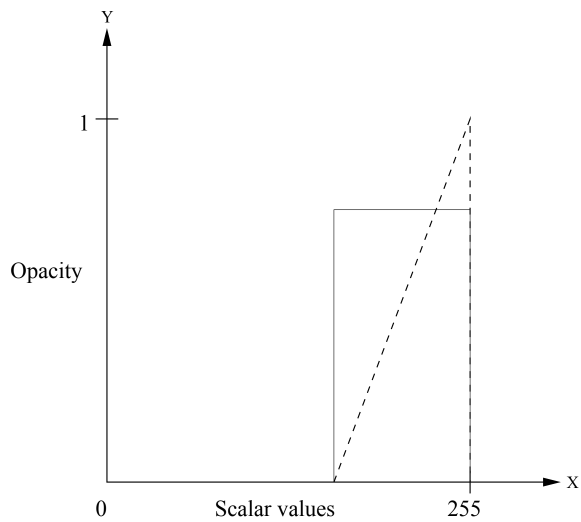

Figure 3.14 Opacity transfer function for bone

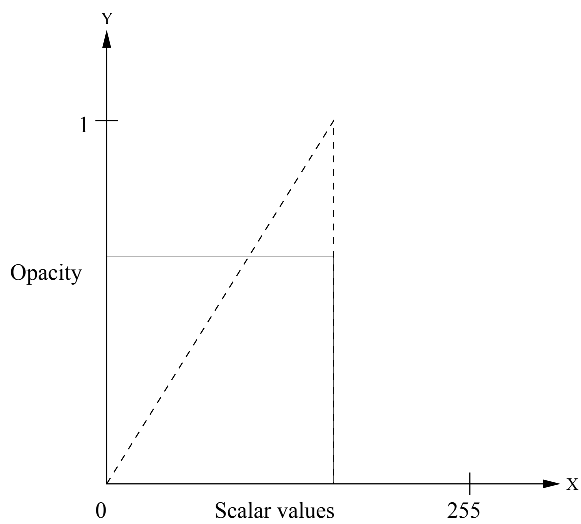

Figure 3.15 Opacity transfer function for soft tissues

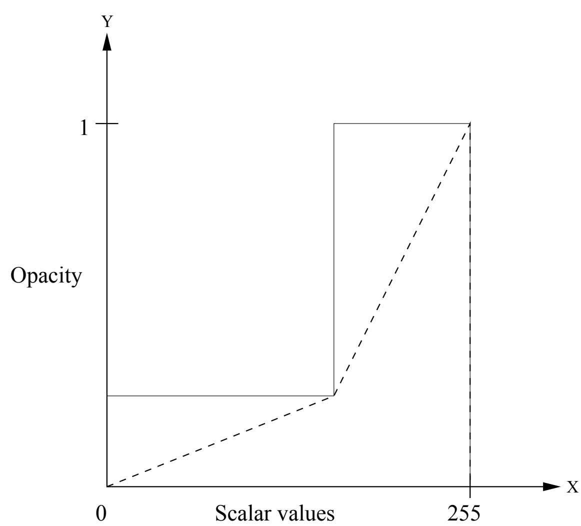

Figure 3.16 Opacity transfer function for bone and soft tissue

CHAPTER IV

RESULTS

There are many uniform rectilinear datsets available in the research community. Out of these, eight have been used to report all the readings and results for the RZSweep algorithm, and two have been considered for lighting, color and opacity transfer functions. These datasets were chosen because they were diverse and cover a wide range of different sizes.

- Timings and readings of RZSweep, before and after all the optimizations. Also, some readings after implementation of lighting, color and opacity transfer functions.
- Images of different datasets that are produced by RZSweep and also images after implementation of lighting, color and opacity transfer functions.
- Results of image comparison with 3D texture method for volume rendering developed by [35].

All readings presented in this chapter have been taken at least five times and the average time has been reported. Care has been taken such that the readings are disk I/O insensitive by having a local copy all the reported datasets, thus avoiding network delays. The computational time cannot be seperated from the rendering time due to the nature of the algorithm, because one face is rendered as its neighboring faces are being computed. The readings reported in this chapter depend on the threshold values mentioned in Table 4.1 and have zero degree rotation of the volume, unless specified otherwise. All timings are

for final images of a size of 256^2 unless specified otherwise. They have been taken on an SGI, the configuration details of which can be found in the Section 4.4. Table 4.1 gives all the details of different datasets that remain constant throughout this chapter.

Table 4.1 Dataset details

Datasets	Sizes	Threshold Range	Opacity	No. of points rendered
CT-Brain	$512 \times 256 \times 512$	165 – 255	0.05	2, 599, 041
CT-Skull	$256 \times 256 \times 256$	40 – 255	0. 0	1, 171, 011
Engine	$256 \times 256 \times 128$	26 – 255	0.10	1, 449, 051
Foot	$256 \times 256 \times 256$	75 – 255	0.08	52, 621
Fuel	$64 \times 64 \times 64$	1 – 255	0.10	15, 566
Lobster	01× 24 × 56	26 – 255	0.05	12, 513
MRI-Head	$256 \times 256 \times 111$	0 – 255	0.03	1, 6 9, 213
Statue-Leg	41 × 41 × 9	50 – 255	0.07	440, 038

4.1 RZSweep Timings

The hardware and software rendering timings are shown in Table 4.2 for four datasets. Table 4.3 gives the hardware rendering timings of the other four datasets. Both, Table 4.2 and Table 4.3 show the final, optimized timing of RZSweep.

Table 4.4 gives the hardware and software rendering times of RZSweep for different image sizes. These timings were taken only for the Engine dataset. Note that the hardware rendering time does not change significantly with the increase in image size but the software rendering times increase with increase in image sizes. This shows that the hardware assisted RZSweep has little or no rendering time degradation, while the software algorithm

slows down significantly for larger images. This almost constant scalability is one of the great advantages of our approach.

Table 4.5 gives the preprocessing times for different datasets as mentioned in Section 3.4.1.

Table 4.6 gives the basic RZSweep times of different datasets without any optimizations. It is interesting to note the difference with the optimized times as given in Table 4.2 and Table 4.3.

Compared to Table 4.6, the rendering times have been reduced significantly after implementing the volume reduction. This is shown in Table 4.7 along with the number of data points that are reduced by this optimization. The speed up is due to the fact that there is a much smaller number of points to render after this optimization as explained in Section 3.4.4.2.

Table 4.8 shows the difference in the times and the number of faces projected after implementing the face reduction from eight to four projectable faces mentioned in Section 3.4.4.3.

Table 4.9 shows further reduction of the number of faces that are projected after the optimization of connected datasets. Refer to Section 3.4.4.4.

The results of implementing color, opacity and lighting model seperately as well as all together, to the optimized RZSweep algorithm is given in Table 4.10. The readings show that there is very little increase in the rendering time when compared with those in

Table 4.2 and Table 4.3. The implementation is explained in Section 3.6 and the results are shown in Figure 4.16 and Figure 4.17.

