APPROVAL SHEET

Title of Thesis: Numerical Integration Techniques for Volume Rendering

Name of Candidate: Preeti Bindu
Master of Science, 2012

Thesis and Abstract Approved: ________________________________
Dr. Marc Olano
Associate Professor
Department of Computer Science
and Electrical Engineering

Date Approved: ________________

Curriculum Vitae

Name: Preeti Subhash Bindu.

Permanent Address: 357, White Ash Place, Gaithersburg, MD – 20878.

Degree and date to be conferred: M.S. in Computer Science, May 2012.

Date of Birth: 02/04/1988.

Place of Birth: Aurangabad, India.

Secondary Education: Deogiri College, Aurangabad, India.

Collegiate institutions attended:
University of Maryland Baltimore County, M.S Computer Science, 2012.
Government College of Engineering, BE Information Technology, 2009.

Major: Computer Science.

Professional positions held:
Software Engineer Idea Fabrik Studios Plc. (November 2011 - Present)
3D Graphics Engineering Intern Global Science and Technology Inc.
(June 2010 – August 2011)

ABSTRACT

Title of Document:	NUMERICAL INTEGRATION TECHNIQUES FOR VOLUME RENDERING
	Preeti Subhash Bindu MS in Computer Science May 2012
Directed By:	Dr. Marc Olano, Associate Professor Department of Computer Science And Electrical Engineering

Medical image visualization often relies on 3D volume rendering. To enable interaction with 3D rendering of medical scans, improvements in the performance of Volume Rendering Algorithms need significant attention. Real-time visualization of 3D image data set is one of the key tasks of Augmented Reality Systems required by many medical imaging applications. Over past five years the development of the Graphics Processing Unit (GPU) has proved beneficial when it comes to Real Time Volume Rendering. We propose a GPU based volume rendering system for medical images using adaptive integration to improve performance. Our system is able to read and render DICOM images, implementing adaptive integration techniques that increase frame rate for volume rendering with the same quality of output images.

NUMERICAL INTEGRATION TECHNIQUES FOR VOLUME RENDERING

By

Preeti Subhash Bindu

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, Baltimore County, in partial fulfillment
of the requirements for the degree of
[Master of Science]
[2012]

© Copyright by
Preeti Subhash Bindu
2012

Dedicated to Aai, Pappa

Acknowledgements

I would like to express my sincere gratitude to my advisor Dr. Marc Olano. I thank him for the constant support and guidance, and for his continued belief in me throughout this thesis work. I am thankful for his words of advice and many skills I have gained by working with him.

I would like to extend my sincere thanks to Dr. Penny Rheingans, Dr. Samir Chettri for graciously agreeing to be on my thesis committee.

I am thankful to all VANGOGH Lab members for their suggestions. My special thanks to Kishlay Kundu for his help during the implementation phase. I thank my roommates and friends for timely reviewing my draft.

Table of Contents

Dedicated to Aai, Pappa i
Acknowledgements ii
Table of Contents iv
List of Tables vii
List of Figures viii
Chapter 1: Introduction 1
  1.1 Volume Rendering 1
  1.2 Applications of Volume Rendering 5
    1.2.1 Geoscience 5
    1.2.2 Astrophysics 5
    1.2.3 Chemistry 5
    1.2.4 Mechanical Engineering 6
    1.2.5 Gaming 6
    1.2.6 Medicine 6
  1.3 GPU Based Volume Rendering 8
    1.3.1 2D Texture Based Rendering 9
    1.3.2 3D Texture Based Rendering 9
    1.3.3 GPU Based Ray Casting 10
  1.4 Role of Integration in Ray Casting 11
  1.5 Thesis Contribution 12
Chapter 2: Background and Related Work 13
  2.1 Image Modalities 13
    2.1.1 PET Scan 13
    2.1.2 CT Scan 14
    2.1.3 MRI Scan 14
    2.1.4 X-Ray 15
    2.1.5 Multi-Modality 15
  2.2 Texture Generation 16
  2.3 Integration Techniques 17

2.3.1 Linear Integration 18
2.3.2 Runge-Kutta Integration 19
2.3.3 Monte Carlo Integration 19
2.4 Ray Casting 20
2.4.1 Ray Casting Explained 20
2.3.2 GPU Based Ray Casting 21
2.5 Integration in Volume Rendering 23
Chapter 3: Approach 25
3.1 System Architecture 25
3.2 Generation of Textures from DICOM Images 26
3.2.1 DICOM Image Format 26
3.2.2 Generation of Textures 27
3.3 Texture Based Ray-Casting for Volume Generation 30
3.4 Shader Programs 31
3.4.1 Vertex Shader Program 32
3.4.2 Fragment Shader Program 33
3.5 Numerical Integration Techniques 34
3.5.1 Relative Error Computation 35
3.5.2 Adaptation of Step Sizes 36
3.5.2 Numerical Integration Techniques Based on Poisson Distribution 38
3.6 Implementation in LAB Color Space 40
Chapter 4: Results 41
4.1 Datasets and Hardware Used 41
4.2 Comparison of Results 42
4.2.1 Level Value Comparison (Linear Changes) 42
4.2.2 Level Value Comparison (Scale Changes) 47
4.2.3 Integrated Value Comparison (Linear Changes) 51
4.2.4 Integrated Value Comparison (Scale Changes) 55
4.2.5 Effects of Threshold 59
4.2.6 Poisson Distribution Algorithm 60
4.2.7 Experiments in LAB Color Space 62

Chapter 5: Conclusion and Future Work 64
Bibliography 66

List of Tables

Table 4.1 Performance Table for Level Value Comparison (Linear Changes) 44
Table 4.2 Performance Table for Level Value Comparison (Scale Changes) 47
Table 4.3 Performance Table for Integrated Value Comparison (Linear Changes) ... 51
Table 4.4 Performance Table for Integrated Value Comparison (Scale Changes) 55

List of Figures

Figure 1.1 (a) & (b) Example of 2D slices of images acquired from a CT scan (c) 3D volume rendering of the input images (Image Courtesy: Osirix) 2

Figure 1.2 Volumetric Grid.. 3

Figure 1.3 Volume Ray Casting Technique.. 4

Figure 1.4 Application of Volume Rendering in Medicine (Image Courtesy: Osirix). 7

Figure 1.5 2D Texture Based Rendering 9

Figure 1.6 3D Texture Based Rendering 10

Figure 1.7 Multipass Ray Casting Approach.. 10

Figure 2.1 A Computed Tomography (CT) Scanner (Image Courtesy: www.medicinenet.com) 14

Figure 2.2 A Magnetic Resonance (MR) Image Scanner (Image Courtesy: www.medicinenet.com) 15

Figure 2.3 A CT and PET Scan and Their Combined Image (Image Courtesy: www.mayoclinic.com) 16

Figure 2.4 Texture Based Ray Casting 17

Figure 2.5 Components of a Ray-Casting System.. 20

Figure 3.1 System Architecture……………………………..……………………25

Figure 3.2 Information Exchange using DICOM Images………………..………..27

Figure 3.4 Texture Slices for Skewed Head Datased in an Increasing Order from (a) to (f)…………………………………………………………………..………28

Figure 3.5 Texture Slices for Lobster Datasein an Increasing Order from (a) to (f)...29

Figure 3.6 Ray Casting Principle, One Ray per Pixel……………………… ……30

Figure 3.7 Fixed Functional and Programmable Units of a GPU Pipeline..…………32

Figure 3.8 Hierarchy of Integration Technics Implemented………………………....36

Figure 4.1 Level Value Comparison (Linear Changes): (a) Original Image (b) Shader Implementation (c) Difference Image [Dataset: Skewed Head]........................ 43

Figure 4.2 Level Value Comparison (Linear Changes): (a) Original Image (b) Shader Implementation (c) Difference Image [Dataset: Skewed Head]........................ 44
Figure 4.3 Level Value Comparison (Linear Changes): (a) Original Image (b) Shader Implementation (c) Difference Image [Dataset: Skewed Head]........................ 45
Figure 4.4 Performance Graph for Level Value Comparison (Linear Changes)........ 46
Figure 4.5 Image Quality Graph for Level Value Comparison (Linear Changes) 46
Figure 4.6 Level Value Comparison (Scale Changes): (a) Original Image (b) Shader Implementation (c) Difference Image [Dataset: Lobster].................................. 48
Figure 4.7 Level Value Comparison (Scale Changes): (a) Original Image (b) Shader Implementation (c) Difference Image [Dataset: Lobster].................................. 49
Figure 4.8 Performance Graph for Level Value Comparison (Scale Changes).......... 50
Figure 4.9 Image Quality Graph for Level Value Comaprison (Scale Changes) 50
Figure 4.10 Integrated Value Comparison (Linear Changes): (a) Original Image (b) Shader Implementation (c) Difference Image [Dataset: Skewed Head]............ 52
Figure 4.11 Integrated Value Comparison (Linear Changes): (a) Original Image (b) Shader Implementation (c) Difference Image [Dataset: Skewed Head]............ 53
Figure 4.12 Performance Graph for Integrated Value Comparison (Linear Changes)54
Figure 4.13 Image Quality Graph for Integrated Value Comparison (Linear Changes) 54
Figure 4.14 Integrated Value Comparison (Scale Changes): (a) Original Image (b) Shader Implementation (c) Difference Image [Dataset: Skewed Head]............ 56
Figure 4.15 Integrated Value Comparison (Scale Changes): (a) Original Image (b) Shader Implementation (c) Difference Image [Dataset: Skewed Head]............ 57
Figure 4.16 Performance Graph for Integrated Value Comparison (Scale Changes). 58
Figure 4.17 Image Quality Graph for Integrated Value Comparison (Scale Changes) 58
Figure 4.18 Threshold Comparison (ε = 0.15): (a) Original Image (b) Shader Implementation (c) Difference Image.. 60
Figure 4.19 Threshold Comparison (ε = 0.2): (a) Original Image (b) Shader Implementation (c) Difference Image.. 60

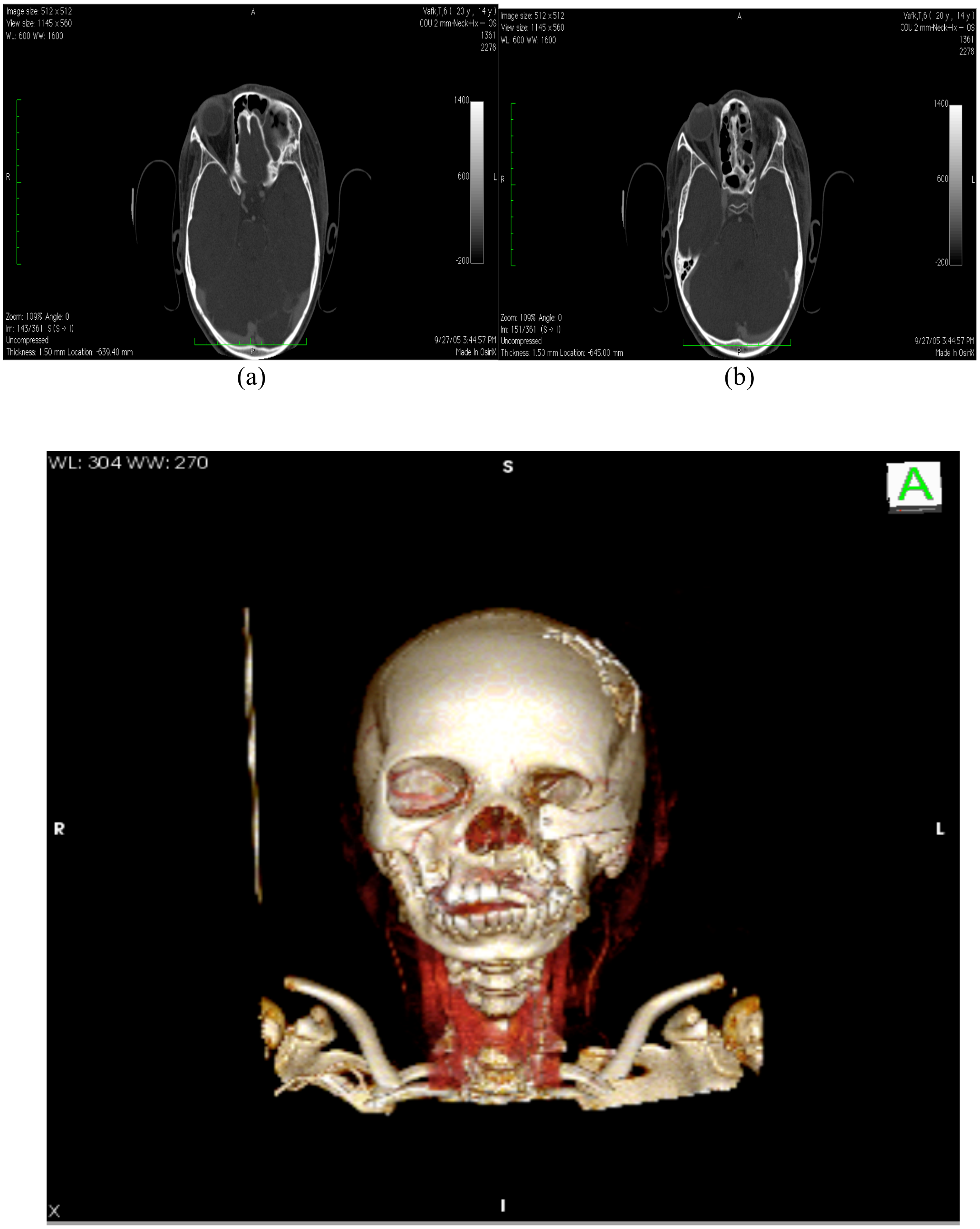
Figure 4.20 Poisson Distribution Algorithm: (a) Original Image (b) Shader Implementation (c) Difference Image.. 61
Figure 4.21 Poisson Distribution Algorithm: (a) Original Image (b) Shader Implementation (c) Difference Image.. 62
Figure 4.22 RGB vs. LAB Comparison (Linear Changes): (a) RGB Image (b) LAB Image.. 63
Figure 4.23 RGB vs. LAB Comparison (Scale Changes): (a) RGB Image (b) LAB Image.. 63

Chapter 1: Introduction

The healthcare industry has always been a motivation for research in computer graphics. The increasing demands on clinicians have led to the real-time visualization of 2D and 3D image data. The inputs to the volume visualization technique are the images acquired from CAT, MR or PET scan, the sizes of which tend to be fairly large. It is a difficult task to achieve interactive frame rates with the highest accuracy. The computational complexity of the CPU based volume rendering algorithms has led to the use of new technologies involving Graphics Processing Units (GPUs). GPUs have hundreds to thousands of parallel processors. During graphics rendering, these processors execute programs; the program is executed once per vertex (called a *vertex shader*), or once per pixel (called a *fragment shader*). Modern graphics cards include a GPU capable of executing small vertex and fragment shader programs. The use of these shader programs can harness the power of the GPU for real-time rendering. In this chapter we explore the process of volume rendering, along with its applications and the motivation behind this thesis and the contributions made.

1.1 Volume Rendering

Volume rendering is the process of transforming 3D discretely sampled dataset into a 2D image [MW10]. The discretely sampled data can be in the form of the images acquired from CAT or MR scans and other such modalities. The 3D data consists of 2D slices of these images. Figures 1.1 (a) and (b) show the 2D slices acquired from a CAT scanner while Figure 1.1 (c) shows the corresponding volume created using the process of volume rendering:



(a) (b)

(c)

Figure 1.1 (a) & (b) Example of 2D slices of images acquired from a CT scan (c) 3D volume rendering of the input images (Image Courtesy: Osirix)

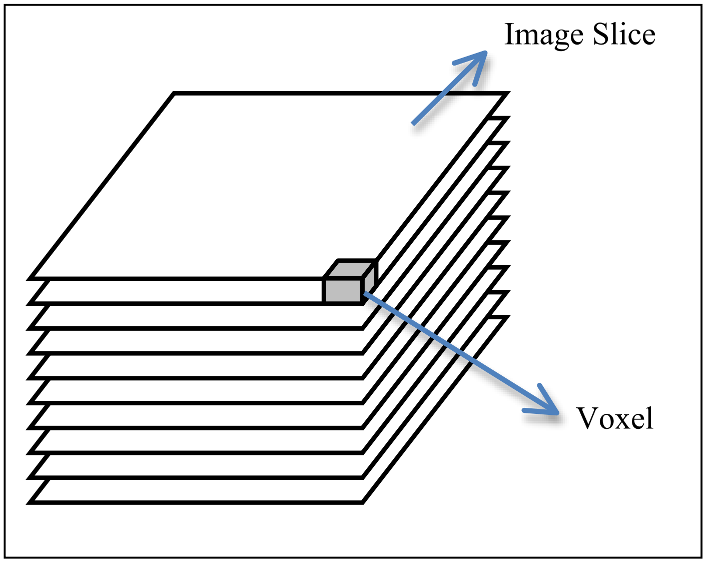


Figure 1.2 Volumetric Grid

As shown in Figure 1.2, the volume dataset is in the form of 2D image slices. These images slices form a volumetric grid each element of which is called as a *voxel* as shown in Figure 1.2. Each *voxel* is assigned an opacity and color value, which contribute to the formation of the volume. The color values are computed by a color transfer function in the form of red, green and blue (RGB) color channels while the opacity is defined by an opacity transfer function in the form of an alpha (A) channel.

There have been numerous methods and algorithms presented for volume rendering of 3D discretely sampled dataset with an aim of increasing the accuracy as well as the speed of the rendering process. These methods can be grouped into the following categories:

1. Volume Ray Casting: In the Volume Ray Casting technique, a ray is cast from the center of the camera model to each image pixel on an imaginary screen [KH84], [DCH88]. The imaginary screen is considered to be in between the camera model and the 3D input dataset. The ray then passes through these image pixels into the dataset where the ray is sampled at various points and the corresponding color and opacity

transfer functions are calculated at each of these points to compute the ultimate color and opacity of the volume [RPSC99].

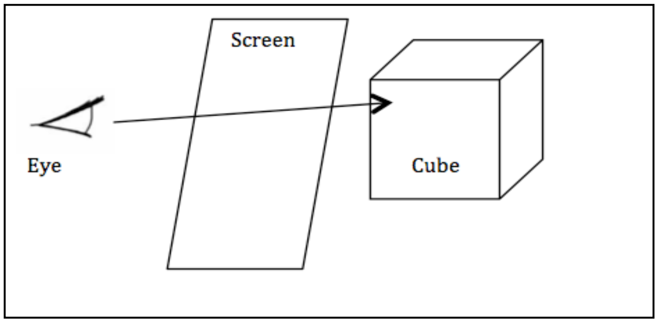


Figure 1.3 Volume Ray Casting Technique

2. Splatting: Splatting considers one voxel at a time to find its contribution across a set of image pixels (as compared to ray casting that considers one pixel at a time across a set of voxels along the ray). The process involves projecting voxels in a back-to-front order and calculating the color and opacity based on the *Gaussian Splat* as discussed in [WA91].
3. Iso-surface Rendering: Surface rendering involves creating surfaces of an object in the form of a 3D image. As opposed to volume rendering, where the volume is rendered using the image data itself, iso-surface rendering involves modeling the object using geometric primitives such as points, lines, triangles, etc. There are many ways of computing this surface, for example the marching cubes algorithm [LC87]. Surface rendering hides the internal information in the volume and thus isn't extensively used as compared to volume rendering [KOR08].
4. Texture Mapping: Using commodity graphics hardware, texture based volume rendering has evolved as a result of the fast and more accurate outputs. Texture mapping consists of the process of applying 3D textures or images to geometric objects. Texture based volume rendering efficiently renders slices of a 3D volume with the real-time interaction capability and thus can be an ideal source for rendering medical image datasets [EHK* 06] [CCF94].

1.2 Applications of Volume Rendering

Volume rendering is an efficient way of visualizing data and thus plays an important role in scientific visualization. Following subsections describe how volume rendering can be applicable in different areas for the visualization of the volumetric datasets.

1.2.1 Geoscience

Geoscience includes all the sciences (geology, geophysics, geochemistry) that study the structure, evolution and dynamics of the planet Earth and its natural mineral and energy resources. The different functions associated with the geoscientists involve geological survey and mapping, energy supplies, finding rocks as natural resources and so on. Volume rendering implementations in geoscience plays a vital role in different tasks such as real time scanning of massive datasets, simultaneous multiple volume viewing, well logs, tops and cultural data display, display of reservoir models, interpretation of multiple 3D seismic surveys into multiple 2D seismic surveys, isolation of different geobodies, etc.

1.2.2 Astrophysics

According to Li et al. **[**LFH08**],** with advancements in the measurement technology for astrophysical imaging, the view of the sky is not limited to 2D Celestial Sphere. A third dimension has been added allowing the examination of wavelength radio, microwaves, very short X-rays, and gamma rays. This leads to the exploration of volume visualization techniques including textured image stacks, the horseshoe representation and GPU-based volume visualization.

1.2.3 Chemistry

Different areas where volume visualization in chemistry plays a vital role include visualization of quantum chemical and molecular dynamic modeling, drug designing, drawing complex molecular structures including crystal structure data, creation of a system for visualizing chemical reactions, etc. [JV09].

1.2.4 Mechanical Engineering

Volume rendering plays a vital role in the field of mechanical engineering in various areas such as computer-aided design, design of power plants, automobile engineering, simulator designs and so on. CADD (Computer Aided Design and Drafting) involving 2D and 3D volume rendering is used in a large scale to provide users an interface for streamlining design processes, drafting, documentation, and manufacturing. The design of different complex structures relating to automobiles involves the creation of 3D volumes followed by their practical implementation. Simulation design is the area that has started gaining importance in different fields such as training, testing, professional usage and so on. Simulators are used to provide an in depth knowledge of physics principles and to use these principles to design virtual models of mechanical designs. Such simulators make use of volume rendering as a tool to make the simulator as realistic as possible [WH03].

1.2.5 Gaming

The gaming industry has been deploying different computer graphics technologies since the earliest computer games. Game designing involves different aspects such as design of gameplay, storyline, environment and characters pertaining to a game. Environment design involves the design of many real-world phenomena such as clouds, smoke, fog, explosions, fire and so on. Volume rendering simulates effects of light emitted, scattered, and absorbed by a large number of tiny particles in the volume. This volume is represented as a uniform 3D array of the samples, which are either pre-computed or procedural [JB10].

1.2.6 Medicine

The main concern of medical visualization is to aid clinicians in their diagnosis, surgery and in various training demonstrations. On this front, direct volume rendering is important as the surface rendering may lead to missing most important inner parts of the data by rendering only the outer surfaces [KM04]. Direct volume rendering consists of making the visualization volumes such that almost all image pixels can contribute to the final visualized volume so that the inner parts can be displayed efficiently. Volume rendering in medicine involves very large input data in the form of images acquired from different scanners such as CAT, MRI, PET, X-Ray, etc. These images contain 3D data that the doctors need to see in different directions, scales that can efficiently be done with the help of 3D volumes. Another importance of volume rendering in medicine lies in the process of making these volumes semitransparent, which allows examining the internal organs and relevant details. Figure 1.4 shows how volume rendering can be useful in rendering the same 3D dataset in different formats allowing doctors for a better volume visualization of the images.

Application of volume rendering in the field of medicine also has significant importance in surgical planning. The surgeon can perform a mock surgery on the 3D volume developed using the patient's image data to decide the actions that need to be taken during the actual surgery.

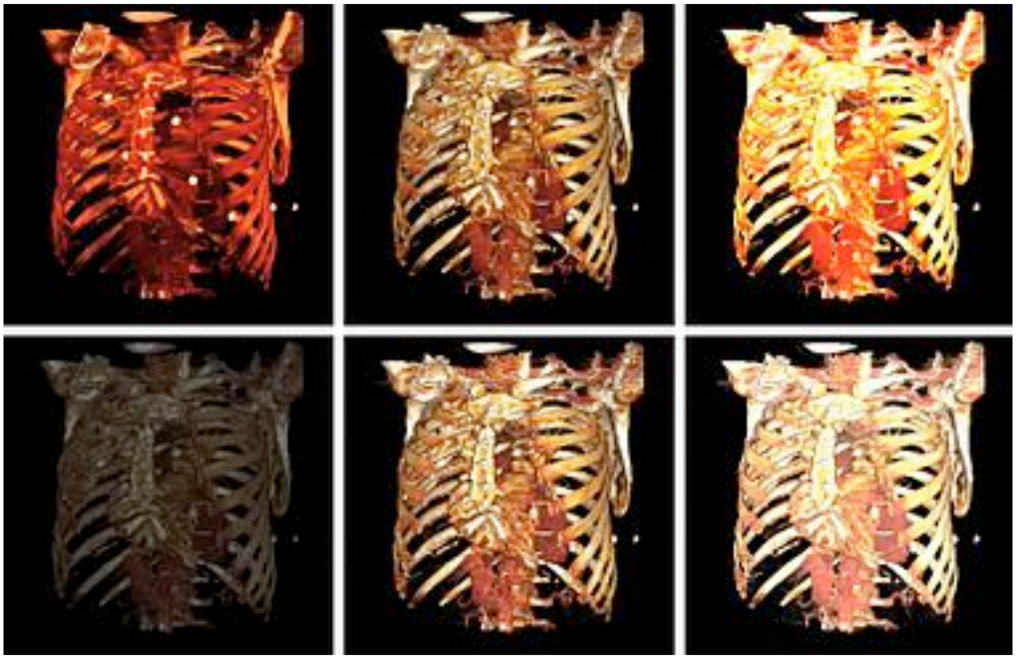


Figure 1.4 Application of Volume Rendering in Medicine (Image Courtesy: Osirix)

1.3 GPU Based Volume Rendering

Commonly used algorithms for volume rendering run on the system CPU. The CPU based execution is computationally expensive given the large datasets and complexities involved in the scientific visualization.

A Graphics processing unit (GPU) is designed with a goal of accelerating the process of building the images in the frame buffer. GPUs have proven helpful in the field of computer graphics by their parallel structure that makes computation of complex data structure faster and more accurate [HN08]. The GPUs were first designed to accelerate the process of texture mapping and rendering of polygons. Later advancements in the GPUs resulted in the accelerated geometric computations such as rotation, translation, etc. With the help of GPUs, efficient programming for various coordinate systems operations is possible. Modern development in GPUs has resulted in the development of shaders in the language similar to C where vertices and textures can be manipulated to perform various operations such as oversampling and interpolation techniques to reduce aliasing, handling very high precision color spaces and so on [HN08].

GPU computing involves usage of both CPU and GPU in a heterogeneous co-processing computing model. The sequential part of an application runs on CPU while GPU handles the computationally expensive part of the application. Following are some of the architectural advancements in GPUs that make them the first choice of many graphics programmers:

1. Chips are based on multiprocessor with eight to ten cores, hundreds of ALUs, thousands of registers and some shared memory.
2. A graphics card contains fast global memory accessible by all multiprocessors, local memory accessible by each multiprocessor, and special memory for constants.

3. The cores in the multiprocessors execute instructions simultaneously, which is the basic style of graphics programming [HL04].

   Following are the three main types of GPU based rendering:

1.3.1 2D Texture Based Rendering

2D texture mapping involves exploiting the 2D texture mapping capabilities by storing volumetric data in several textures. In order to perform the rendering, an object-aligned stack of texture slices is created as shown in Figure 1.5. All the polygons in the stack must be aligned to a single axis since for the 2D textured data one of the coordinates should be constant. The polygons are mapped with the respective 2D texture, which, in turn, is resampled by the hardware-native bilinear filtering. Three stacks of polygons corresponding to each of the axis to which the textures are aligned are created. The selection of the stack depends upon the camera orientation. [EHK* 06]

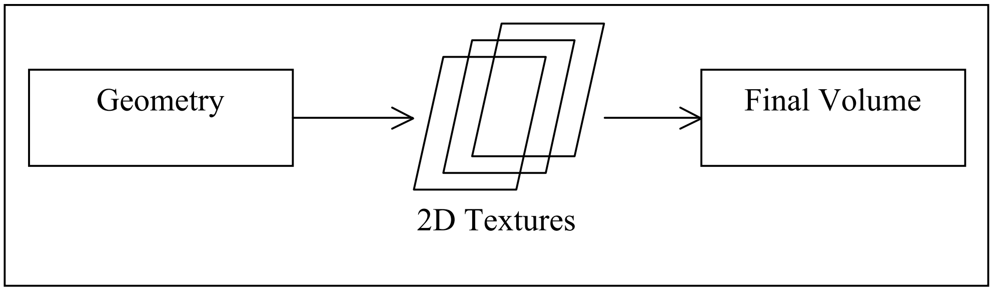


Figure 1.5 2D Texture Based Rendering

1.3.2 3D Texture Based Rendering

A fixed number of slices and the axis-alignment cause several problems to the 2D based rendering. The volumes appear similar to being rendered using CPUs. Thus in order to acquire better accuracy for the volumes, 3D textured based rendering can be used. Unlike the 2D texture based rendering, 3D textures based rendering uses viewport-aligned slices that allow the user to form the stack of the slices based on what the application demands. Figure

1.6 depicts the viewport-aligned slices. Thus 3D texture based rendering allows selecting and sampling arbitrary points in the 3D textures to form the polygons.

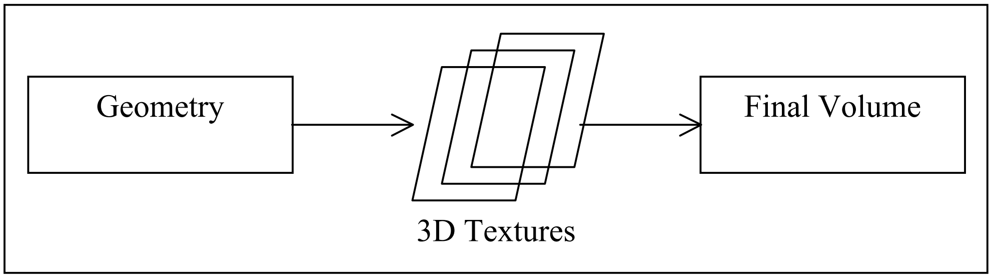


Figure 1.6 3D Texture Based Rendering

1.3.3 GPU Based Ray Casting

Ray casting is the most natural approach to volume rendering, and, when implemented in software, this approach is most often used. A ray is cast from the camera location to each visible pixel on the surface of the volume-bounding box and values are sampled at regular intervals along the segment of each ray inside the volume. It is computationally expensive, as a large number of samples have to be taken along each ray to get acceptable quality. Figure 1.7 shows principle structure for multipass rendering for GPU ray casting [EHK*06].

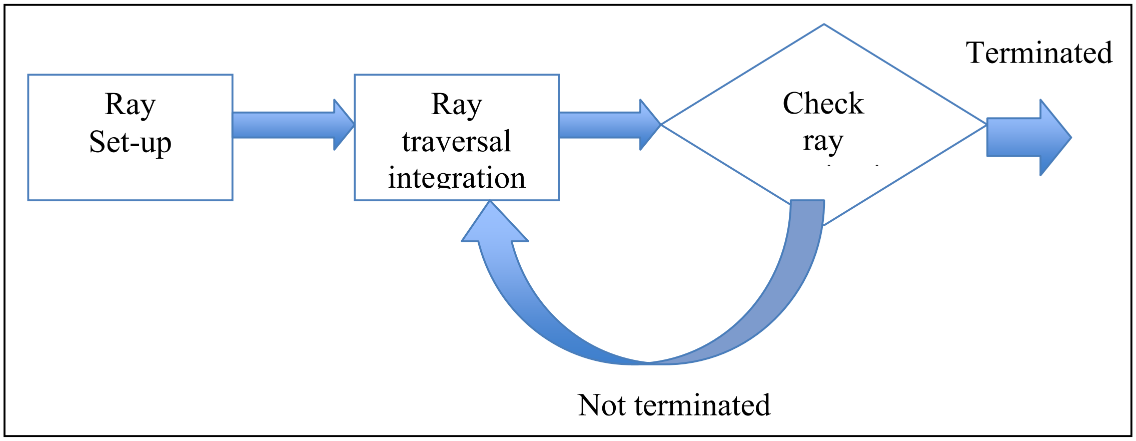


Figure 1.7 Multipass Ray Casting Approach

1.4 Role of Integration in Ray Casting

As explained in the Section 1.3.3, a ray casting technique transforms a limited form of data into 3D projection by tracing rays from the viewing point into the volume. The geometry setup for a texture based volume rendering involves creating a view-port aligned stack of textures generated from 3D images. In order to perform the volume rendering, a ray is cast from the viewing point through this stack intersecting the pixels in the volume.

Each pixel along the ray has a color and opacity value associated with it which are used to determine the transparency, semi-transparency or opacity of the volume as well as the color in the form of RGB values. As the ray progresses from the viewpoint into the volume these chromaticity and opacity values are integrated to calculate the final color and opacity of the volume. For a texture-based volume rendering, when we slice the volume in a back-to-front order these values are stored in a one-dimensional lookup table, which is used to transform the volume data into color, and opacity values [RGW* 03].

The chromaticity and opacity values of a value depend upon the effect of light on a volume. For example, for a given volume, the transparent volume corresponds to all the particles inside the volume emitting the light in contrast to absorbing it. Similarly there can be some opaque particles, which absorb all the amount of light being inserted to it. Reflection of light can also develop the scattering effect in a volume. Shadows can be formed based on the direction of light and the opacity of the particles inside the volume. Based on these shadows some of the particles may not be a part of the final volume and need to be discarded from the rendering process. All these calculations need to be performed for each ray-volume intersection and should be stored in the form of color and opacity values in a lookup table. This lookup table is then used to find the integrated value of the final color and opacity of the volume. This integration is given by equation (1.1) [DCH88].

$$I(D) = I_0 e^{-\int_0^D \tau(t)\,dt} + \int_0^D L(s)\tau(s) e^{-\int_0^s \tau(t)\,dt}\, ds \tag{1.1}$$

This equation says the illumination at distance D along a ray, I(D), is a combination of the background illumination I_0, attenuated by the volume from 0 to D, and the local lighting contribution at each point, s, along the path from 0 to D, attenuated by the portion of the volume between 0 and s.

1.5 Thesis Contribution

The main contributions of this thesis include:

1) Three algorithms for adapting the integration step size based on local color and opacity: a Poisson distribution algorithm to determine the number of samples in the given interval of two consecutive steps, an adaptive step size integration scheme making linear changes to the level step, and an adaptive step size integration scheme making scaling changes to the integration step size

2) Comparison of results of different integration techniques on image quality and performance in frames per second.

Chapter 2: Background and Related Work

A significant amount of work has been done with a goal of accelerating the process of volume rendering. Given the applications of volume rendering as discussed in Chapter 1, an efficient implementation of the advancements in graphics hardware would result in better and faster volume rendering methods. Understanding the application of volume rendering in medicine requires some knowledge in medical imaging techniques. This chapter discusses different medical imaging techniques, followed by an overview of some mathematical integration techniques. We then discuss some prior research in the field of integration for volume rendering.

2.1 Image Modalities

The overall objective of medical imaging is to acquire useful information about physiological processes or organs of the body by using external or internal sources of energy [PS02]. This subsection discusses the ways by which images are acquired from various medical scanning instruments, their modalities and the differences among them. Based upon the energy source used for acquiring the images, following are the main categories of medical images that are handled in our work.

2.1.1 PET Scan

A PET scan stands for Positron Emission Tomography that allows doctors to examine signs of disease, such as areas with increased activity that may signal the origins of cancer. A PET/CT scan involves acquiring the images in a computer-assisted manner [www.mayoclinic.com].

2.1.2 CT Scan

CT stands for Computed Tomography and it is considered to be a sophisticated form of X-Ray imaging. The scanner takes X-Ray images of the body at many different angles around the body thus generating a series of images. A CT scan shows clear images of bone, internal organs, muscles and blood vessels and allows doctors to distinguish between normal and diseased or injured tissue [www.medicinenet.com]. Figure 2.1 shows a CT scanner:

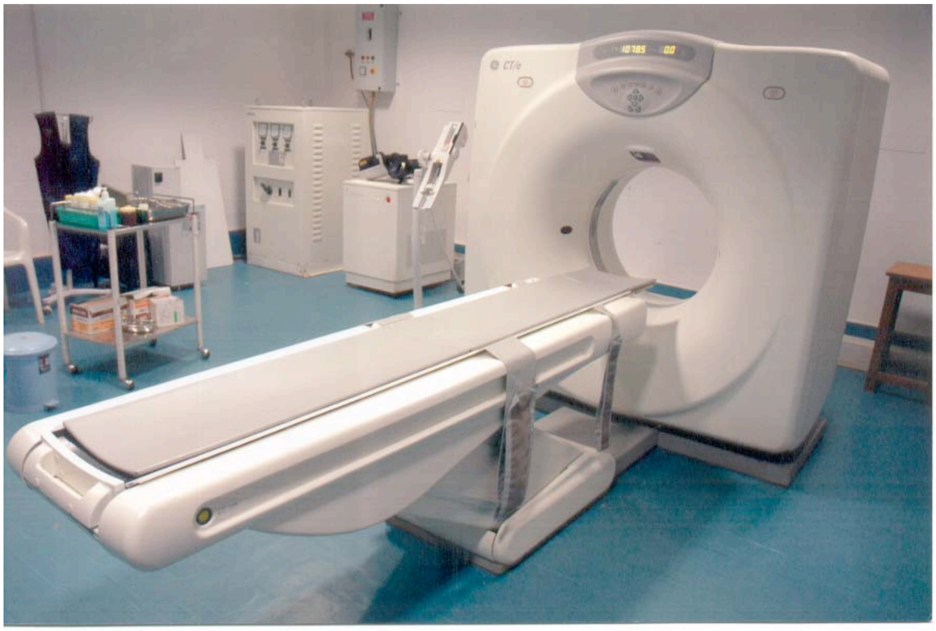


Figure 2.1 A Computed Tomography (CT) Scanner (Image Courtesy: www.medicinenet.com)

2.1.3 MRI Scan

MRI stands for Magnetic Resonance Imaging. This type of imaging modality uses radio waves that force the nuclei of a body's atoms to different locations that send out their own waves while coming back to their original positions. These waves are then used by a computer to generate images of body structures. The MRI scanner contains a giant circular magnet that creates a strong magnetic field that aligns the protons of hydrogen atoms, which are then exposed to a beam of radio waves. This spins the various protons of the body, and they produce a faint signal that is detected by the receiver portion of the MRI scanner [www.medicinenet.com]. Figure 2.2 shows the MRI scanner:

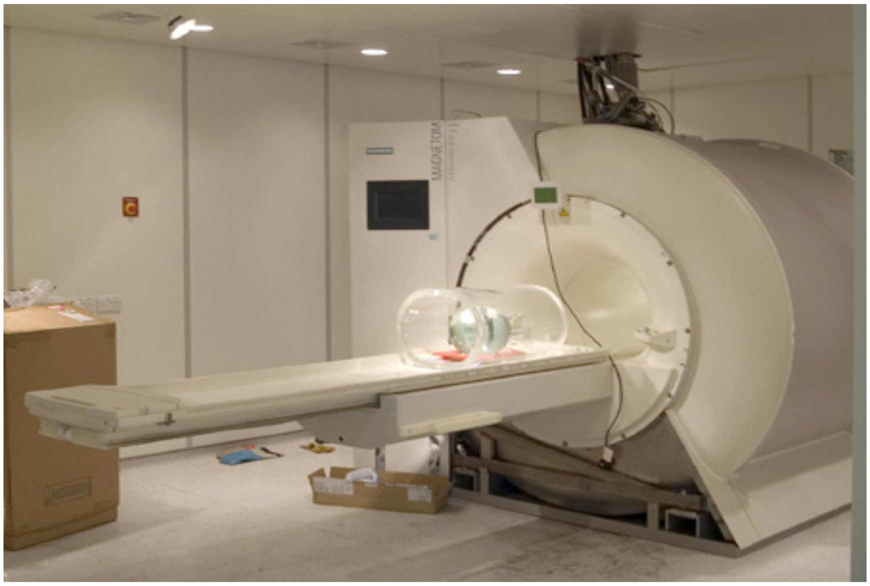


Figure 2.2 A Magnetic Resonance (MR) Image Scanner (Image Courtesy: www.medicinenet.com)

2.1.4 X-Ray

An x-ray image is produced when a small amount of radiation passes through the body. The ability of x-rays to penetrate tissues and bones varies according to its composition and mass, which allows doctors to obtain images from inside the body [PS02].