Table 4.2 Hardware and software rendering times of RZSweep

Datasets	Hardware (sec.)	Software (sec.)
CT-Brain	10.69	69. 5
Fuel	0.07	0.82
Lobster	1. 4	9.56
MRI-Head	5. 8	47.06

Table 4.3 Hardware rendering times of RZSweep

Datasets	Time (sec.)
CT-Skull	.75
Engine	5. 7
Foot	.09
Statue-Leg	1.59

4.2 RZSweep Images

Resulting images from the optimized RZSweep algorithm using the graphics hardware for rendering are shown in the images in Figure 4.3, Figure 4.4, Figure 4.5, Figure 4.6, Figure 4.7, Figure 4.8, Figure 4.9, and Figure 4.10. All these have a uniform opacity function for all scalar values. The image in Figure 4.6 has a threshold range of $61 - 255$

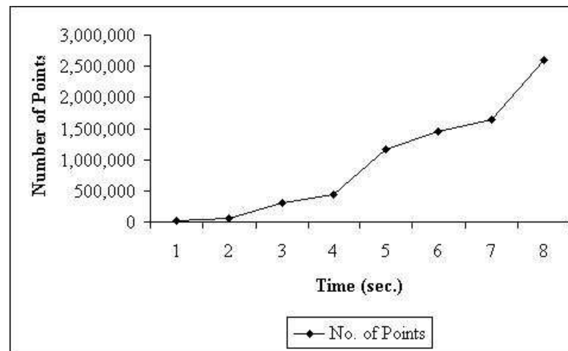

Figure 4.1 Graph showing rendering time for increasing number of data elements

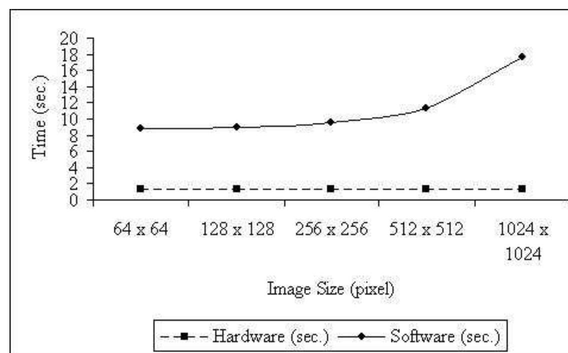

Figure 4.2 Graph showing the rendering times for different image sizes of Engine dataset

Table 4.4 Rendering times of RZSweep for different image sizes of Engine dataset

Image Sizes	Hardware (sec.)	Software (sec.)
64^2	1.34	8.78
128^2	1.34	8.97
256^2	1.34	9.56
512^2	1.34	11.36
1024^2	1.34	17.64

Table 4.5 Preprocessing times of RZSweep for different datasets

Datasets	Time (sec.)
CT-Brain	10.01
CT-Skull	4.21
Engine	1.42
Foot	.62
Fuel	0.03
Lobster	1.15
MRI-Head	1.46
Statue-Leg	2.08

Table 4.6 RZSweep timing without any optimizations

Datasets	Unoptimized Time (sec.)
CT-Brain	422.43
CT-Skull	86.43
Engine	46.2
Foot	85.36
Fuel	1.01
Lobster	33.45
MRI-Head	41.42
Statue-Leg	66.45

Table 4.7 RZSweep timing with volume reduction

Datasets	Reduced size	No. of points reduced	% reduced	Time (sec.)
CT-Brain	$391 \times 224 \times 262$	$44,161,856$	65.81	137.44
CT-Skull	$256 \times 256 \times 256$	0	0.00	86.43
Engine	$146 \times 199 \times 108$	$5,250,776$	62.59	16.70
Foot	$214 \times 238 \times 2\ \ 9$	$4,604,468$	27.44	65.29
Fuel	$6\ \ \times 32 \times 32$	$197,632$	75. 9	0.27
Lobster	$261 \times 298 \times 50$	$1,572,444$	28.79	19.66
MRI-Head	$202 \times 189 \times 109$	$,113,094$	42.79	21.67
Statue-Leg	$41 \times 217 \times 93$	$,932,412$	6.36	35.45

Table 4.8 RZSweep with eight and four projected faces

Datasets	8 faces		4 faces	
	No. of faces	Time (sec.)	No. of faces	Time (sec.)
CT-Brain	7, 88 , 286	137.44	2, 657, 742	115.17
CT-Skull	4, 359, 95	86.43	1, 46 , 695	75.95
Engine	4, 292, 913	16.70	1, 449, 622	13.32
Foot	2, 716, 848	65.29	892, 594	56.97
Fuel	47, 436	0.27	15, 711	0.13
Lobster	1, 184, 985	19.66	390, 321	17.21
MRI-Head	5, 06 , 452	21.6	1, 662, 535	16.95
Statue-Leg	1, 46, 826	35.45	443, 084	31.58

Table 4.9 RZSweep with connected data optimization

Datasets	No. of faces projected
CT-Brain	2, 599, 041
CT-Skull	1, 171, 011
Engine	1, 449, 051
Foot	526, 211
Fuel	15, 566
Lobster	312, 513
MRI-Head	1, 639, 213
Statue-Leg	440, 038

Table 4.10 RZSweep after implementation of light, color and opacity

Datasets	Color	Opacity	Light	All
CT-Brain	10.96	11.41	12.28	13.15
Engine	5.71	5.83	6.05	6.2

and image in Figure 4.9 has a threshold range of $10-255$. The image shown in Figure 4.10 has a threshold range of $0-255$ and a uniform opacity of 0.1. All other datasets have the same threshold range and uniform opacity as mentioned in Table 4.1.

Figure 4.3 and Figure 4.11 have the same opacity but different threshold range. The image in Figure 4.3 shows the bone and the other shows soft tissues like flesh and skin with uniform opacity. Figure 4.11 has a threshold range of $100-255$. This shows the effect of different threshold ranges.

The image in Figure 4.12 has the same threshold range as the one in Figure 4.3 but has a different opacity value of 0.005. The image in Figure 4.12 is more transparent due to a smaller opacity value. Here the opacity is uniform for all scalar values.

Figure 4.13 and Figure 4.14 show the effect of the implementation of a lighting model as mentioned in Section 3.6.1. Lighting adds realism to the final image, and this is evident when compared with Figure 4.3 and Figure 4.5.

Figure 4.15, Figure 4.16, and Figure 4.17 show the final images after implementation of a lighting model, color tables and opacity transfer functions, as mentioned in Section 3.6. Figure 4.15 uses the *Bone and Soft tissue* opacity transfer function. Figure 4.17 uses a *Triangle* opacity transfer function. Both these types of transfer functions are mentioned in Section 3.6.3.

Figure 4.3 CT-Brain dataset

Figure 4.4 CT-Skull dataset with 90 deg rotation about the axis

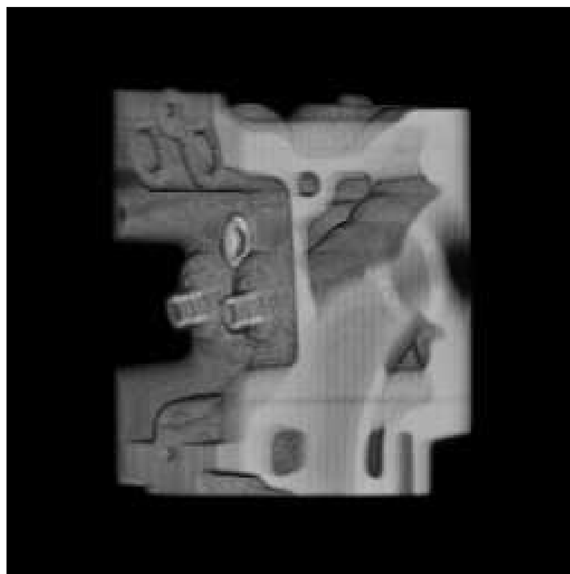

Figure 4.5 Engine dataset with $0\,\mathrm{deg}$ rotation about the Y axis

Figure 4.6 Foot dataset with $90\,\mathrm{deg}$ rotation about the Y axis

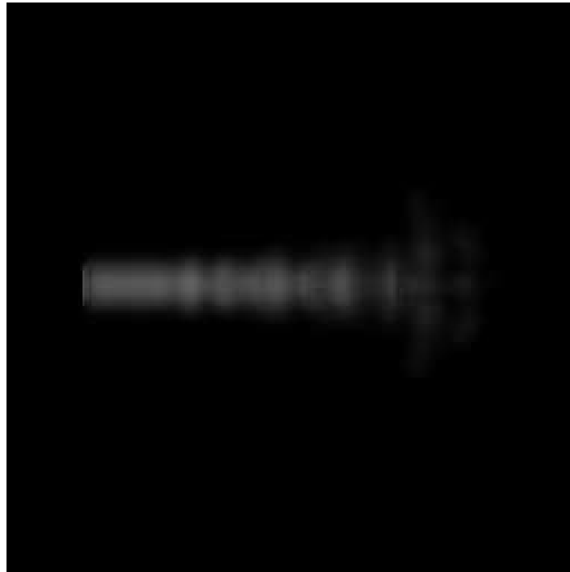

Figure 4.7 Fuel dataset

Figure 4.8 Lobster dataset

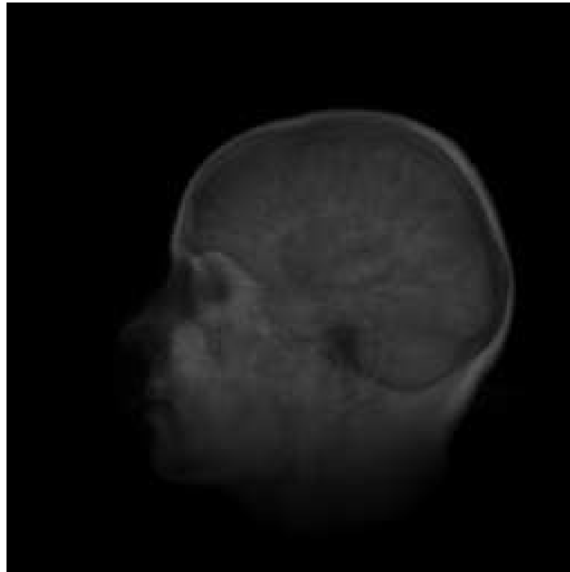

Figure 4.9 MRI-Head dataset

Figure 4.10 Statue-Leg dataset with $180\deg$ rotation about the Y axis

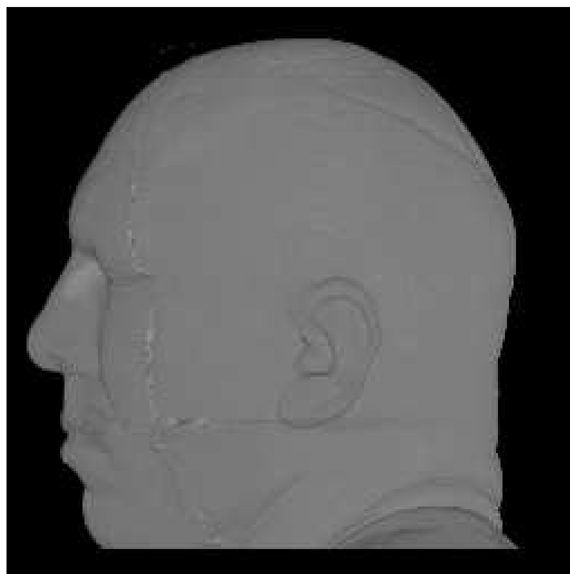

Figure 4.11 CT-Brain dataset with threshold range of $100 - 255$

Figure 4.12 CT-Brain dataset with uniform opacity of 0.005

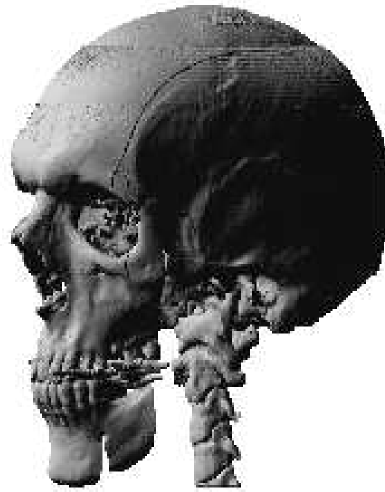

Figure 4.13 CT-Brain dataset with lighting

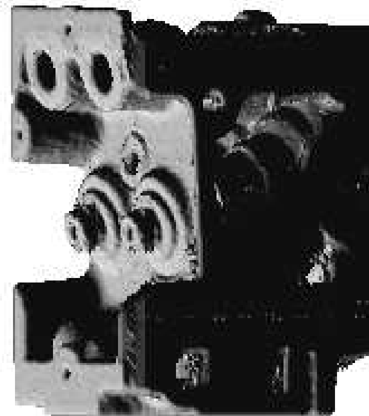

Figure 4.14 Engine dataset after implementation of a lighting model

Figure 4.15 CT-Brain dataset with threshold range of $100 - 255$

Figure 4.16 CT-Brain dataset after implementing light, color and opacity

Figure 4.17 Engine dataset with lighting, color and opacity

4.3 RZSweep Image Comparison

In this section, images generated by RZSweep are compared to images rendered with a 3D texture-based method for volume rendering developed by [35]. The CT-Brain dataset is used for this purpose. It has been tried to generate the two images under similar projection, opacity transfer function, rotation and threshold range of $0 - 255$. Since the dataset is a single scalar value for every data element and no color transfer function has been used, all images are in gray scale. If different lighting models were used, this would result in a lot of image differences. Hence, lighting has not been incorporated to generate the results. The 3D texture method uses 256 slices of data and 512 texture planes (Nyquist Theorem) [35].