2.1.5 Multi-Modality

As discussed above, a PET scan can efficiently detect physiologic changes in the body while a CT scan is helpful in detecting the anatomical structure of the body where changes are taking place. Combining these two scans in one PET/CT imaging technique provides, during a single outpatient exam, detailed information to physicians about the presence or spread of disease and accurately identifies its precise location. Figure 2.3 shows a PET and CT scan separately and also a combination of them.

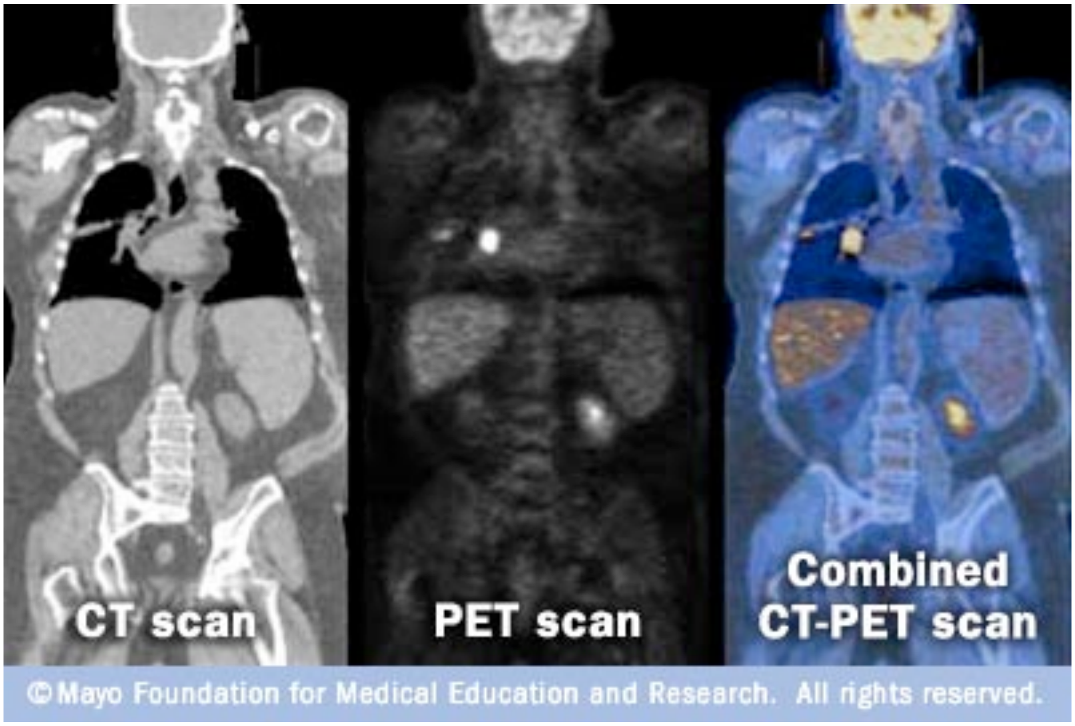

CT scan
PET scan
Combined
CT-PET scan
©Mayo Foundation for Medical Education and Research. All rights reserved.

Figure 2.3 A CT and PET Scan and Their Combined Image (Image Courtesy: www.mayoclinic.com)

2.2 Texture Generation

In the most generalized form, texture mapping is the process used for adding more realism to the rendering process. The simplest way to imagine texture mapping is to consider pasting a photograph onto the surface of a polygon. This photograph can be of wood, stone, brick, cloth and so on. Textures are arrays of data. These data values can be color, luminance or color and opacity data.

In case of the GPU programming textures can be used as a memory. Data values pertaining to chromaticity and opacity values can be stored in the texture memory. This texture memory is then used in order to perform the volume rendering by ray casting or the ray-tracing approach. This process is called as texture based volume rendering. The basic idea behind texture based volume rendering is to render a stack of texture slices. This is shown in the following figure.

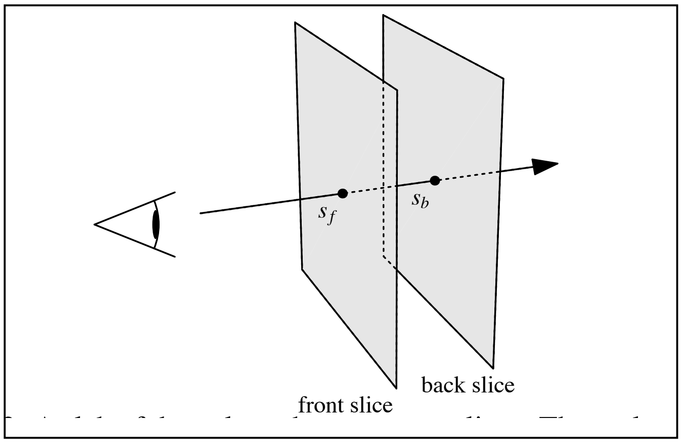


Figure 2.4 Texture Based Ray Casting

The figure also shows how a ray can be cast through this stack of textures. As each fragment of a slice corresponds to one ray segment, the whole slice corresponds to a slab of the volume as depicted.

2.3 Integration Techniques

In calculus, numerical integration corresponds to finding a numerical estimate for a definite integral. The problem that the numerical integration handles is to compute a solution for a definite integral:

$$\int_{a}^{b} f(x)dx \tag{2.1}$$

Here, we try to compute the integral over a definite interval [a,b] for a function *f(x)*. The need for calculating the integral numerically arises in many cases. The integrand *f(x)* may be too complex to find a closed-form solution. It is also possible that the integrand *f(x)* is known only at certain arbitrary points and we need the composite value. This is also the case for determining the integral in volume rendering which will be discussed in next subsection.

Below we discuss some of the main techniques of calculating the integral as defined in calculus.

2.3.1 Linear Integration

Linear integration is the most basic form of the integration methods where the integration is defined by the following formula:

$$I(f) = \int_a^b f(x)dx \tag{2.2}$$

This is a linear functional defined over a vector space C[a, b] of continuous functions on the interval [*a*, *b*] to the real numbers. The linearity of *I*(*f*) follows from the standard facts about the integral: $y' = f(t,y), y(t_0) = y_0$

$$\begin{aligned} I(f+g) &= \int_a^b (f(x)+g(x))\,dx \\ &= \int_a^b f(x)\,dx + \int_a^b g(x)\,dx \\ &= I(f) + I(g) \end{aligned} \tag{2.3}$$

where, *f(x)* and *g(x)* are two integrals for the defined interval of [a, b] and their addition corresponds to the addition of the respective integral values for the given interval [DR07].

2.3.2 Runge-Kutta Integration

The Runge-Kutta methods are used for the implicit and explicit approximation of solutions of ordinary differential equations. The most commonly used Runge-Kutta method is the fourth order integration known as “RK4”.

In a Runge-Kutta method an initial value problem and the corresponding RK4 method is given by the following equations:

$$y' = f(t, y), \qquad t_{n+1} = t_n + h \;\; y(t_0) = y_0$$

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

$$t_{n+1} = t_n + h \tag{2.4}$$

where y’ is a function of time *t* and y and y_{n+1} is the RK4 approximation of $y(t_n+1)$ and the constants k_1, k_2, k_3 and k_4 are calculated based upon *y* and *t* [DR07]. Runge-Kutta integration method can be used adaptively wherein the step size can be altered adaptively based upon an estimation of the truncation error.

2.3.3 Monte Carlo Integration

Monte Carlo integration is a family of numerical integration techniques, which integrate the values at random points as opposed to other numerical integration techniques. Monte Carlo integrations are used for the approximate evaluation of the definite integrals; normally working on the multidimensional ones. The distinguishing property of Monte Carlo integration lies in the way the integrand is operated over random points unlike a regular grid of points [DR07].

The traditional Monte Carlo algorithm distributes the evaluation points uniformly over the integration region. Adaptive algorithms such as VEGAS [GL80] and MISER [PF90] use importance sampling and stratified sampling techniques to get a better result.

2.4 Ray Casting

The process of ray casting takes 3D images as input and makes it possible to render them on a two-dimensional screen. This is accomplished by tracking the rays of light that trace a direct path from the eye to some source of light. However, ray casting discounts the influence of any element that may intersect that path between the eye and the light source. In this subsection we discuss the process of ray casting and previous work done in this area.

In its simplest form, ray casting can be considered as an approach where a ray of light is cast from an eye and continuing on its path until it is blocked by some object. As opposed to ray tracing where the process involves calculation of ray of light being reflected refracted and absorbed by an object; ray casting considers the path of a single ray along a direction.

2.4.1 Ray Casting Explained

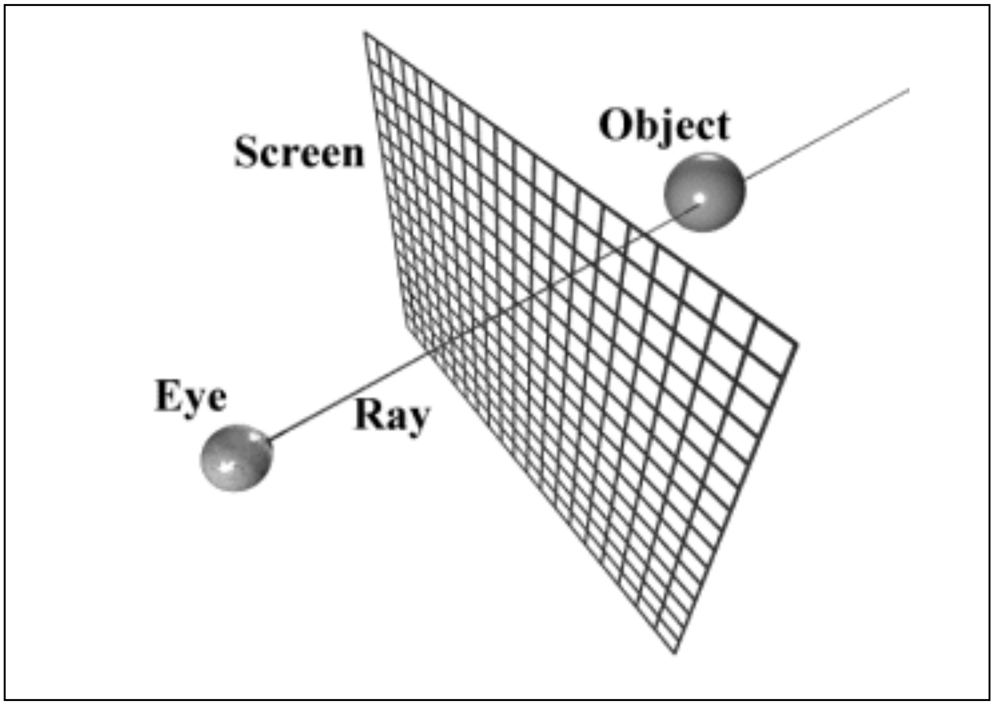


Figure 2.5 Components of a Ray-Casting System

Figure 2.5 depicts the common components of a ray casting method. As shown in the figure a ray is cast from an eye that goes through the image plane or the screen where we are trying to render the 3D image data and continues on till it hits the object.

The complex geometry involves multiple objects in the scene with multiple refractive indexes that change the percentage of the ray that is going to form the pixel on the screen. This calculation of the percentage of the light ray being reflected refracted or absorbed by the object; depends upon the color and opacity of each pixel in the 3D image data. Ultimately the color and opacity of the final volume is calculated as a composition of the color and opacity values of these individual pixels. This process is explained in detail in the following subsection. We limit our discussion to the process of ray casting in this subsection.

Following is the list of the components of the ray casting system as explained in [RPSC99]

1. **Memory System** provides the necessary voxel values.
2. **Ray-path Calculation** determines the voxels that are penetrated by the given ray
3. **Interpolation** estimates the value at a sample location using a small neighborhood of voxels
4. **Gradient estimation** estimates a surface normal using the neighborhood voxels
5. **Classification** maps interpolated sample values and the estimated surface normal to a color and opacity
6. **Shading** uses gradient and classification information to compute a color based upon the interaction of light with the estimated surfaces in the dataset
7. **Composition** uses shaded color values and opacity to compute a final pixel color for display.

2.3.2 GPU Based Ray Casting

We had a brief walkthrough of what a GPU is and how GPU based rendering algorithms work in chapter one. This subsection deals with some previous work in the field of ray casting on GPUs.

A detailed explanation of GPU based ray casting technique is discussed in [HL04]. As per [HL04] the basic idea of this technique is to store the entire volume in a single 3D texture and use GPU based implementations to render 3D volumes. These implementations include creating a vertex and fragment program. Eide explores various GPU based volume rendering algorithms in [KE05]. The document explains in detail the evolution of graphics cards as well as numerous rendering techniques. It also talks about GPU programming using OpenGL Shading Language called GLSL. Some interesting work in the field of hardware-based ray casting can be seen in work by Stegmaier et al. [SSKE05]. Stegmaier et al. present a flexible framework for GPU-based volume rendering. Their work can be extended to other shader functionalities. First introduced by Cullip and Neumann [CN94] and Cabral et. al [CCF94], the basic concept of ray casting involves creating planar slices based upon the geometry of the volume data and sampling through them. A volume rendering integral is evaluated at each sample point, by accumulating the textured slices.

More advanced texture mapping capabilities of the graphics hardware are used to the simple GPU-based ray-casting approach. Research in the field of GPU based volume rendering mainly focuses on two aims: To increase the quality of the volume being rendered and to accelerate the process of volume rendering. Research in improving the quality of volume renderings involves implementing multidimensional transfer functions as explained by Kniss et al. [KKH02]. In order to achieve high quality 3D volumes of medical image datasets Kreeger and Kaufman [KK99] present an algorithm that renders opaque and/or translucent polygons embedded within volumetric data. Two streams of algorithms are considered for accelerating the process of volume rendering on GPUs: The early ray termination technique was first introduced by Whitted [TW80]. As discussed in this paper and used by Levoy in his work in [ML90] the early ray termination algorithm has a very simple structure. It uses the basic ray casting approach and calculates the color and opacity values along the ray path. While calculating the opacity values if a value with highest opacity is encountered, the assumption that nothing beyond that point can be visible is used to terminate the ray at that point. Levoy [ML90] incorporated the empty space-skipping algorithm. The empty space-skipping algorithm avoids processing empty voxels with the help of various pre-computed data structures such as binary volumes, proximity clouds and so on. Li et. al. have implemented the empty space skipping algorithm by skipping the rendering of invisible voxels and also presented an algorithm that computes incrementally the intersection of the volume with the slicing planes in [LMK03]. An integration of early ray termination and empty space skipping approach, in order to accelerate the process of GPU-based volume rendering is presented by Kruger and Westermann [KW03].

2.5 Integration in Volume Rendering

As explained in Chapter 1, volume rendering is the process of representing 3D image data in 2D form. Many implementations are used in order to achieve this representation. The field of computer graphics has always been relying on mathematics for the representation of complex phenomena such as lighting, reflection, refraction, coordinate systems, projections, collisions and so on. This subsection explores the role of integration in volume rendering and the past research that has taken place in this area.

In a ray casting approach, a transfer function, which maps scalar values and gradients to color and opacities, is applied to the volume integral. These color and opacity properties of the volume are composited along the rays to obtain final pixel colors. This process of composition relies on mathematical integration. The integration process depends upon the number of sampling intervals in the volume data, which in turn depends upon the size of the data. With increasing size of input dataset, improvements in the integration process are needed in order to accelerate the volume rendering which resulted in numerous mathematical integration techniques being employed in fragment program of GPU-based ray casting technique [EHK*06].

The transfer function must be smooth in order to compute the numerical integral efficiently and accurately as no general, closed-form solutions are available. For the case of a smooth transfer function, well-known adaptive numerical schemes can be employed [NA92], and theory from Monte Carlo integration can guide their design [DH92]. However for certain rendering effects an infinite amount of sampling density is required. Implementation of smooth transfer function limits this range. To handle this problem Roettger et al. introduced a now well-known approach of preintegration in their work [RGW* 03]. In this work they have demonstrated a way to merge extensions of the original slicing approach namely pre-integration, volumetric clipping and advanced lighting. According to this work and the work of Westermann and Ertl [WE98], the slicing of the volume results in the ring artifacts that can be resolved by rendering slabs instead of slices. The ray integral of the ray segments inside each slab is a function of the scalar values at the entry and exit points of the ray, and the thickness of the slab. The pre-integration technique discussed by Engel et al. [EKE01] involves the ray integral calculation of transfer function for each combination of scalar values, which is looked up for each rendered pixel.

Chapter 3: Approach

In this chapter, we discuss the system implementation. The focus of our implementation was to experiment different integration techniques and compare them. We discuss the adaptive integration techniques and Poisson distribution technique in this chapter. In order to better describe our implementation we start with the system architecture followed by describing each phase in the architecture in detail.