1	2	1
2	4	2
1	2	1

1	1	2	1	1
1	2	3	2	1
2	3	4	3	2
1	2	3	2	1
1	1	2	1	1

Figure 4.18 Weight matrix of $\times$ and 5×5 pixels

Every pixel can be compared individually between two images. But sometimes if every pixel is slightly different and the images do not line up perfectly, it would make sense to compare larger areas instead of individual pixels. Therefore, two types of weighted averaging have also been done to reduce the image details and to compare primarily low-

frequency components in the image. $\times$ and 5×5 pixel areas have been considered and weights have been assigned to the neighboring pixels. These weights of the neighboring pixels are shown in Figure 4.18 for averaging $\times$ pixels and 5×5 pixels. The following is the nomenclature of the two images that are frequently used during discussion of image comparison.

1. Image(1) is generated by RZSweep as shown in Figure 4.19. Image(1)$_{3\times3}$ is after averaging $\times$ pixels of Image(1) (see Figure 4.25), and Image(1)$_{5\times5}$ is after averaging 5×5 pixels of Image(1) (see Figure 4.31).
2. Image(2) is generated by 3D texture mapping method as shown in Figure 4.20. Image(2)$_{3\times3}$ is after averaging $\times$ pixels of Image(2) (see Figure 4.26), and Image(2)$_{5\times5}$ is after averaging 5×5 pixels of Image(2) (see Figure 4.32).

The following are the four types of differences reported here for each pixel and average of $\times$ and 5×5 pixels. The list of images for these differences can be found in Table 4.11.

$$p = \ Image(1) - Image(2) \tag{4.1}$$

$$p = Image(1) - Image(2) \tag{4.2}$$

$$q = Image(1) - Image(2), \qquad p = \begin{cases} q, & q \geq 0 \\ 0, & q < 0 \end{cases} \tag{4.3}$$

$$q = Image(2) - Image(1), \qquad p = \begin{cases} q, & q \geq 0 \\ 0, & q < 0 \end{cases} \tag{4.4}$$

Absolute difference (Equation 4.1): All numerical differences are positive values.

Signed difference (Equation 4.2): Differences are either positive or negative in nature. The maximum negative difference is rendered black, and the maximum positive difference is white. When there is no difference, the pixel is gray.

Image(1) - Image(2) (Equation 4.3): If the difference is less than zero, i.e. the difference is negative, then it is treated as zero (black). This image shows those pixel values of Image(1) that are greater than the corresponding pixels of Image(2).

Image(2) - Image(1) (Equation 4.4): This image shows those pixel values of Image(2) that are greater than the corresponding pixels of Image(1).

In order to obtain a quantitative measure for the overall difference between the two images, we define two metrics: Euclidian and geometric difference.

$$\frac{\sum_{i=0}^{i=N-1}(P_i \quad Q_i)}{N} \tag{4.5}$$

Euclidian difference is the sum of all pixel differences between two images divided by the total number of pixels. It is given by Equation (4.5) where P_i is a pixel of Image(1), and the corresponding pixel of Image(2) is given by Q_i. Euclidian difference is applied to all the four types of differences mentioned above for per-pixel, $\times$ and 5×5 pixels. All these are reported in Table 4.12. Euclidian difference considers regular and linear distances and hence, does not provide emphasis of any kind.

$$\frac{\sqrt{\sum_{i=0}^{i=N-1}(P_i \quad Q_i)^2}}{N} \tag{4.6}$$

Geometric difference is the square root of the sum of all squares of the differences between corresponding pixels of two images divided by the total number of pixels. It is given by Equation (4.6) where P_i is pixel of Image(1), and the corresponding pixel of

Image(2) is given by Q_i. Geometric difference is applied to all the four types of differences mentioned above for per-pixel, $\times$ and 5×5 pixels. All these are reported in Table 4.13. The absolute and signed geometric difference values are the same since the square of the difference always gives a positive value, irrespective of the sign. Geometric difference emphasizes on outliers.

Higher order metrics like cubic etc. lack any kind of geometric intuition, and hence have not been implemented.

Comparing these images, it is seen that both methods generate very similar images. From images that show either Image(1)−Image(2) or Image(2)−Image(1), it can be seen that pixels of Image(2) (image generated by 3D texture method) only have a little higher gray-scale values. For quantitative values, see Table 4.12 and Table 4.13. For this reason images in Figure 4.23, Figure 4.29 and Figure 4.35 are almost black. Also, figures that show the absolute difference and Image(2)−Image(1) are almost similar because of the above reason.

Image in Figure 4.37 is generated using the RZSweep algorithm with a threshold range of $100 - 255$. All other aspects such as orthographic projection, opacity transfer functions, gray-scale data values and rotation angles, have been kept the same as in Figure 4.19 and Figure 4.20. It can be seen that the image in Figure 4.37 has a lot of unnecessary noise eliminated without compromising the image quality. This shows that quality of the final image improves after implementation of the various optimization and accleration techniques mentioned in Section 3.4.4 to the basic RZSweep algorithm.

Table 4.11 Difference image list

Comparison types	1 × 1 Pixels	× Pixels	5 × 5 Pixels
Absolute Difference	Figure 4.21	Figure 4.27	Figure 4.33
Signed Difference	Figure 4.22	Figure 4.28	Figure 4.34
Image1 - Image2	Figure 4.23	Figure 4.29	Figure 4.35
Image2 - Image1	Figure 4.24	Figure 4.30	Figure 4.36

Table 4.12 Euclidian difference between RZSweep and 3D texture-based rendering

Euclidian	1 × 1 Pixels	× Pixels	5 × 5 Pixels
Absolute Difference	4.43	4.11	.92
Signed Difference	− .74	− 66	− 54
Image1 - Image2	0.34	0.22	0.18
Image2 - Image1	4.08	.88	.7

Table 4.13 Geometric difference between RZSweep and 3D texture-based rendering

Geometric	1 × 1 Pixels	× Pixels	5 × 5 Pixels
Absolute Difference	0.02	0.07	0.11
Signed Difference	0.02	0.07	0.11
Image1 - Image2	0.006	0.01	0.01
Image2 - Image1	0.02	0.07	0.11

Figure 4.19 Image(1): Image generated by RZSweep

Figure 4.20 Image(2): Image generated by 3D texture-based rendering method

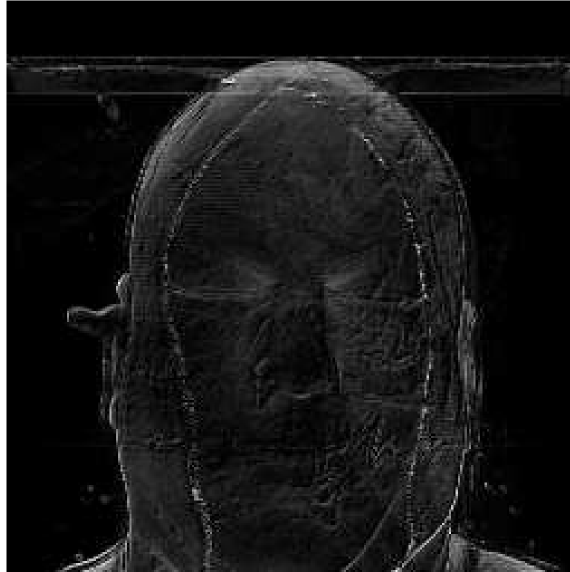

Figure 4.21 Absolute difference image between Image(1) and Image(2)

Figure 4.22 Signed difference image between Image(1) and Image(2)

Figure 4.23 Image(1) − Image(2)

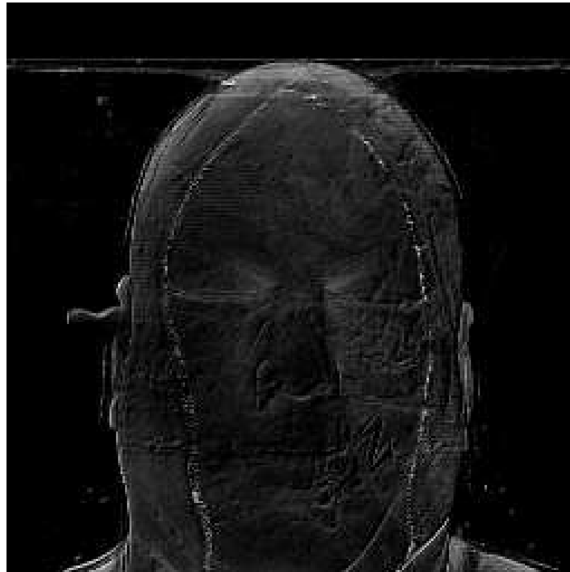

Figure 4.24 Image(2) − Image(1)

Figure 4.25 $\text{Image}(1)_{3\times3}$: Image after averaging $\times$ pixels of RZSweep

Figure 4.26 $\text{Image}(2)_{3\times3}$: Image after averaging $\times$ pixels of 3D texture-based rendering

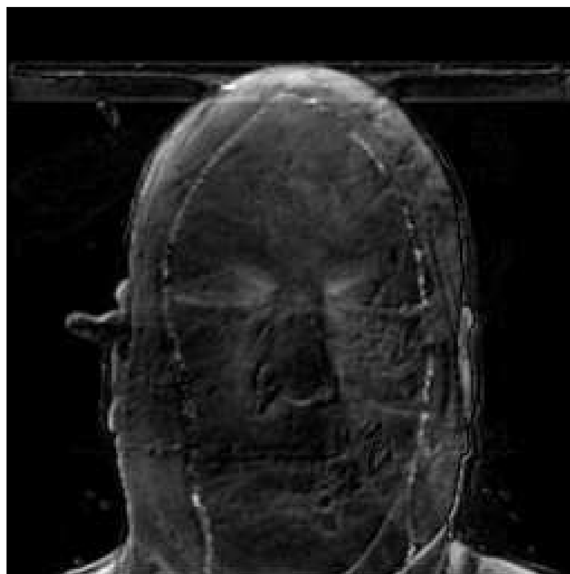

Figure 4.27 Absolute difference image between $\text{Image}(1)_{3\times3}$ and $\text{Image}(2)_{3\times3}$

Figure 4.28 Signed difference image between $\text{Image}(1)_{3\times3}$ and $\text{Image}(2)_{3\times3}$

Figure 4.29 $\text{Image}(1)_{3\times3} - \text{Image}(2)_{3\times3}$

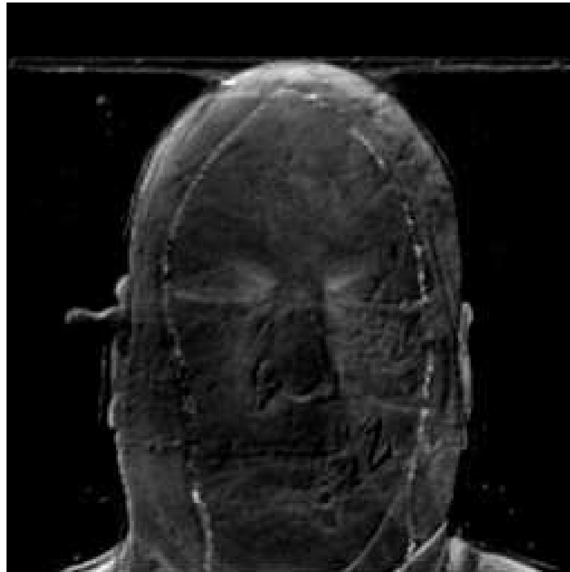

Figure 4.30 $\text{Image}(2)_{3\times3} - \text{Image}(1)_{3\times3}$

Figure 4.31 Image$(1)_{5\times5}$: Image after averaging 5×5 pixels of RZSweep

Figure 4.32 Image$(2)_{5\times5}$: Image after averaging 5×5 pixels of 3D texture-based rendering

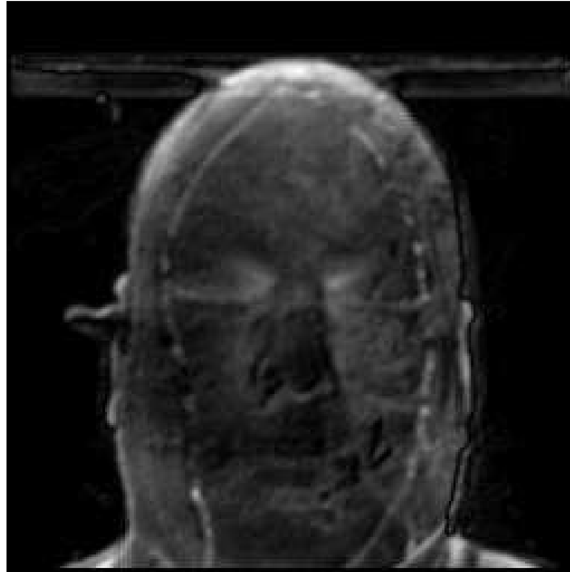

Figure 4.33 Absolute difference image between Image(1)$_{5\times5}$ and Image(2)$_{5\times5}$

Figure 4.34 Signed difference image between $\text{Image}(1)_{5\times5}$ and $\text{Image}(2)_{5\times5}$