3.1 System Architecture

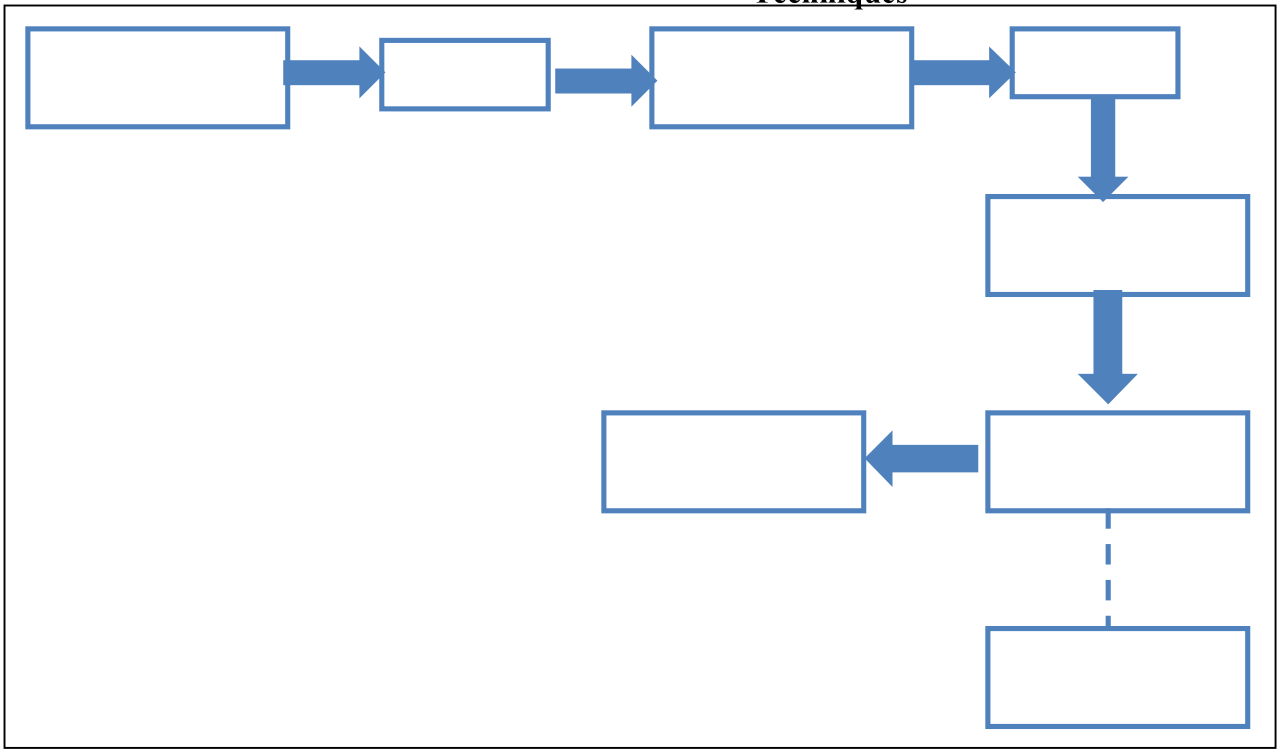


Figure 3.1 System Architecture

Figure 3.1 depicts the architecture of our system. As shown in the figure we have used the DICOM images acquired from the imaging scanners explained in Chapter 2 as input. The method that we have used for volume rendering is GPU-based ray casting where the ray casting algorithm takes 3D textures as input. Thus it was required to generate 3D textures of the DICOM images. Shaders are the basic functional units of GPU programming. We implemented new integration techniques to compute the frame rate required for volume rendering. The following subsections discuss each of these phases in more detail followed by the next chapter where we discuss the results.

3.2 Generation of Textures from DICOM Images

3.2.1 DICOM Image Format

Digital information management plays and important role in modern healthcare industry. As discussed by Graham et al. in [GPS05] the Digital Imaging and Communication in Medicine (DICOM) standard facilitates the communication of digital image information regardless of the device the images were acquired from. Digital images are generated by a wide variety of radiological hardware. Each device collects data, which are then encoded and stored electronically in DICOM format. Various radiological hardware developed by different manufacturers collect data in different manners; universal file type DICOM facilitates exchange between them.

Each DICOM file has a header containing amongst other items, patient demographic information, acquisition parameters, referrer, practitioner and operator identifiers and image dimensions. The remaining portion of the DICOM file contains the image data. Because they often contain multiple high-resolution images, DICOM files tend to be large and are frequently compressed before storage and transfer.

Figure 3.2 shows how the DICOM images make it easy to exchange image information acquired form various image equipment. The images can also be stored in a database from where they can be queried or retrieved as per the requirement.

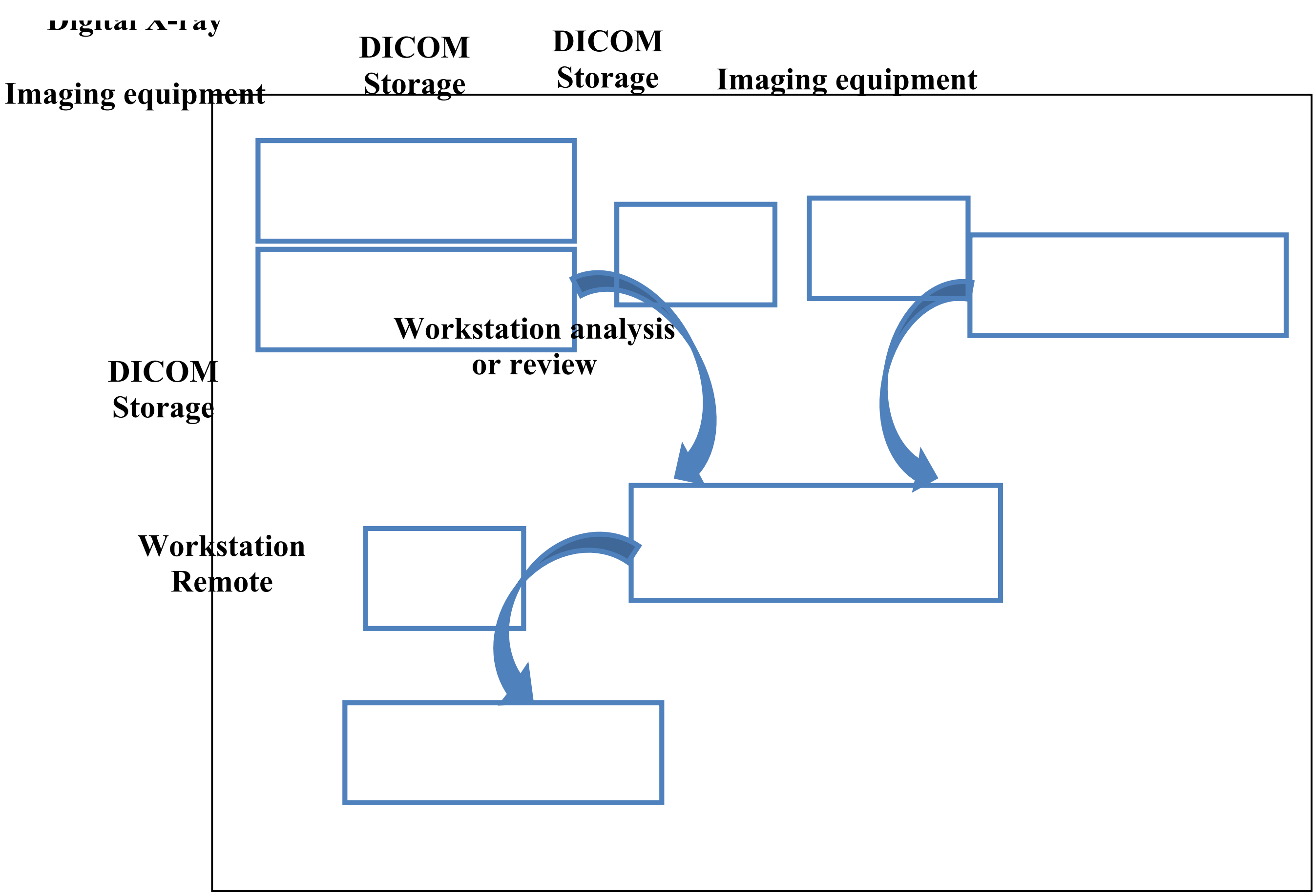


Figure 3.2 Information Exchange using DICOM Images

3.2.2 Generation of Textures

Based upon the application textures can be one, two or three-dimensional. Since the DICOM images used as the input for our system contained 3-dimensional data, we have used 3D textures for our implementation. As the volume data increases along the Z-direction, each image slice is a 2D image along X and Y-direction. The datasets are discussed in more detail in the following chapter. Each image in the series is used to create one texture slice. The stack of these textures looked as shown in the following Figure 3.4. These images contain textures generated for a selection of image slices in the dataset. We used two datasets for testing our implementations namely the skewed head dataset of Figure 3.4 and lobster dataset of Figure 3.5.

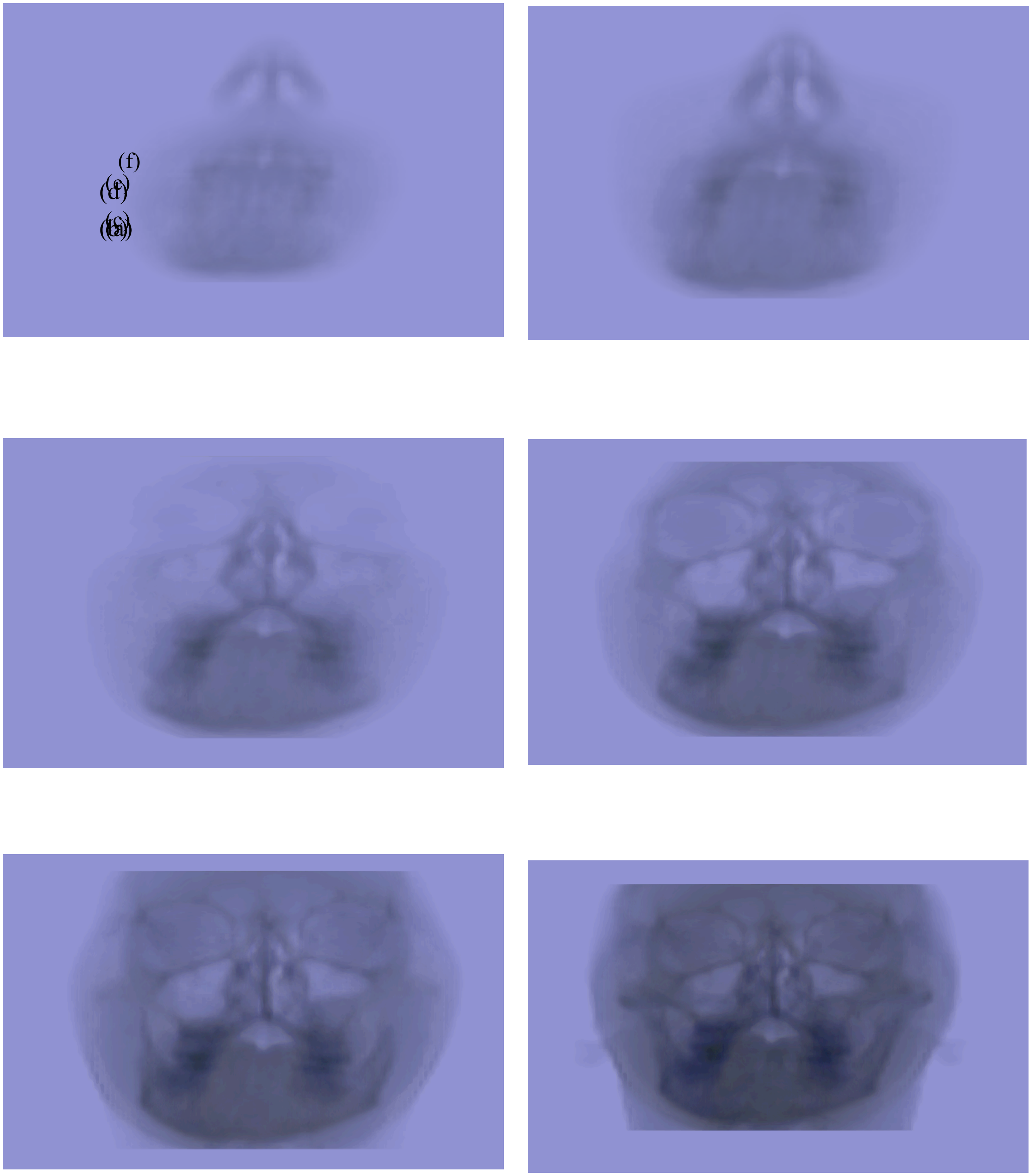


Figure 3.3 Texture Slices for Skewed Head Datased in an Increasing Order from (a) to (f)

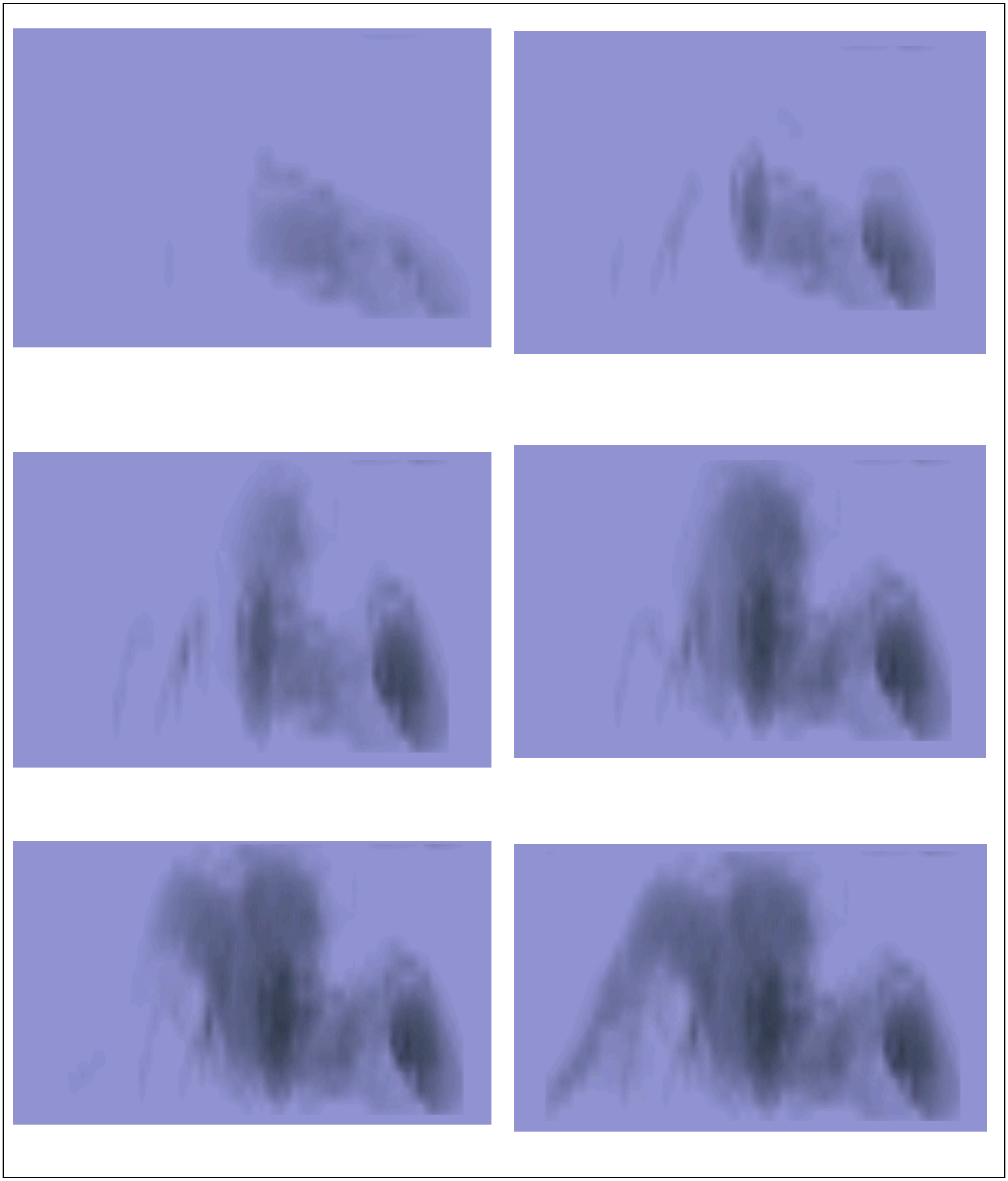


Figure 3.4 Texture Slices for Lobster Dataset an Increasing Order from (a) to (f)

3.3 Texture Based Ray-Casting for Volume Generation

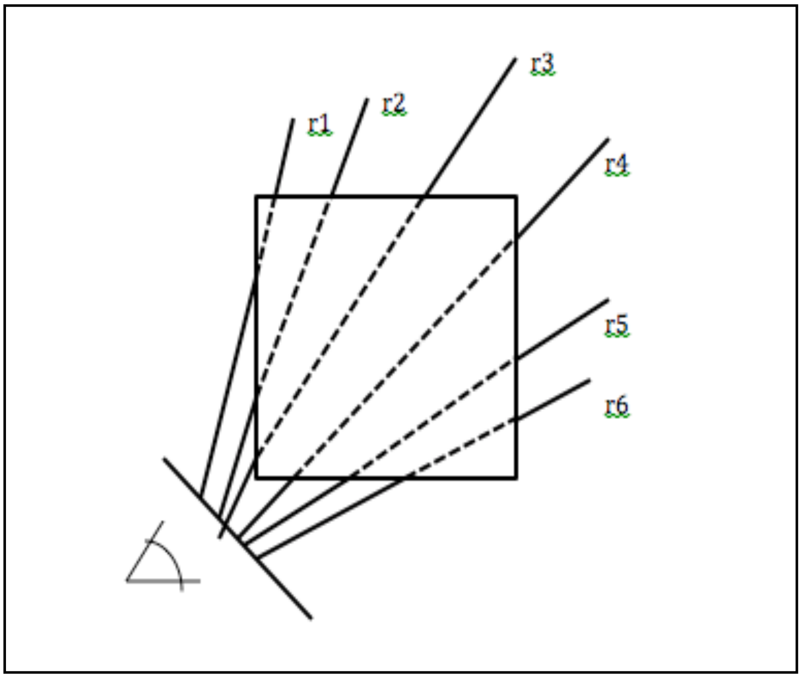


Figure 3.5 Ray Casting Principle, One Ray per Pixel

Our implementation involves commonly used ray-casting approach. As explained in the previous chapters the basic idea behind ray casting is to evaluate the volume rendering integral along rays that are traversed from the camera into the volume data. A single ray corresponds to one pixel in the images. In order to accumulate light information along the ray we need the optical properties of the image pixels. Using the transfer function, scalar data values given in the volume data are mapped to these optical properties. This accumulation involves compositing chromaticity and opacity values computed at each sample location along the ray. Following equations are normally used to compute the color and opacity values along the ray:

$$C_{dst} = C_{dst} + (1 - \alpha_{dst}) C_{src}$$
$$\alpha_{dst} = \alpha_{dst} + (1 - \alpha_{dst}) \alpha_{src}$$

```
Determine volume entry position
Compute ray direction
While (ray position in volume)
     Access data value at current position
     Compositing of color and opacity
     Advance position along ray
End while
```

The implementation uses the following pseudocode as explained by Engel et al. [EHK* 06] for ray casting:

We used 3D textures to store our volume data as explained in the previous subsection. In order to make efficient use of graphics hardware, we created vertex shader and fragment shader that implemented the volume ray-casting algorithm.

3.4 Shader Programs

Figure below shows the basic stages of the pipeline through which an application goes in order to render an output onto the screen. As depicted in the figure the Vertex, Geometry and Pixel processing stages are the programmable units of GPU while the Rasterizing and Output stage are fixed functional units.