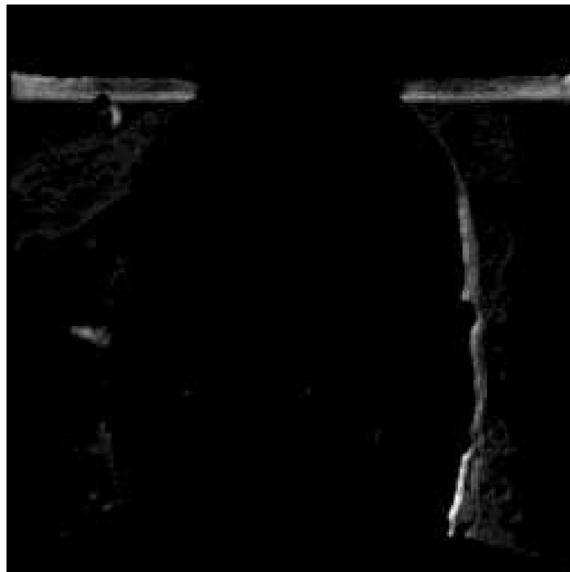

Figure 4.35 $\text{Image}(1)_{5\times5} - \text{Image}(2)_{5\times5}$

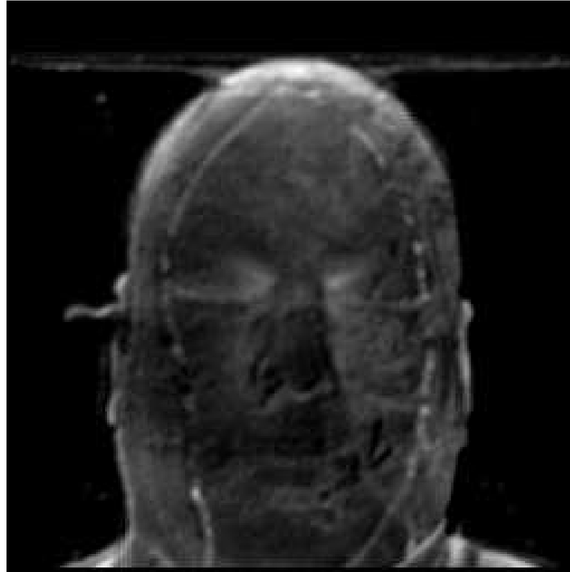

Figure 4.36 Image$(2)_{5\times5}$ $-$ Image$(1)_{5\times5}$

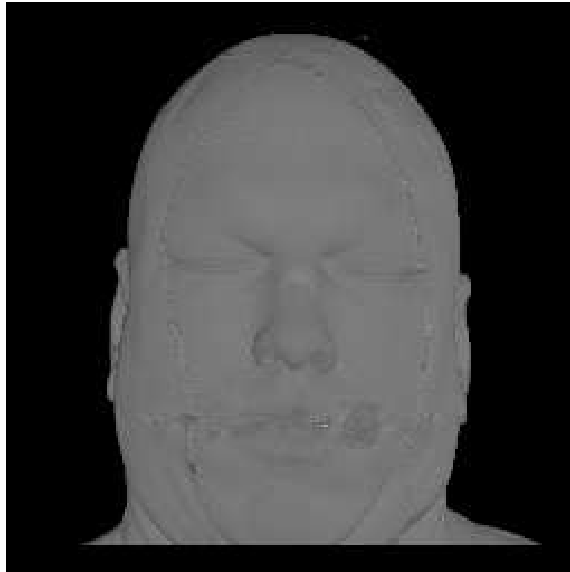

Figure 4.37 Clean image generated by RZSweep with a threshold range of $100 - 255$

4.4 SGI Machine's System Hardware

Mentioned here are the system details of the SGI machine on which all the timings reported in this document were taken. The *hinv* command was used. This command displays the contents of the system hardware inventory table. This table is created each time the system is booted and contains entries describing various pieces of hardware in the system. The items in the table include main memory size, cache sizes, floating point unit, and disk drives. Without arguments, the hinv command displays a one line description of each entry in the table, as shown below.

```
4 400 MHZ IP27 Processors
CPU: MIPS R12000 Processor Chip Revision: 3.5
FPU: MIPS R12010 Floating Point Chip Revision: 3.5
Main memory size: 4096 Mbytes
Instruction cache size: 32 Kbytes
Data cache size: 32 Kbytes
Secondary unified instruction/data cache size: 8 Mbytes
Integral SCSI controller 2: Version QL1040B (rev. 2), single ended
Integral SCSI controller 3: Version QL1040B (rev. 2), differential
Integral SCSI controller 4: Version QL1040B (rev. 2), differential
Integral SCSI controller 5: Version QL1040B (rev. 2), differential
Integral SCSI controller 1: Version QL1040B (rev. 2), single ended
Integral SCSI controller 0: Version QL1040B (rev. 2), single ended
Disk drive: unit 1 on SCSI controller 0
CDROM: unit 6 on SCSI controller 0
IOC3 serial port: tty1
IOC3 serial port: tty2
IOC3 serial port: tty3
IOC3 serial port: tty4
IOC3 parallel port: plp1
Graphics board: InfiniteReality3
Integral Fast Ethernet: ef0, version 1, module 1, slot io1, pci 2
ATM PCA-200E OC-3: module 1, xio_slot 2, pci_slot 0, unit 0
Iris Audio Processor: version RAD revision 7.0, number 1
Origin MSCSI board, module 1 slot 7: Revision 4
```

Origin BASEIO board, module 1 slot 1: Revision 4
Origin PCI XIO board, module 1 slot 2: Revision 4
IOC3 external interrupts: 1

CHAPTER V

CONCLUSIONS

5.1 Contributions

The work presented here is a new volume-rendering technique for uniform rectilinear datasets called RZSweep. RZSweep is the first attempt to incorporate the capabilities of the sweep paradigm to render volumetric rectilinear data. The algorithm is based on a virtual plane sweeping through an entire dataset in the -direction, creating the volume-rendered image in the process. The uniqueness of the algorithm is that it exploits the inherent properties of a rectilinear dataset to obtain the information of the neighboring vertices. This saves precious memory space and decreases the space complexity of the algorithm. RZSweep employs several optimization techniques to increase the rendering speed, decrease the memory requirement and get high quality results in a reasonable amount of time. RZSweep utilizes the graphics pipeline for achieving high rendering speed. In most volume- rendering algorithms it is seen that the rendering time increases with an increase in the image size. But the hardware assisted RZSweep algorithm has little or no rendering time degradation for different image sizes, and this would be very significant for rendering larger images. Results of the RZSweep algorithm with and without the different optimizations are presented in Section 4.1 and Section 4.2.

RZSweep is memory efficient because besides the data itself, the only other data structure that requires memory is the heap. As discussed in Section 3.4.1, the heap takes negligible memory compared to the entire dataset. RZSweep is a face projection algorithm. After identifying the faces incident on a vertex, there is only a small range of neighboring vertices that needs to be considered. This gives the algorithm the ability to perform on-the-fly segmentation and also eliminate non-relevant data from the process at the same time.