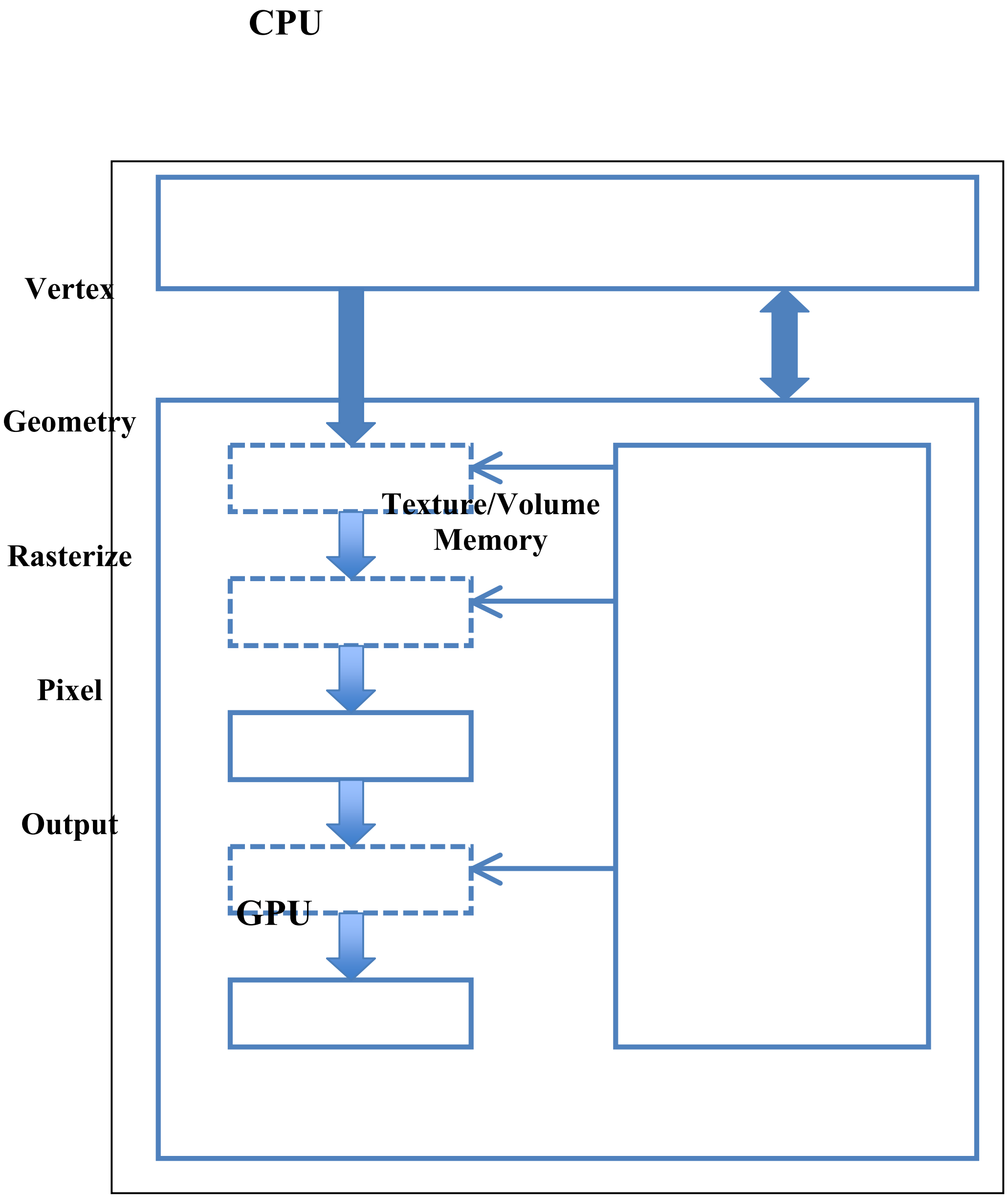


Figure 3.6 Fixed Functional and Programmable Units of a GPU Pipeline

Our implementation involves these stages that are written in two shaders namely Vertex Shader and Fragment Shader.

3.4.1 Vertex Shader Program

A vertex shader is the first stage in GPU pipeline and is responsible for transforming the position of each vertex according to the current view. In the vertex shader we implemented the vertex transformations as explained in the first stage of the pipeline. Vertex shader computed the eye location in the object space in order to transform the vertices in the object coordinate system. Since the volume data was stored in 3D texture, which was already generated, no change in the vertices was needed to be done in vertex shader, we simply passed the vertex position to fragment shader. The bounding box of the volume was rendered as *proxy geometry* with the dimensions equal to the (*x, y, z*) dimensions of the volume data. The *proxy geometry* in the form of a bounding box helped us render the volumes in fragment shader phase.

3.4.2 Fragment Shader Program

The fragment shader in the GPU pipeline is responsible for computing color of each rendered pixel. The ray casting algorithm is implemented in the fragment shader for which the shader steps along the ray through the volume to determine the pixel color. Our fragment shader program involved the implementation of the ray-casting algorithm. As required by the ray-casting algorithm, we defined and initialized the maximum number of steps to be taken along the ray. We also specified a point to access the volume data from the 3D texture.

The implementation of the ray-casting algorithm starts with computing the ray origin and direction. Ray origin is computed based upon position of the vertex as defined by the vertex shader and the size of the volume data. Ray direction is computed based upon the origin of the ray and the eye location given by the vertex shader. The eye position affects the ray direction thereby affecting the volume being rendered which is discussed in next chapter in more detail.

We then computed the single ray step. For this computation we deal with two types of steps namely step in the volume and the step in the optical space. The step in the volume was calculated based upon the size of the ray step and the size of volume. Whereas the step in the optical space is computed based upon the step in volume space and an optical scale. We also define an exit point for the ray. The exit point is computed based upon the smaller of following two values: Value of the last coordinate on the axis parallel to which the ray was traversed or the smallest coordinate value of the remaining two axes.

Once these factors affecting the bounding conditions of the ray are computed we implement the volume ray-casting loop. For the given number of steps and the ray that is inside the volume; we compute the color and opacity at each ray-step. This computed value is then added to the final resulting color and opacity value that ultimately gives us the composited color and opacity value for the volume being rendered.

This implementation gave us a framework for simple GPU-based ray casting. In order to implement accelerated volume rendering of the input images our goal was to implement new integration techniques. The derivation and implementation of these techniques is discussed in the following subsection of the chapter.

3.5 Numerical Integration Techniques

We have mentioned in the thesis contribution subsection that one of the goals of our work was to render the volumes with higher frame rates. We implemented better integration techniques in order to obtain a significant performance gain. This chapter discusses the evaluation of these integration techniques.

In a ray-casting algorithm we cast a ray through the volume data and at each sample interval we compute the color and opacity value of the image pixel. When the ray advances to next sampling interval we compute the color and opacity value for the pixel and add it to the previous value. Thus a linear computation of color and opacity values is done. This integration is thus given a name *linear integration.*

Our implementation involved adapting to the sizes of the ray inspired from the discussion by Philip J Davis and Philip Rabinowitz [DR07]. This adaptation was performed based upon the changes in the volume data. Thus if there are small changes in attributes of the volume data then bigger steps were taken while for more interesting changes the step size was reduced. Thus in our work we computed the step size which adapted to the changes in the volume data. There were two factors that needed to be considered:

1) When to change the step size?

2) How to change the step size?

These two factors are discussed in the following two subsections.

3.5.1 Relative Error Computation

In this subsection we deal with the first question above. In order to adapt to changes in the volume data we need to know when should the step size be incremented and when should it be decremented. We computed a relative error, which was then compared with a threshold in order to see whether more interesting changes are happening, or not.

We computed the relative error based on color values computed in two consecutive ray steps. Consider C_{old} to be the color value at previous step and C_{new} to be the color value at current step. Then we first computed the absolute color values based upon the RGB components as follows:

$$C = \sqrt{C_r^2 + C_g^2 + C_b^2} \tag{3.1}$$

Based upon these absolute values we computed the difference between them as follows:

$$relative\ error = \frac{|C_{old} - C_{new}|}{C_{new}} \tag{3.2}$$

The relative approximation error was at first independent of opacity. In order to adapt the sampling as the opacity increases we multiplied the *relative error* by (α_i) where α_i is the opacity at i^{th} sampling step. We then defined a *threshold* ε to be the limiting value with which this relative error was compared. Thus the answer to the first question above in a mathematical form was:

$$\frac{|C_{old}-C_{new}|}{C_{new}}(1-\alpha_i) < \varepsilon < \frac{|C_{old}-C_{new}|}{C_{new}}(1-\alpha_i) \tag{3.3}$$

This equation computes the relative error and compares it to the threshold. The threshold was kept user-defined in order to be dependent upon the dataset.

3.5.2 Adaptation of Step Sizes

Once we know when should we change the size of the ray-step, the next question that needs to be dealt with is how to change it? We implemented two types of algorithm each of which had two separate sub-types. Figure below shows a tree diagram of our implementation algorithms.

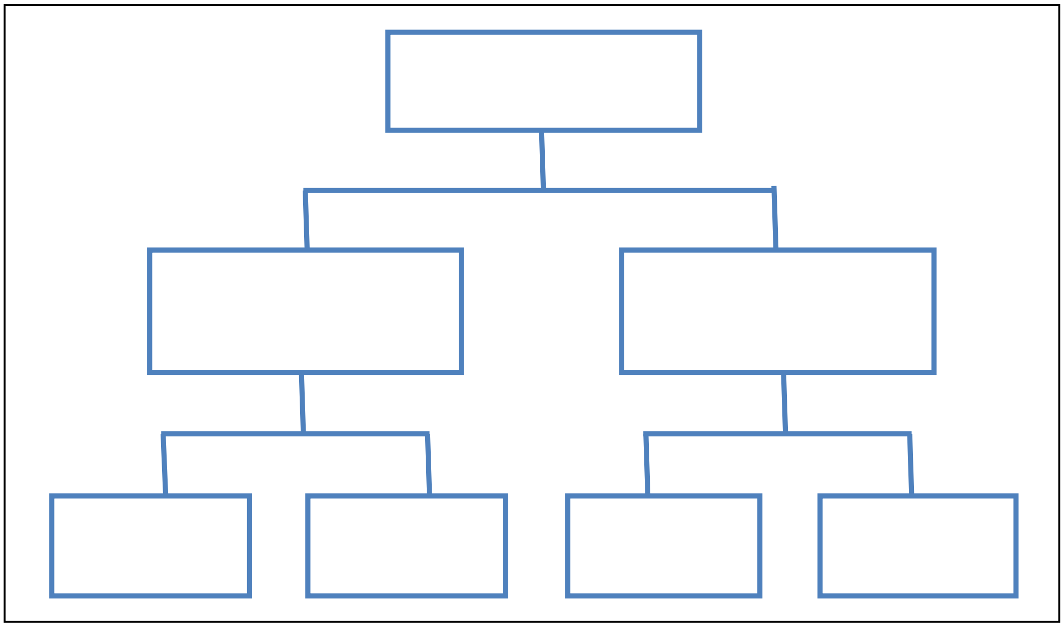


Figure 3.7 Hierarchy of Integration Techniques Implemented

1) Comparison of Level Values: As discussed above we know that for a ray-casting algorithm we have defined the number of steps the ray should take in the volume. Thus for the first ray-step we have compute the size of the step and the corresponding chromaticity and opacity values for first sampling pixel. Let's store these values in C_{old} and α_1. Now in order to compute the relative error we need to increment the step to next sampling position. Thus we

move on to the next step and compute the color and opacity values at this pixel sampling position. Let's call these values C_{new} and α_2. We then compute the relative error using equations (1) and (2) above. Thus in this type of integration technique we compared the change in volume data based on each of the levels in integration.

Next, we perform the comparison with the threshold. For the given threshold we compare the relative error to see if it is greater or less than the threshold. If the relative error is less than the threshold it indicates that not much interesting changes are happening in the volume data and thus we can increase size of the ray-step; while a greater relative error indicates that there are interesting changes going on in the volume data and thus we take smaller steps. These changes in the step size are either linear or scaling as explained below:

i. Linear Changes: As explained above for a relative error less than the threshold, the step size needs to be incremented. Linearly, this was done by incrementing the ray-step size with a constant factor for example 0.5. Thus our equation looked like raystep += 0.5. When the relative error was greater than the threshold the step size was linearly decremented by using raystep -= 0.5. The choice of the incrementing and decrementing factor is discussed in next chapter. We also needed to prevent the step size from incrementing or decrementing linearly such that it does not reach a value that was not acceptable. To achieve this we limited the increase as well as the decrease in these values by a certain defined number.

ii. Scaling Changes: Similar to the linear changes, the step size could also be changes by scaling up or down. For incrementing the step size when not much interesting changes were going on in the volume data we used equation that looked like raystep *= 0.5 while when more interesting changes were seen the raystep was reduced by the equation raystep /= 0.5.

2) Comparison of Integrated Values: In this type of integration technique we defined C_{old} and C_{new} to be the integrated values. Consider for instance that we are at 7th step in the volume data. We have the integrated color value for steps 1-6 stored in C_{old} while the color value computed for steps 1-7 are stored in C_{new}. We then compute the *relative error* using equations (3.1) and (3.2) above and then compare it with the threshold. Similar to the level comparison, we change the step size linearly or by scaling up and down with a predefined factor. This type of integration would be beneficial when the ray gets deep in the volume, where errors in integration may be less visible allowing the possibility of taking larger steps in case of step-value comparison.

3.5.2 Numerical Integration Techniques Based on Poisson Distribution

The other stream of our implementation involved changing the number of samples in a single raystep based upon the changes in the color and opacity between two consecutive raysteps. This was implemented using the Poisson distribution algorithm, which will be discussed in detail in this subsection of the chapter.

Poisson distribution was first introduced by Simeon Denis Poisson and involves finding the probability of a number of events that can occur in a given time or space based upon a previous knowledge. We used this algorithm in our work to determine the number of samples in one given raystep based upon the previous knowledge of how frequently the color and opacity between two raysteps change. The result of this algorithm was in the form of number of samples in a given raystep for which the resulting color and opacity values were then computed.

As discussed by Knuth [K69], we implemented the following algorithm to achieve Poisson distribution algorithm for finding the number if samples in a given raystep.

```
Init: L = $e^{-\lambda}$, k = 0, p = 1
do
     k = k + 1
     generate random number u in [0, 1]
     p = p * u
while p > L
return k – 1
```

The algorithm takes as initial values in the form of *L*, *k* and *p* where, *k* represents the number of samples in the given raystep. Our implementation based on the similarity of the color values of the pixels in the consecutive raysteps and thus to make the Poisson algorithm implementation proportionate to this difference, we initialized *L* to $e^{-\lambda}$ where, λ represented the difference between the color values of pixels in the consecutive raysteps. Thus, similar to our earlier implementations we first computed C_{old} and C_{new} for the first two raysteps. Then we calculated the difference between these two values and assigned it to λ. *k* and *p* were initialized to 0 and 1 respectively. The output of Poisson implementation was used as the number of samples in a given raystep for which the color and opacity was recursively computed and assigned to the final volume.

1) Computation of Random Numbers: Another important factor in the Poisson distribution is the way random numbers are generated. These random numbers work as a multiplier, which ultimately helps in determining the output of the Poisson distribution. We used Linear Congruential Generator method to compute the random numbers. Linear congruential generator method provides a set of pseudo-random numbers in the given range using the following formula:

$$I_k = (aI_{k-1} + c) \bmod m \qquad (3.4)$$

where, *a, c* and *m* are the pre-selected values for multiplier, increment and modulus respectively. *I* is called as the seed value and each I_k gives a random number in the given range. The effectiveness of LCG depends largely upon the initialization values of *a, c* and *m.*

After trying a set of values as discussed by Knuth [K97], we initialized *a, c, m* to 1664525, 1013904223, and 2^{32} respectively. Since we needed to compute the number of samples in the single raystep, we divided *a, c, m* by a uniform value of 2^{32}. The output of this implementation was the multiplier used in Poisson distribution.

3.6 Implementation in LAB Color Space

In the previous subsection we discussed numerical integration techniques for RGB Color Space. To compare our integration techniques with more minute details, we also implemented these techniques for LAB Color Space. The LAB color space as compared to RGB color space represents L channel for lightness and the A and B channels for the color opponent dimensions.

We used following standard equations to convert the RGB color space to XYZ and then XYZ to LAB. This color in LAB space was then used to compute the final result value.

$$C_{xyx.x} = C_{rgb.r} * 0.4124 + C_{rgb.g} * 0.3576 + C_{rgb.b} * 0.1805 \tag{3.5}$$

$$C_{xyx.y} = C_{rgb.r} * 0.2126 + C_{rgb.g} * 0.7152 + C_{rgb.b} * 0.0722 \tag{3.6}$$

$$C_{xyx.z} = C_{rgb.r} * 0.0193 + C_{rgb.g} * 0.1192 + C_{rgb.b} * 0.9505 \tag{3.7}$$

where, C_{xyz} and C_{rgb} indicate colors in XYZ and RGB color space respectively.

$$C_{lab.l} = 116 * (\sqrt[3]{C_{xyz.y}} - 16) \tag{3.8}$$

$$C_{lab.a} = 500 * (\sqrt[3]{C_{xyz.x}} - \sqrt[3]{C_{xyz.y}}) \tag{3.9}$$

$$C_{lab.b} = 200 * (\sqrt[3]{C_{xyz.y}} - \sqrt[3]{C_{xyz.z}}) \tag{3.10}$$

where, C_{xyz} and C_{lab} indicate colors in XYZ and LAB color space respectively.

Chapter 4: Results

In this chapter we discuss the results of our implementation. We start with discussing the datasets we have used, the hardware on which we performed our experiments and conclude the chapter with the comparisons of performance gains and image quality acquired.

4.1 Datasets and Hardware Used

The datasets that we used for our implementation were taken from the webpage of Osirix DICOM Image Viewer software. The sizes of the datasets were in the range of 5MB to 300MB. We selected the datasets of sizes in the range of 100MB to 200MB. The DICOM images were read by the Grassroots DICOM image library (GDCM) and then the image data was stored in a data file (.dat) to gain simplicity in accessing it. The results discussed in this chapter are rendered on a NVIDIA GeForce 320M graphics processor with 256MB of DDR3 SDRAM.

We needed to design a way in which the images acquired by our implementation could be compared for quality with the original images. We devised two ways for the image quality comparison. The first one was a graphical comparison for which, we developed a MATLAB code to find the difference between the two images. We frequently use these difference images to illustrate the image quality of our implementation in the remaining of this chapter. The second stream of image quality comparison involved quantitative analysis for which we used Structural SIMilarity (SSIM) index. As discussed by Zhou Wang et al. in [WBSS04] SSIM computes the difference between two images based on the statistical features of the images, image distortions, viewing distances for the images and localized quality measurements. The implementation of SSIM algorithm in the form of DSSIM toolkit by Pornel gave us a numerical representation of the image quality, which is also presented in this chapter. Based on the SSIM algorithm implementation, a SSIM index of 0 represents no difference between the two images being compared while as the index increases more difference between the two images is found. Thus, given to our implementation, a SSIM index value close to 0 indicated the similar quality of the output image to that of the original image and as the index value increased, the image quality was considered to be deteriorated.

In order to compare the results generated by each type of integration technique some interactivity in the system was needed. We implemented this interactivity in the form of a run-time change in shaders containing different integration techniques. This resulted in smooth transition between shaders.

4.2 Comparison of Results

4.2.1 Level Value Comparison (Linear Changes)

In case of a level value comparison for a *relative error,* which was less than the threshold, we increased the step size linearly with a factor of 1. In order to avoid the linear increment to reach infinity, we limited the maximum step size value to 6.0. We gained a significant amount of frames per second for the same quality of the images as that of the ones produced by the original integration technique. These images and the performance gain are depicted in the following diagrams.

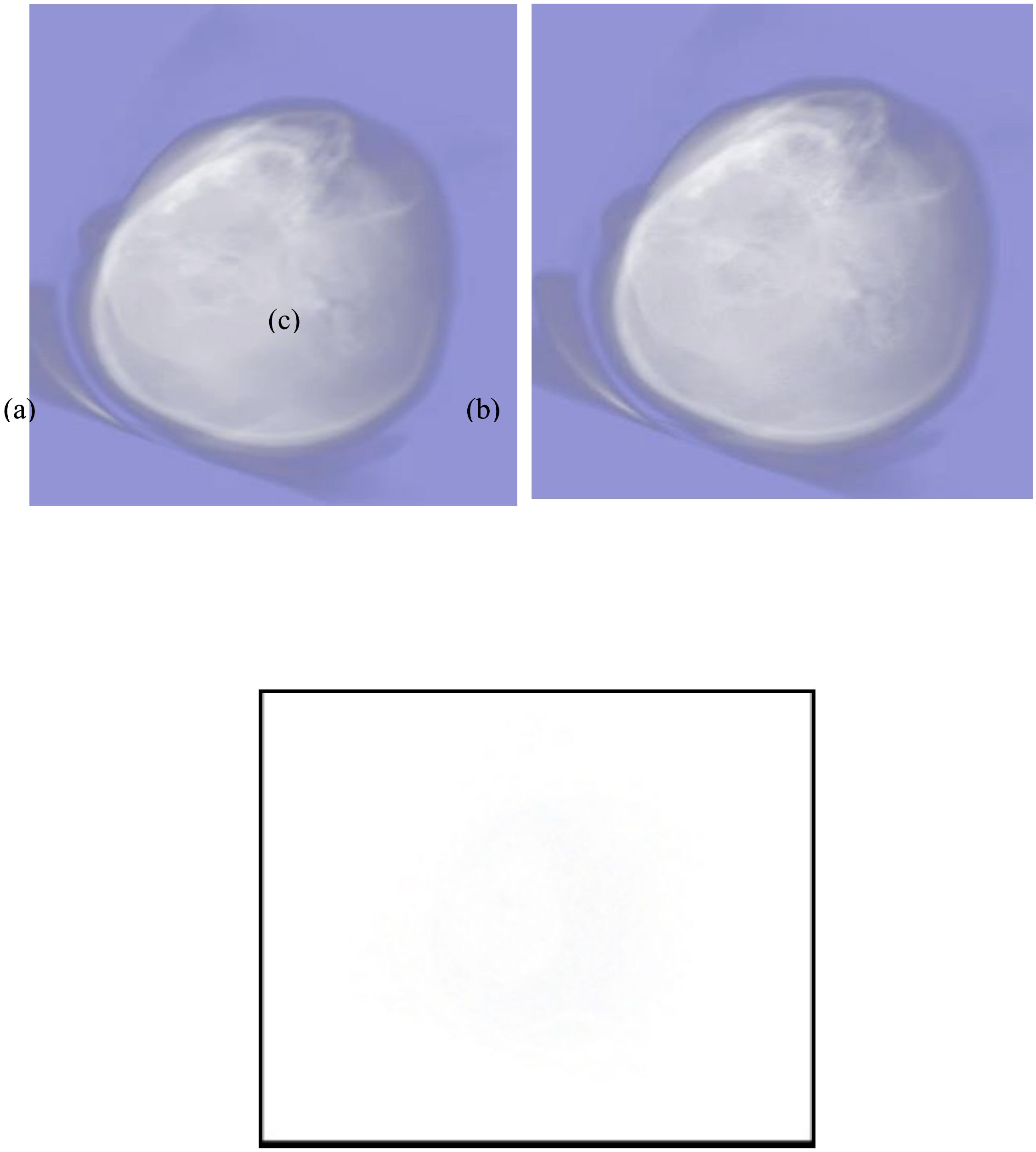
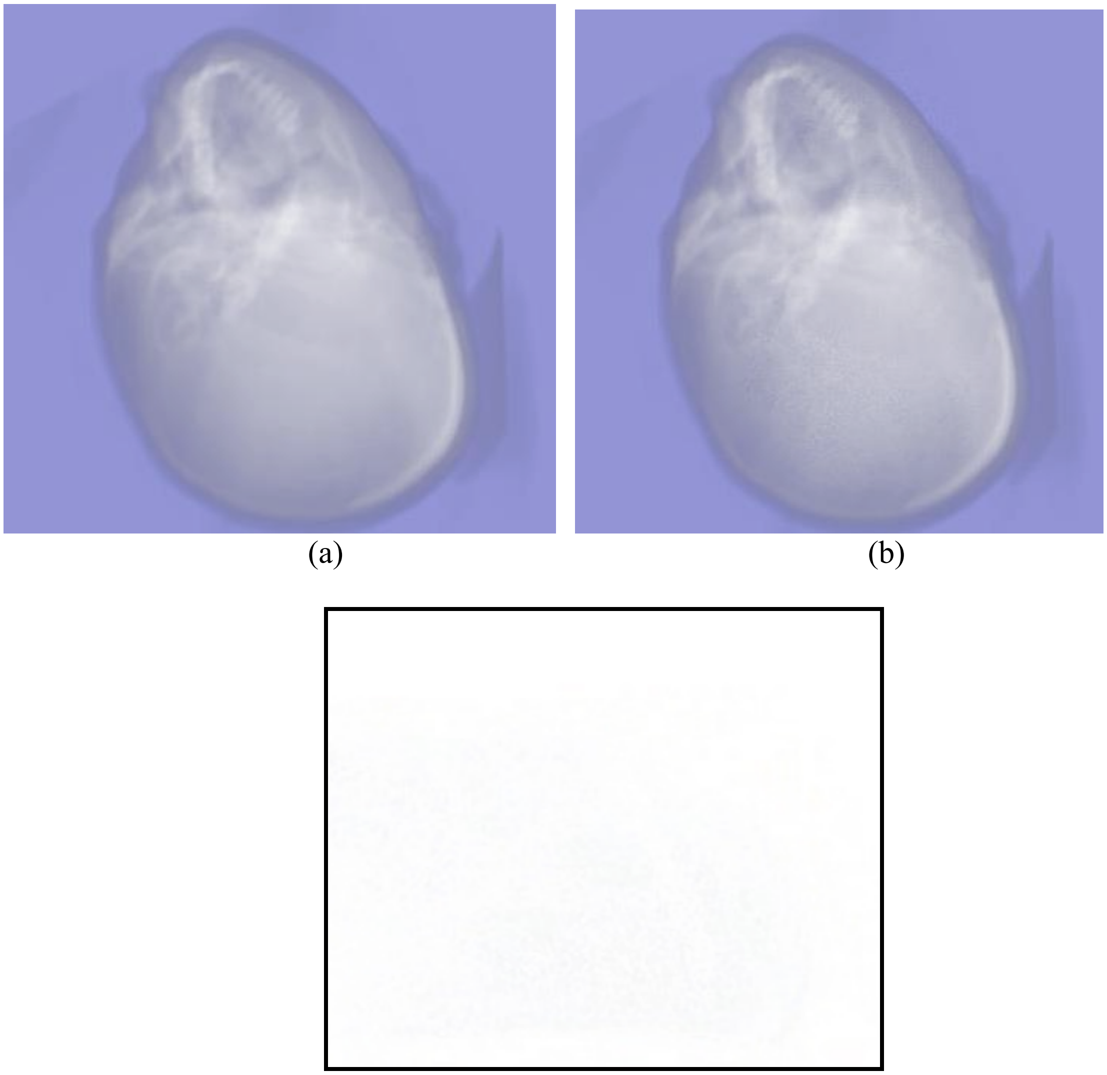


Figure 4.1 Level Value Comparison (Linear Changes): (a) Original Image (b) Shader Implementation (c) Difference Image [Dataset: Skewed Head]



(a) (b)

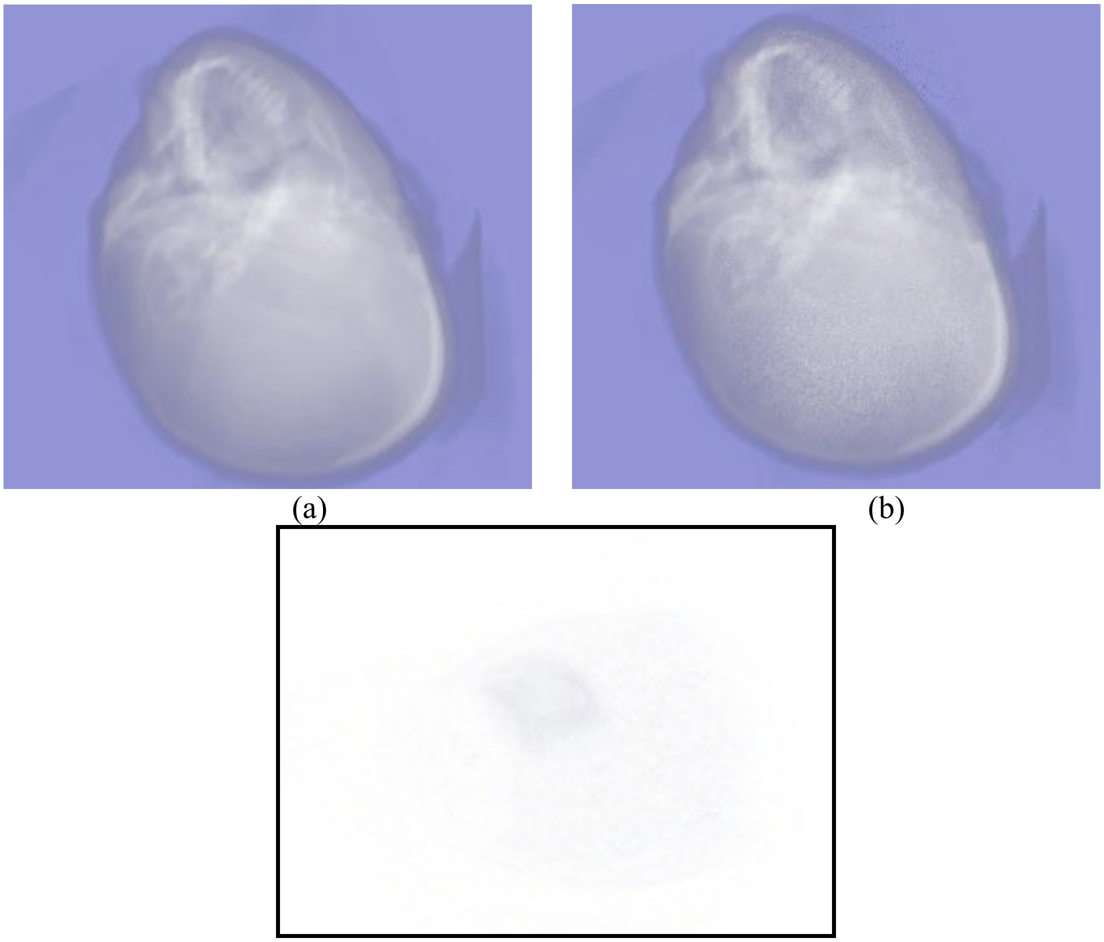
(c)

Figure 4.2 Level Value Comparison (Linear Changes): (a) Original Image (b) Shader Implementation (c) Difference Image [Dataset: Skewed Head]

Dataset	Size	Dimensions	Threshold (ε)	FPS	SSIM Index
Skewed Head	120 MB	184x256x170	0.03	60	0.0067
Lobster	27 MB	120x120x34	0.03	548	0.0004
Present	69 MB	246x246x221	0.03	27	0.002

Table 4.1 Performance Table for Level Value Comparison (Linear Changes)

Figures 4.1 and 4.2 show the images taken from a single viewpoint. The first two parts of the figures, (a) and (b), show the original image and the output image after the implementation of the linear integration. These two figures show the images when the raysteps were incremented and decremented by a factor of 0.1 and 1.0 respectively. The SSIM index values for these two comparisons were 0.0067 and 0.0006 respectively. Since these index values are almost close to 0, it shows that there was no significant difference between the two images for a performance gain as shown in the table 4.1. However, figure 4.3 below shows that when the raystep was incremented and decremented by a factor of 2.0 some difference in the two images with a SSIM index of 0.0326 was found for the same amount of performance gain.


(a) (b)

(c)

Figure 4.3 Level Value Comparison (Linear Changes): (a) Original Image (b) Shader Implementation (c) Difference Image [Dataset: Skewed Head]

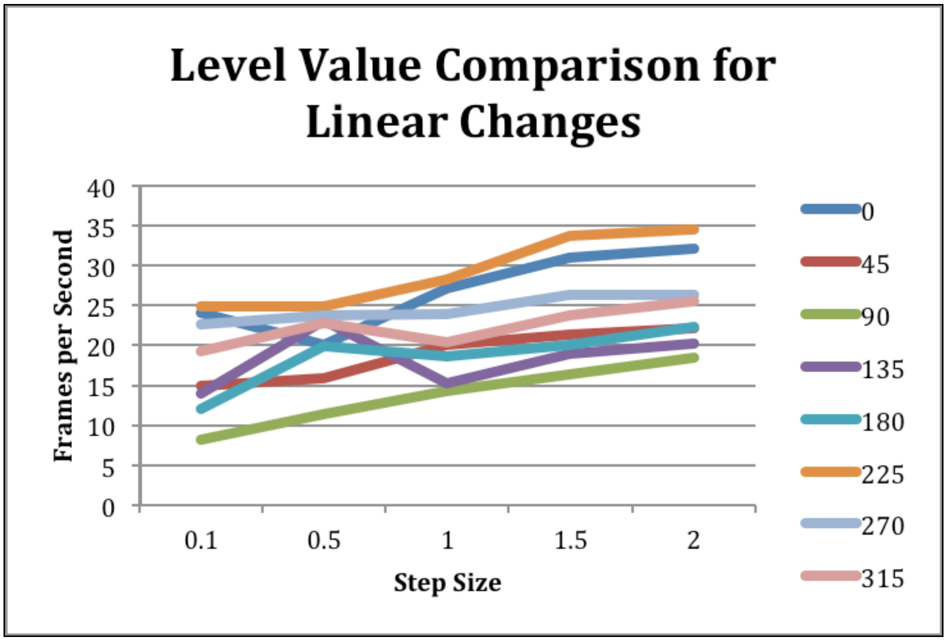


Figure 4.4 Performance Graph for Level Value Comparison (Linear Changes)

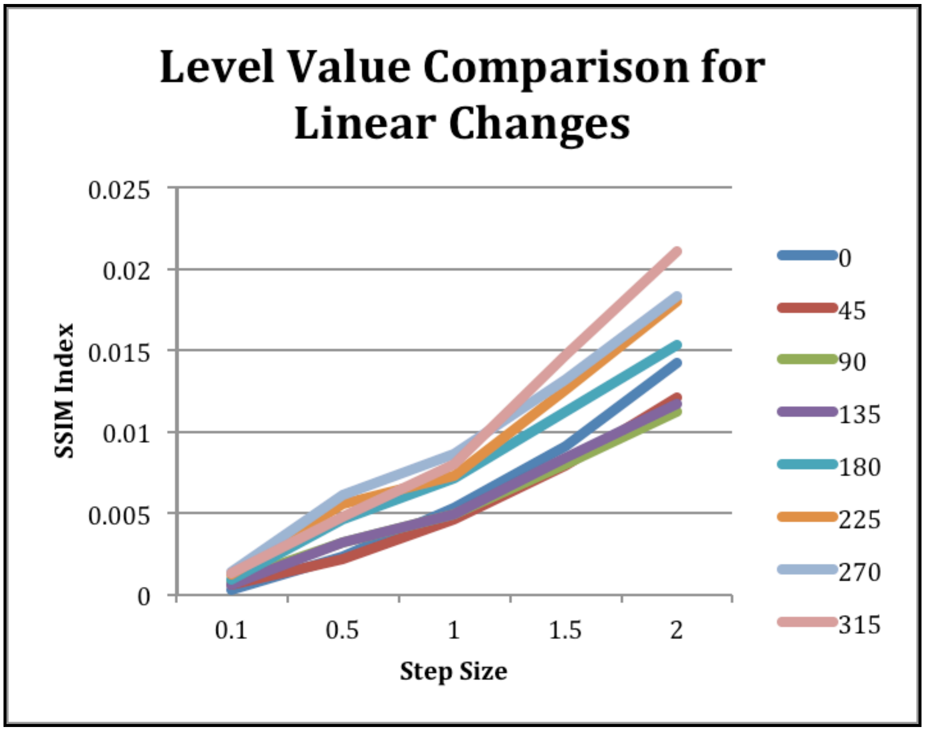


Figure 4.5 Image Quality Graph for Level Value Comparison (Linear Changes)

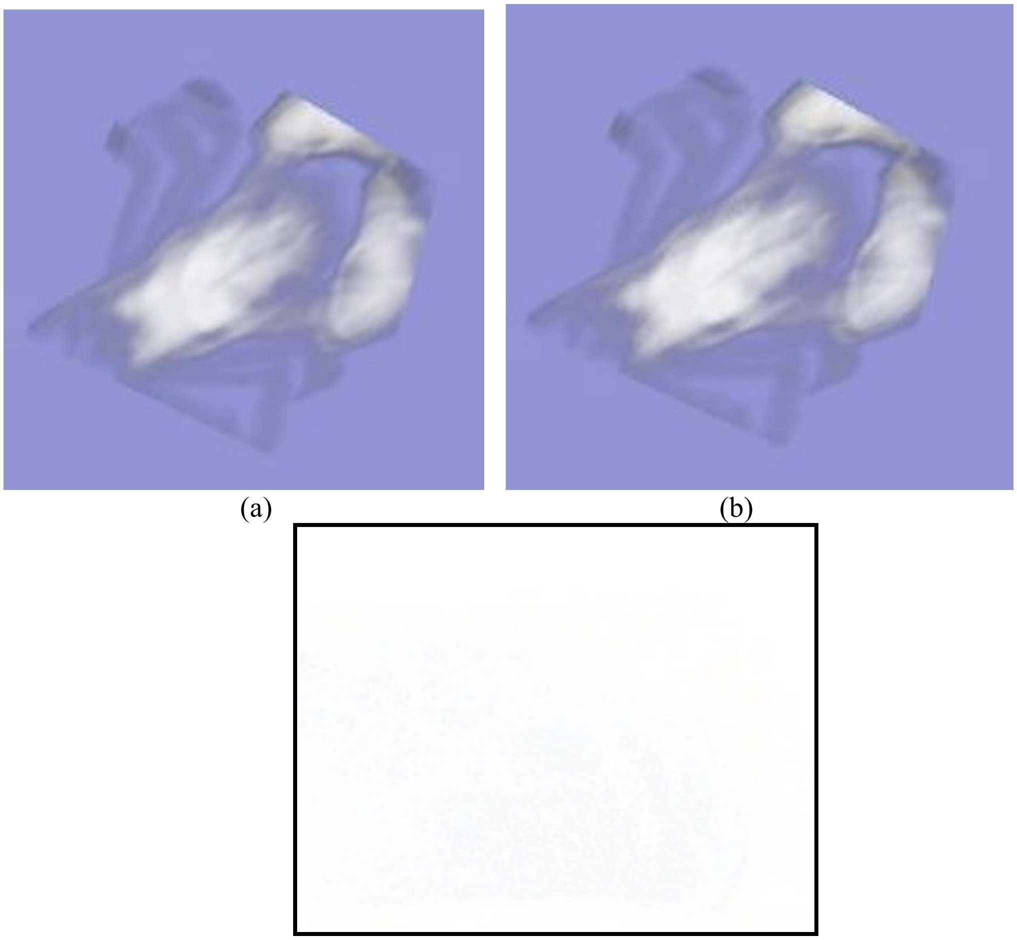
Figure 4.4 shows a graph of the performance in the form of frames per second against the different step size values used for linearly changing the ray step. Different lines in the graph represent the different angles at which the images were taken. This graph shows that when the step sizes were linearly changed within a range of 1 to 2 we found a significant performance gain compared to the frame rate of original implementation (17 fps). The graph in Figure 4.5 indicates the image quality in the form of SSIM index at different step sizes. As can be seen from this graph when the step sizes were linearly changed in the range of 0.1 to 1 we found the images, which were close to the original image. The combination of these graphs can lead to a conclusion that when the step sizes were linearly changed within a range of 0.1 to 1.5 we found a significant gain of upto 20 frames per second without losing image quality in the form of SSIM index.

4.2.2 Level Value Comparison (Scale Changes)

For a *relative error, which* was greater than the threshold, we decreased the step size linearly with a factor of 0.5. In order to avoid the linear decrement to go below 0, we limited the minimum step size value to 0.0. We gained a significant amount of frames per second for the same quality of the images as that of the ones produced by the original integration technique. These images and the performance gain are depicted in the following diagrams.