The explicit colors associated with the scalar values are implemented using different color lookup tables. Various standard and specific opacity transfer functions have also been implemented. The Phong lighting model and OpenGL's smooth Gouraud shading has been used to bring realism to the final images. Section 4.1 reports the timings of all these, and it is seen that the rendering time does not increase significantly after implementation of color, opacity functions and lighting. Results of these can be found in Section 4.2.

In Section 4.3 an image comparison is done between the results of RZSweep and a 3D texture-based rendering algorithm. The two methods generate very similar images. Euclidian and geometric differences image metrics give quantitative differences. Comparative images are also given for per-pixel and weighted average of $\times$ and 5×5 pixel areas.

5.2 Further Research

A graphical user interface (GUI) needs to be developed for RZSweep. This algorithm, in its current stage, takes in all the user-defined inputs in various commandline arguments.

One-dimensional color and opacity transfer functions have been used here. There has been recent research in the area of implementing two dimensional and multidimensional transfer functions for direct volume rendering (DVR) algorithms, as mentioned in Section 2.3. Since RZSweep is also a DVR algorithm, such multidimensional transfer functions also can be implemented on it.

The comparison of RZSweep's performance on the SGI with that of PC, under different conditions, would be performed in the immediate future. Based on these performance comparisons, certain architecture-dependent optimizations might be possible. This would also open up the area of exploiting the capabilities of todays dedicated high performance graphics cards for RZSweep.

Parallelization of a serial algorithm to improve its performance has been a common practice in the research community. RZSweep can be parallelized in the software version as well as in the hardware version.

REFERENCES

[1] R. Avila, T. He, L. Hong, A. Kaufman, H. Pfister, C. Silva, L. Sobierajski, and S. Wang, "Volvis: A Diversified Volume Visualization System," *Proceedings of IEEE Visualization 1994*, Washington, DC, October 17-21 1994, pp. 31–38.

[2] C. L. Bajaj, V. Pascucci, and D. R. Schikore, "The Contour Spectrum," *Proceedings of IEEE Visualization 1997*, Phoenix, Arizona, October 19-24 1997, pp. 167–173.

[3] L. D. Bergman, B. E. Rogowitz, and L. A. Treinish, "A Rule-based Tool for Assisting Colormap Selection," *Proceedings of IEEE Visualization 1995*, Atlanta, Georgia, October 29 - November 03 1995, pp. 118–125.

[4] B. Cabral, N. Cam, and J. Foran, "Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware," *Proceedings of the 1994 Symposium on Volume Visualization*, Tysons Corner, Virginia, United States, October 17-18 1994, pp. 91–98.

[5] F. Dachille, K. Kreeger, B. Chen, I. Bitter, and A. Kaufman, "High-quality Volume Rendering Using Texture Mapping Hardware," *Proceedings of the 1998 EUROGRAPHICS/SIGGRAPH workshop on Graphics hardware*, Lisbon, Portugal, August 31-September 01 1998, pp. 69–76.

[6] J. Danskin and P. Hanrahan, "Fast Algorithms for Volume Ray Tracing," *Proceedings of 1992 Workshop on Volume Visualization*, Boston, Massachusetts, October 19-20 1992, pp. 91–98.

[7] R. Drebin, L. Carpenter, and P. Hanrahan, "Volume Rendering," *Computer Graphics (Proceedings SIGGRAPH '88)*, vol. 22, no. 4, August 1988, pp. 65–74.

[8] T. T. Elvins, "A Survey of Algorithms for Volume Visualization," *Computer Graphics*, vol. 26, no. 3, October 2000, pp. 194–201.

[9] R. Farias, *Efficient Rendering of Volumetric Irregular Grids Data*, doctoral dissertation, State University of New York at Stony Brook, Department of Applied Mathematics and Statistics, June 2001.

[10] R. Farias, J. S. B. Mitchell, and C. T. Silva, "ZSweep: An Efficient and Exact Projection Algorithm for Unstructured Volume Rendering," *Proceedings of ACM/IEEE Symposium on Volume Visualization 2000*, Salt Lake City, Utah, October 9-10 2000, pp. 91–99.

[11] R. Farias and C. Silva, “Parallelizing the ZSweep algorithm for Distributed-Shared Memory Architectures,” *Proceedings of IEEE/EG International Workshop on Volume Graphics 2001*, Stony Brook, New York, June 21-22 2001.

[12] I. Gargantini, T. R. S. Walsh, and O. L. Wu, “Displaying a Voxel-based Object via Linear Octtrees,” *Proceedings of SPIE 626*, 1986, pp. 460–466.

[13] A. V. Gelder and K. Kim, “Direct Volume Rendering With Shading via Three-dimensional Textures,” *Proceedings of the 1996 Symposium on Volume Visualization*, San Francisco, California, United States, October 28-29 1996, pp. 23–30.

[14] C. Giersten, “Volume Visualization of Sparse Irregular Meshes,” *IEEE Computer Graphics and Applications*, vol. 12, no. 2, March 1992, pp. 40–48.

[15] P. Haeberli and M. Segal, “Texture Mapping as a Fundamental Drawing Primitive,” *Proceedings of the Fourth Eurographics Workshop on Rendering*, Paris, France, June 1993, pp. 259–266.

[16] T. He, L. Hong, A. Kaufman, and H. Pfister, “Generation of Transfer Functions with Stochastic Search Techniques,” *Proceedings of IEEE Visualization 1996*, San Francisco, CA, October 27 - November 1 1996, pp. 227–234.

[17] K. H. Hoehne, B. Pflesser, A. Pommert, M. Riemer, T. Schiemann, R. Schubert, and U. Tiede, “A ‘Virtual Body’ Model for Surgical Education and Rehearsal,” *Computer*, vol. 29, no. 1, January 1996, pp. 25–31.

[18] G. Kindlmann and J. W. Durkin, “Semi-Automatic Generation of Transfer Functions for Direct Volume Rendering,” *Proceedings of IEEE Symposium On Volume Visualization 1998*, Research Triangle Park, North Carolina, October 19-20 1998, pp. 79–86.

[19] J. Kniss, G. Kindlmann, and C. Hansen, “Interactive Volume Rendering Using Multi-Dimensional Transfer Functions and Direct Manipulation Widgets,” *Proceedings of IEEE Visualization 2001*, San Diego, California, October 21-26 2001, pp. 255–262.

[20] J. Kniss, G. Kindlmann, and C. Hansen, “Multidimensional Transfer Functions for Interactive Volume Rendering,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 8, no. 3, July-September 2002, pp. 270–285.

[21] P. G. Lacroute, *Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation*, doctoral dissertation, Stanford University, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, CA 94305-4055, September 1995.

[22] P. G. Lacroute and M. Levoy, “Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation,” *Proceedings of SIGGRAPH ’94*, Orlando, Florida, July 1994, pp. 451–458.

[23] D. H. Laidlaw, *Geometric Model Extraction from Magnetic Resonance Volume Data*, doctoral dissertation, California Institute of Technology, May 1995.