Dataset	Size	Dimensions	Threshold (ε)	FPS	SSIM Index
Skewed Head	120 MB	184x256x170	0.03	50	0.0011
Lobster	27 MB	120x120x34	0.03	560	0.0121
Present	69 MB	246x246x221	0.03	36	0.0062

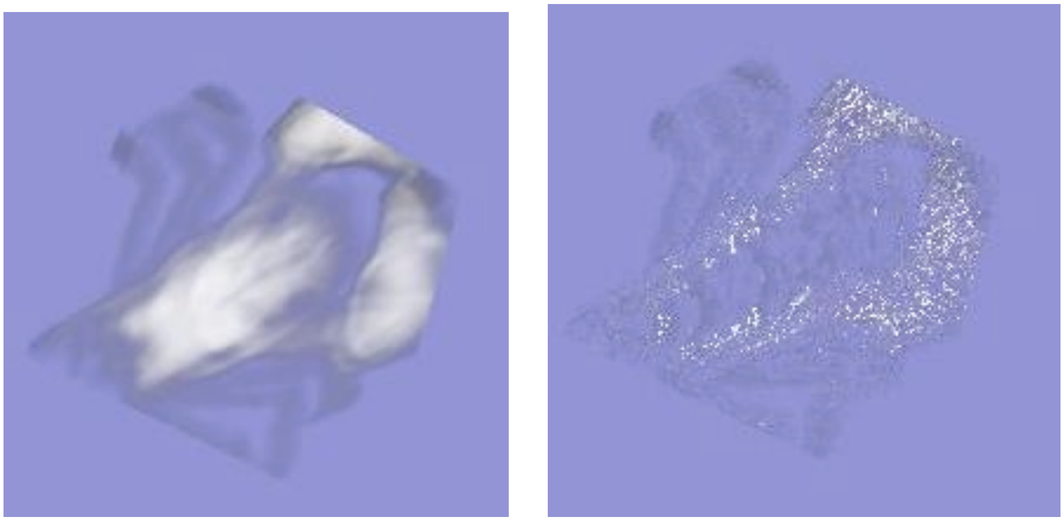
Table 4.2 Performance Table for Level Value Comparison (Scale Changes)



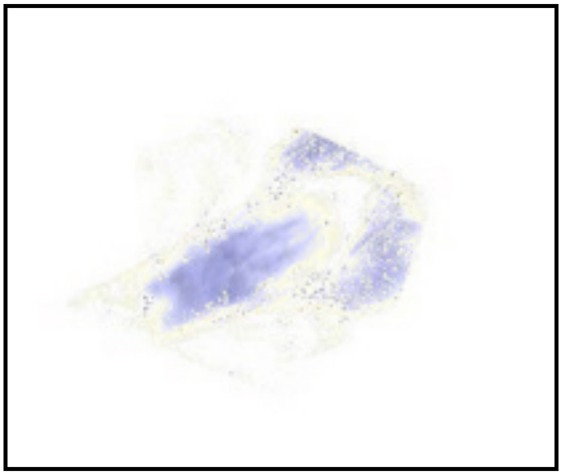
(a) (b)

(c)

Figure 4.6 Level Value Comparison (Scale Changes): (a) Original Image (b) Shader Implementation (c) Difference Image [Dataset: Lobster]



(a) (b)



(c)

Figure 4.7 Level Value Comparison (Scale Changes): (a) Original Image (b) Shader Implementation (c) Difference Image [Dataset: Lobster]

In order to evaluate the quality of images acquired using this integration method, we use the lobster dataset as shown in Figure 4.6. Figure 4.6 shows the output when the stepsize was increased and decreased with a scaling factor of 1.2 and 0.5 respectively. For this implementation we achieved a SSIM index of 0.0011. For a very high boost in the performance gain we got a deteriorated quality output when the stepsize was changed by a factor of 0.1 as depicted by the figure 4.7. At this higher performance gain the SSIM index was 0.2410. This performance gain vs. stepsize changing factor comparison is shown in the following graph of figure 4.8. Based on the following graph and the images shown in the previous and current subsections it can be seen that linear changes in stepsizes are more efficient than the scale changes.

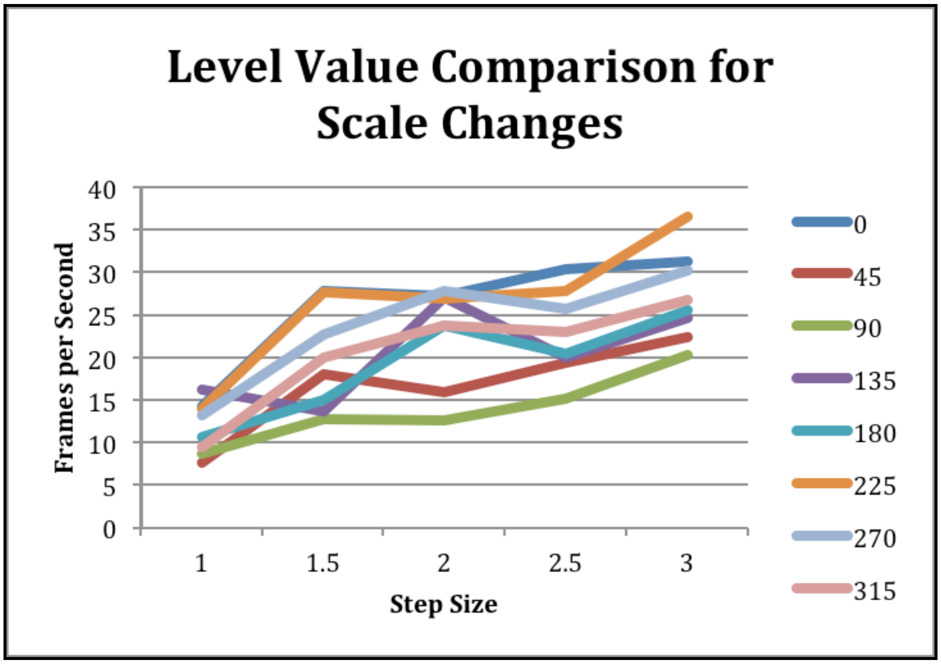


Figure 4.8 Performance Graph for Level Value Comparison (Scale Changes)

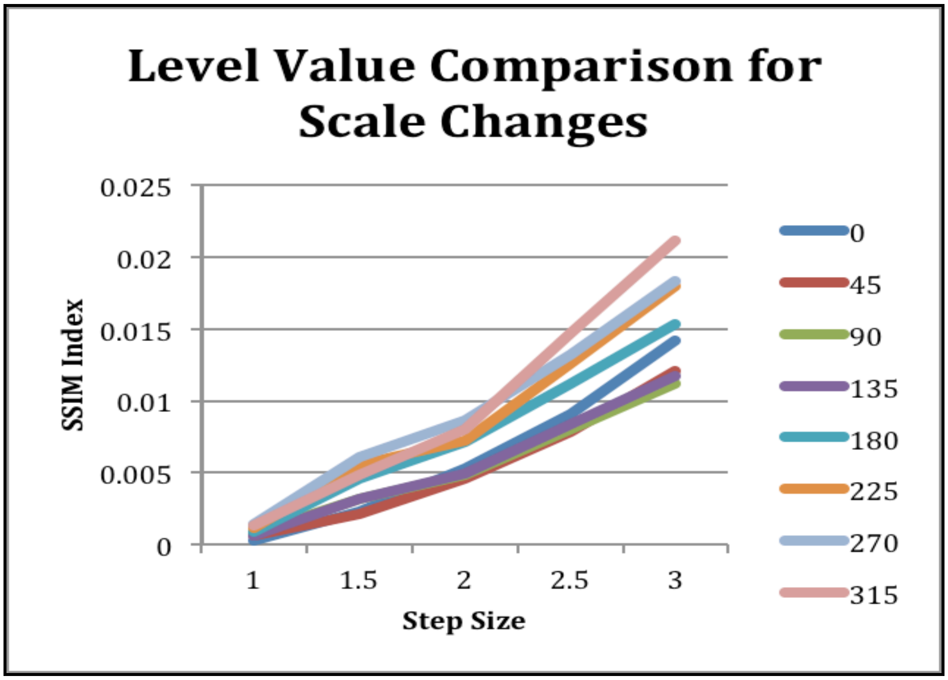


Figure 4.9 Image Quality Graph for Level Value Comaprison (Scale Changes)

Figure 4.8 shows a graph of the performance in the form of frames per second against the different step size values used for scaling the ray step. Different lines in the graph represent the different angles at which the images were taken. This graph shows that when the step sizes were scaled up or down within a range of 1.5 to 3 we found a significant performance gain compared to the frame rate of original implementation (17 fps). The graph in Figure 4.9 indicates the image quality in the form of SSIM index at different step sizes. As can be seen from this graph when the step sizes were changed in the range of 1 to 2 we found the images, which were close to the original image. This graph also shows that as the scaling factor increased, the image quality degraded for all viewpoints. The combination of these graphs can lead to a conclusion that when the step sizes were changed within a range of 1.5 to 2.5 we found a significant gain of upto 20 frames per second without losing image quality in the form of SSIM index.

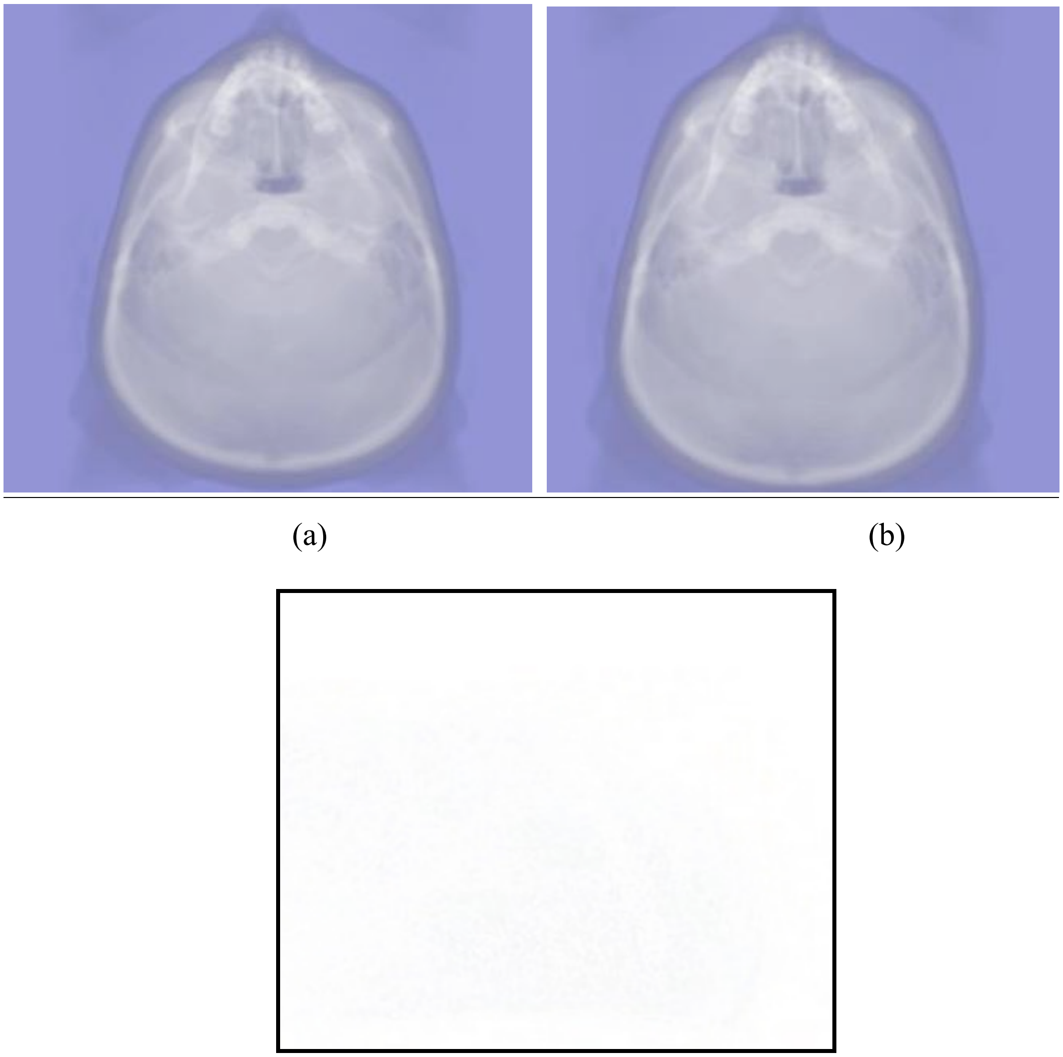
4.2.3 Integrated Value Comparison (Linear Changes)

We also discussed in previous chapter how the integrated value comparison was implemented. For a *relative error, which* was less than the threshold, we increased the step size linearly with a factor of 1. In order to avoid the linear increment to reach infinity, we limited the maximum step size value to 6.0. We gained a significant amount of frames per second for the same quality of the images as that of the ones produced by the original integration technique. These images and the performance gain are depicted in the following diagrams.

Dataset	**Size**	**Dimensions**	**Threshold (ε)**	**FPS**	**SSIM Index**
Skewed Head	120 MB	184x256x170	0.03	58	0.0003
Lobster	27 MB	120x120x34	0.03	420	0.0025
Present	69 MB	246x246x221	0.03	28	0.0002

Table 4.3 Performance Table for Integrated Value Comparison (Linear Changes)

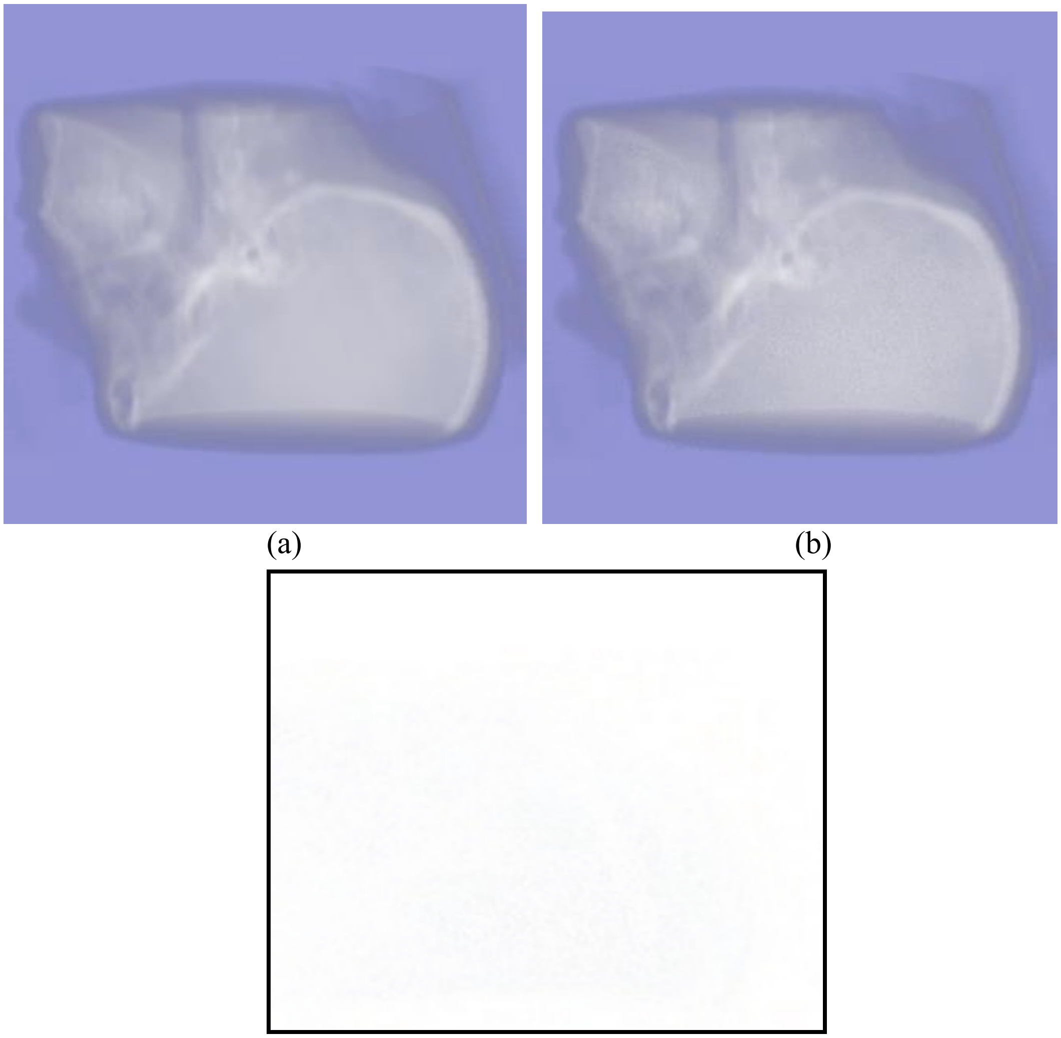
Figures 4.10 and 4.11 show the images taken from a single viewpoint. These two figures show the images when the raysteps were incremented and decremented by a factor of 0.5 and 1.0 respectively. SSIM indices for these two image comparisons were found to be 0.0003 and 0.0028 respectively. Since the figure (c) is blank and based on the SSIM index values, we can say that there was no significant difference between the two images for a performance gain as shown in the Table 4.3. As can be seen in the following figures, there was not much change in the results found when the comparison involved integrated values as opposed to the values at individual levels.