[24] M. Levoy, “Display of Surfaces from Volume Data,” *IEEE Computer Graphics and Applications*, vol. 8, no. 3, May 1988, pp. 29–37.

[25] M. Levoy, “Efficient Ray Tracing of Volume Data,” *ACM Transactions on Graphics*, vol. 9, no. 3, July 1990, pp. 245–261.

[26] M. Levoy, “Volume Rendering by Adaptive Refinement,” *The Visual Computer*, vol. 6, no. 1, February 1990, pp. 2–7.

[27] B. Lichtenbelt, R. Crane, and S. Naqvi, *Introduction to Volume Rendering*, chapter 4, Prentice Hall, Upper Saddle River, New Jersey, 1998, pp. 87–102.

[28] J. Marks, B. Andalman, P. Beardsley, W. Freeman, S. Gibson, J. Hodgins, T. Kang, B. Mirtich, H. Pfister, W. Ruml, and K. Ryall, “Design Galleries: A General Approach to Setting Parameters for Computer Graphics and Animation,” *ACM Computer Graphics (Proceedings of SIGGRAPH ’97)*, August 1997, pp. 389–400.

[29] D. Meagher, “Geometric Modeling Using Octree Encoding,” *Computer Graphics and Image Processing*, vol. 19, no. 2, June 1982, pp. 129–147.

[30] M. Meissner, U. Hoffmann, and W. Strasser, “Enabling Classification and Shading for 3D Texture Mapping based Volume Rendering using OpenGL and Extensions,” *Proceedings of the IEEE Visualization ’99*, San Francisco, California, October 24-29 1999, pp. 207–214.

[31] M. Meissner, J. Huang, D. Bartz, K. Mueller, and R. Crawfis, “A Practical Evaluation of Popular Volume Rendering Algorithms,” *Proceedings IEEE Symposium on Volume Visualization 2000*, Salt Lake City, Utah, October 9-10 2000, pp. 81–90.

[32] K. Mueller and R. Crawfis, “Eliminating Popping Artifacts in Sheet Buffer-based Splatting,” *Proceedings of IEEE Visualization ’98*, Research Triangle Park, North Carolina, October 18-23 1998, pp. 239–245.

[33] K. Mueller, N. Shareef, J. Huang, and R. Crawfis, “High-Quality Splatting on Rectilinear Grids With Efficient Culling of Occluded Voxels,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 5, no. 2, April-June 1999, pp. 116–134.

[34] H. Pfister, C. Bajaj, W. Schroeder, and G. Kindlann, "The Transfer Function Bake-Off," *Proceedings of IEEE Visualization 2000*, Salt Lake City, Utah, October 18-23 2000, pp. 523–526.

[35] P. Pinnamaneni, *Wavelet Based Volume Rendering System*, master's thesis, Mississippi State University, Department of Electrical and Computer Engineering, Mississippi State University, Mississippi State, MS 39762, September 2002.

[36] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, New York, 1985.

[37] P. Rheingans, "Task-Based Color Scale Design," *Proceedings of Applied Image and Pattern Recognition '99, SPIE*, October 1999, pp. 35–43.

[38] P. Sabella, "A Rendering Algorithm for Visualizing 3D Scalar Fields," *Proceedings SIGGRAPH '88*, 1988, pp. 51–58.

[39] C. Silva and J. Mitchell, "The Lazy Sweep Ray Casting Algorithm for Rendering Irregular Grids," *IEEE Transactions on Visualization and Computer Graphics*, vol. 3, no. 2, April-June 1997, pp. 142–157.

[40] U. Tiede, K. H. Hoehne, M. Bomans, A. Pommert, M. Riemer, and G. Wiebecke, "Investigation of Medical 3D Rendering Algorithms," *IEEE Computer Graphics and Applications*, vol. 10, no. 2, March 1990, pp. 41–53.

[41] U. Tiede, T. Schiemann, and K. H. Hoehne, "High Quality Rendering of Attributed Volume Data," *Proceedings of IEEE Visualization 1998*, Research Triangle Park, North Carolina, October 18-23 1998, pp. 255–262.

[42] H. Tuy and L. Tuy, "Direct 2D Display of 3D Objects," *IEEE Computer Graphics and Applications*, vol. 4, no. 10, November 1984, pp. 29–33.

[43] C. Upson and M. Keeler, "V Buffer: Visible Volume Rendering," *Proceedings SIGGRAPH '88*, 1988, pp. 59–64.

[44] C. Ware, "Color Sequences for Univariate maps: Theory, Experiments, and Principles," *IEEE Computer Graphics and Applications*, vol. 8, no. 5, September 1988, pp. 41–49.

[45] R. Westerman and T. Ertl, "Efficiently Using Graphics Hardware in Volume Rendering Applications," *Proceedings of SIGGRAPH '98*, Orlando, Florida, July 24-29 1998, pp. 169–177.

[46] L. Westover, "Interactive Volume Rendering," *Proceedings of the Chapel Hill Workshop on Volume Visualization*, C. Upson, ed., Department of Computer Science, University of North Carolina, Chapel Hill, NC, May 1989, pp. 9–16.

[47] L. Westover, "Footprint Evaluation for Volume Rendering," *Computer Graphics (Proceedings of SIGGRAPH '90)*, vol. 24, no. 4, August 1990, pp. 367–376.

[48] L. Westover, *SPLATTING: A Parallel, Feed-Forward Volume Rendering Algorithm*, doctoral dissertation, University of North Carolina, Chapel Hill, Department of Computer Science, July 1991.

[49] T. Whitted, "An Improved Illumination Model for Shaded Display," *Communications of the ACM*, vol. 23, no. 6, June 1980, pp. 343–349.

[50] J. Wilhelms and A. V. Gelder, "A coherent projection approach for direct volume rendering," *ACM SIGGRAPH Computer Graphics*, vol. 25, no. 4, July 1991, pp. 275–284.

[51] R. Yagel, *Volume Rendering Polyhedral Grids by Incremental Slicing*, Tech. Rep. OSU-CISRC-10/93-TR35, Department of Computer and Information Science, Ohio State University, 2036 Neil Avenue, Columbus, OH 43210-1277, 1993.

[52] R. Yagel, "Towards Real Time Volume Rendering," *Proceedings of GRAPHICON '96*, Russia, July 1996, pp. 230–241.

[53] R. Yagel, D. Reed, A. Law, P. W. Shih, and N. Shareef, "Hardware Assisted Volume Rendering of Unstructured Grids by Incremental Slicing," *Proceedings of IEEE-ACM 1996 Volume Visualization Symposium*, San Francisco, California, October 28-29 1996, pp. 55–62.

[54] R. Yagel and Z. Shi, "Accelerating Volume Animation by Space-Leaping," *Proceedings of Visualization '93*, San Jose, California, October 1993, pp. 62–69.