(a) (b)

(c)

Figure 4.10 Integrated Value Comparison (Linear Changes): (a) Original Image (b) Shader Implementation (c) Difference Image [Dataset: Skewed Head]



(a) (b)

(c)

Figure 4.11 Integrated Value Comparison (Linear Changes): (a) Original Image (b) Shader Implementation (c) Difference Image [Dataset: Skewed Head]

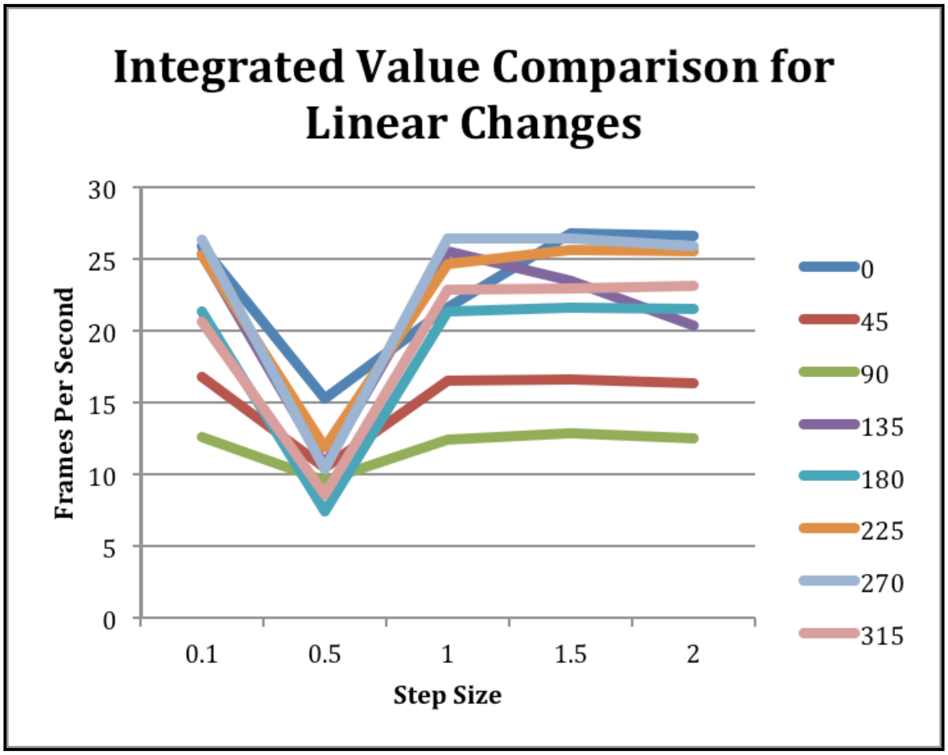


Figure 4.12 Performance Graph for Integrated Value Comparison (Linear Changes)

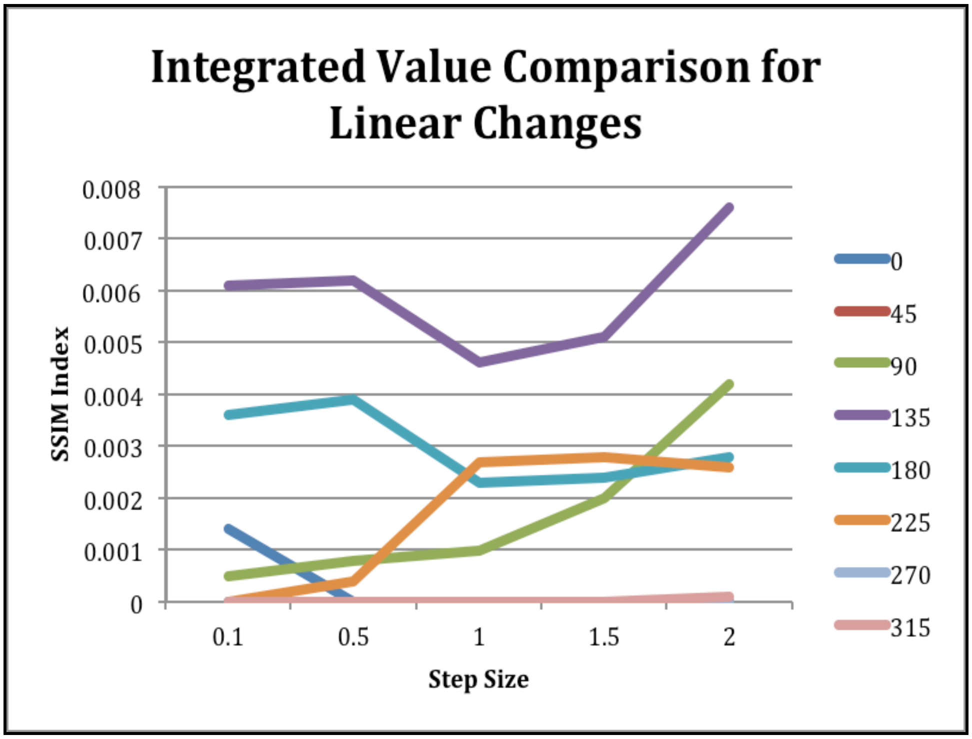


Figure 4.13 Image Quality Graph for Integrated Value Comparison (Linear Changes)

Figure 4.12 shows a graph of the performance in the form of frames per second against the different step size values used for linearly changing the ray step. Different lines in the graph represent the different angles at which the images were taken. This graph shows that we found a significant performance gain compared to the frame rate of original implementation (17 fps) except when the step sizes were changed with a factor of 0.5. The graph in Figure 4.13 indicates the image quality in the form of SSIM index at different step sizes. As can be seen from this graph for all step sizes we found the images, which were close to the original image.

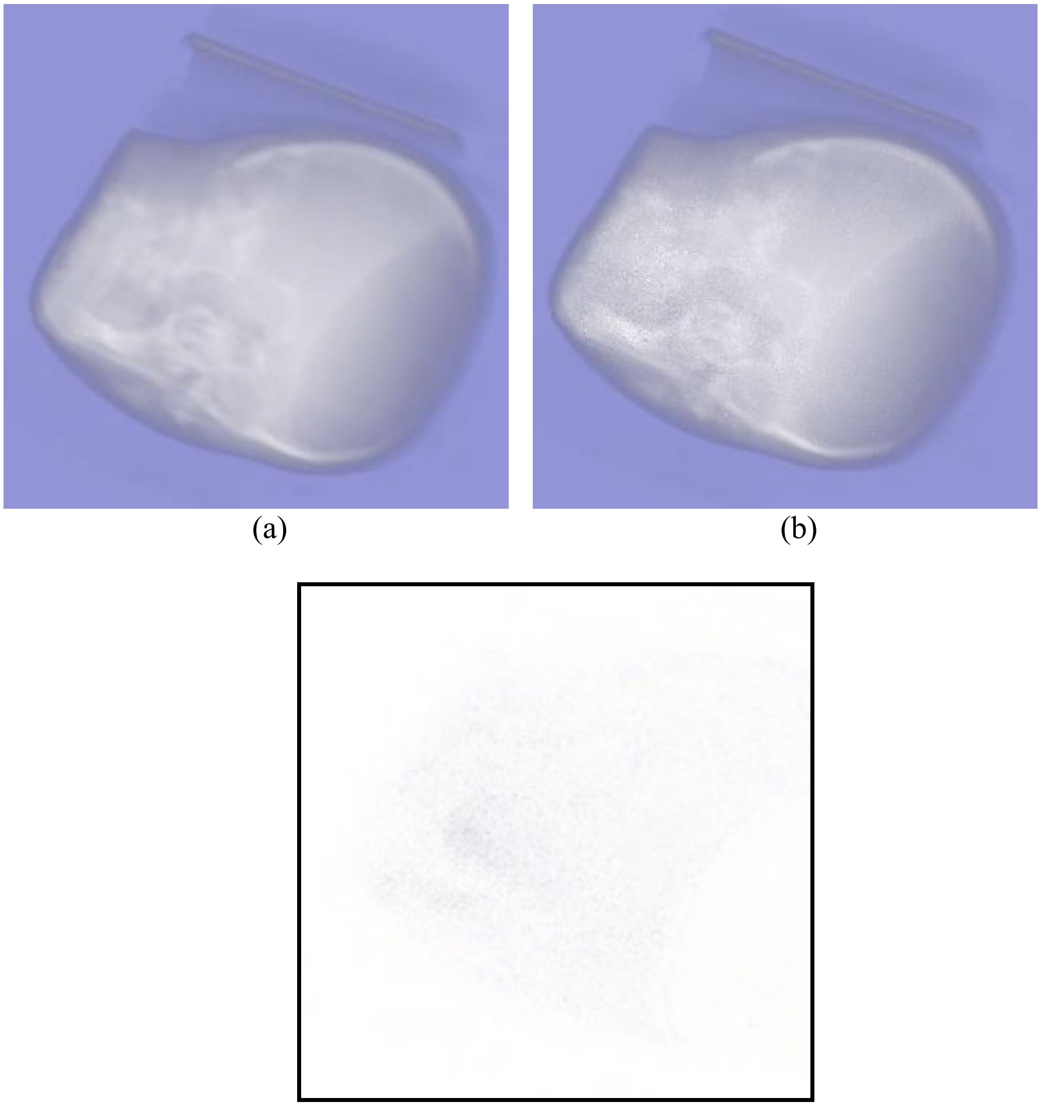
4.2.4 Integrated Value Comparison (Scale Changes)

For a *relative error, which* was greater than the threshold, we decreased the step size linearly with a factor of 0.5. In order to avoid the linear decrement to go below 0, we limited the minimum step size value to 0.0. We gained a significant amount of frames per second for the same quality of the images as that of the ones produced by the original integration technique. These images and the performance gain are depicted in the following diagrams.

Dataset	Size	Dimensions	Threshold (ε)	FPS	SSIM Index
Skewed Head	120 MB	184x256x170	0.03	40	0.0201
Lobster	27 MB	120x120x34	0.03	511	0.0314
Present	69 MB	246x246x221	0.03	33	0.0035

Table 4.4 Performance Table for Integrated Value Comparison (Scale Changes)

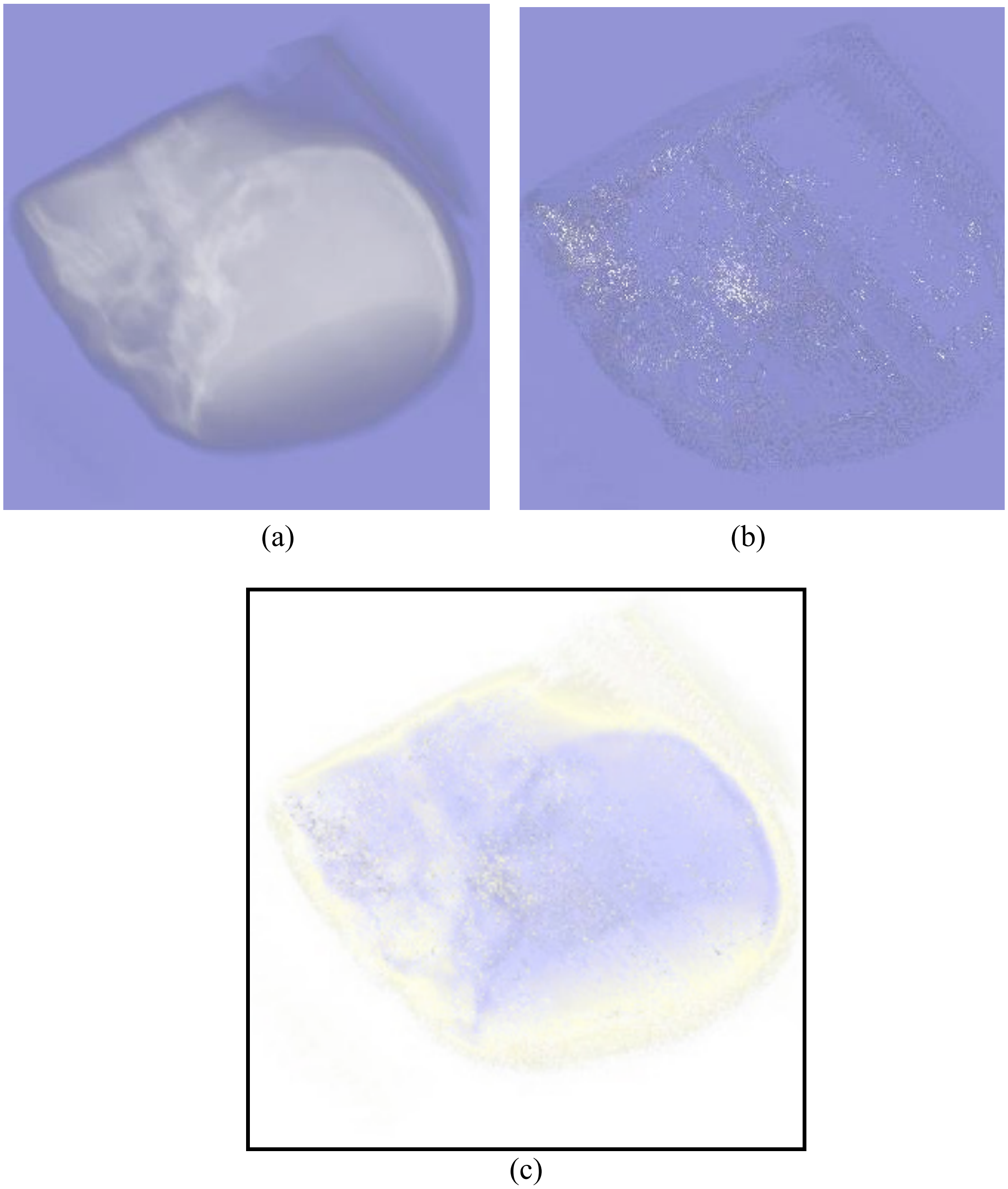
In order to evaluate the quality of images acquired using this integration method, we used the skewed head dataset as shown in Figure 4.14. Figure 4.14 shows the output when the stepsize was increased and decreased with a scaling factor of 1.2 and 0.1 respectively. With the above-mentioned values, the SSIM index value achieved was 0.0201. For a very high boost in the performance gain we got a deteriorated quality output with SSIM index of 0.1386 when the stepsize was changed by a factor of 0.1 as depicted by the Figure 4.15. This performance gain vs. stepsize changing factor comparison is shown in the following graph of Figure 4.16.



(a) (b)

(c)

Figure 4.14 Integrated Value Comparison (Scale Changes): (a) Original Image (b) Shader Implementation (c) Difference Image [Dataset: Skewed Head]


(a) (b)

(c)

Figure 4.15 Integrated Value Comparison (Scale Changes): (a) Original Image (b) Shader Implementation (c) Difference Image [Dataset: Skewed Head]

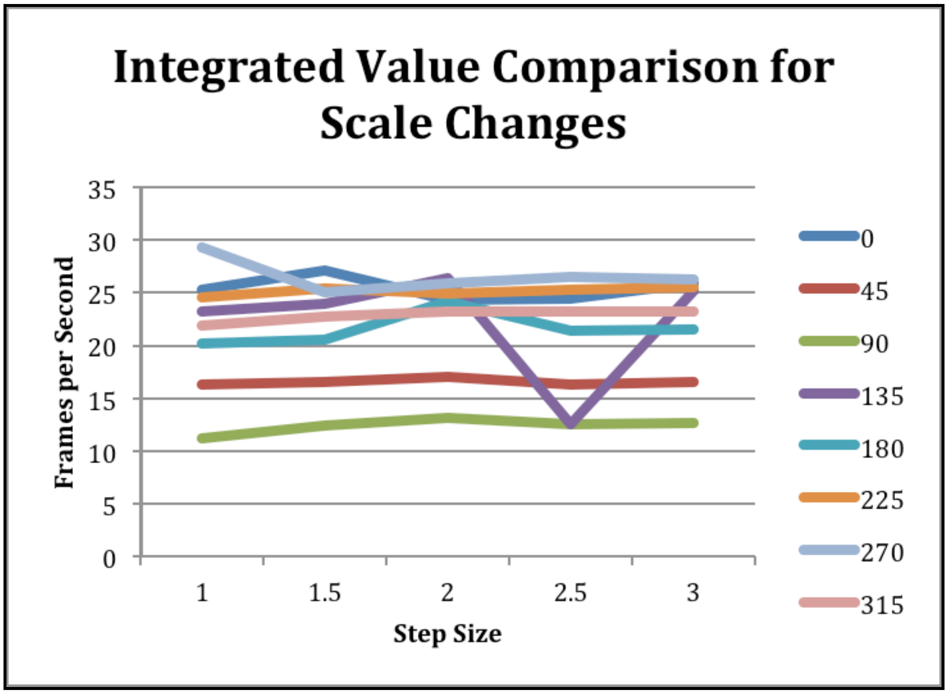


Figure 4.16 Performance Graph for Integrated Value Comparison (Scale Changes)

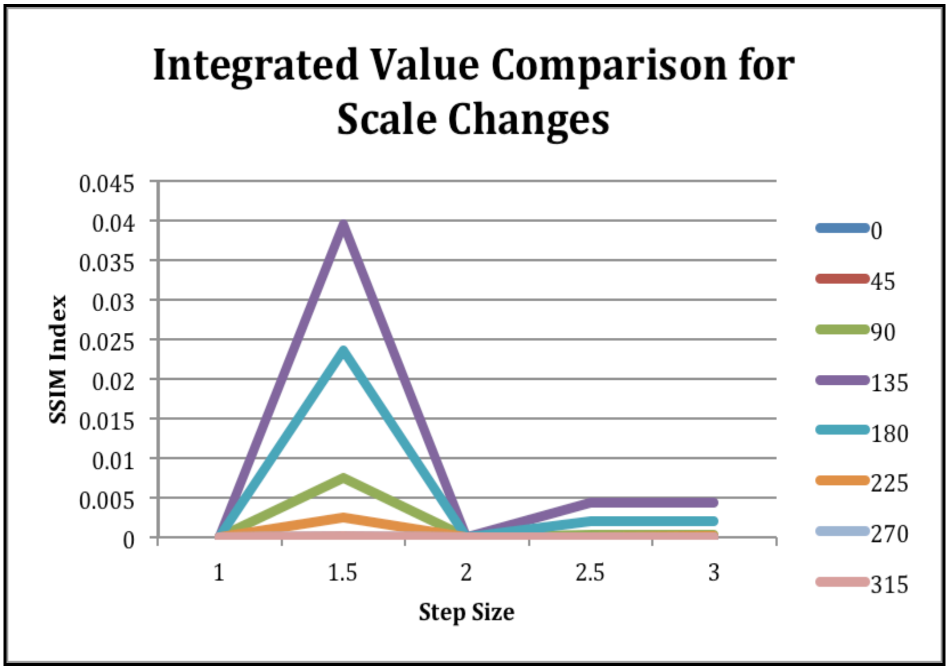


Figure 4.17 Image Quality Graph for Integrated Value Comparison (Scale Changes)

Figure 4.16 shows a graph of the performance in the form of frames per second against the different step size values used for scaling the ray step. Different lines in the graph represent the different angles at which the images were taken. This graph shows that when for all the step sizes we found a significant performance gain, excluding at some viewpoints, compared to the frame rate of original implementation (17 fps). The graph in Figure 4.17 indicates the image quality in the form of SSIM index at different step sizes. As can be seen from this graph when the step sizes were changed in the range of 2 to 3 we found the images, which were close to the original image. This graph also shows that for a scaling factor of 1.5, the image quality degraded when the volume was viewed at 135^o. The combination of these graphs can lead to a conclusion that when the step sizes were changed within a range of 2 to 3 we found a significant gain of upto 30 frames per second without losing image quality in the form of SSIM index.

4.2.5 Effects of Threshold

In this subsection we talk about the effects of changing the threshold with which the difference between color values at intermediate stages in the integration process were compared. As we have discussed in previous sections the threshold value was bound between 0.0 and 0.05 in order to check smaller changes in the volume data while it was bound between 0.05 and 0.1 to check greater changes. Thus to compare the results based on threshold, we changed the bounding values of threshold from 0.05 to 0.15 and 0.1 to 0.2. The image quality was highly deteriorated with SSIM indices of 0.0935 and 0.0314 for both the datasets of lobster and skewed head while the frames per second were boosted. Thus irrespective of the size of dataset a change in the threshold value affected the image quality negatively. Figures 4.18 and 4.19 show the images we got when we had set the threshold value to 0.15 and 0.2 respectively.

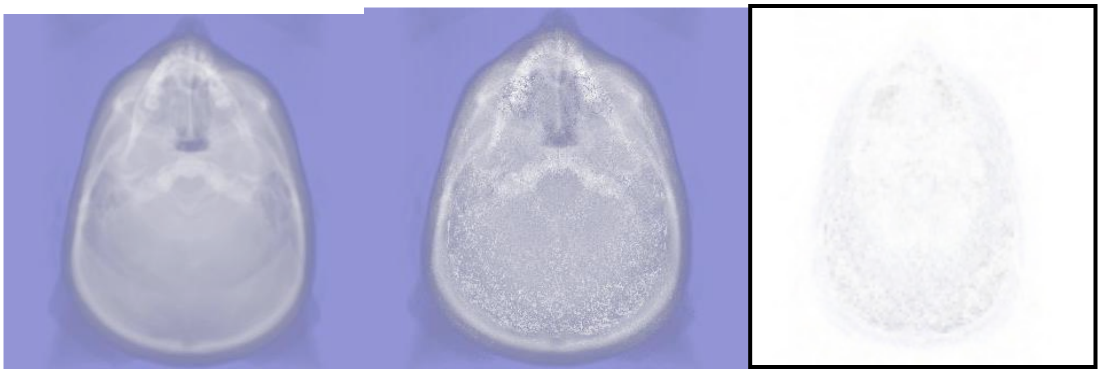

(a) (b) (c)

Figure 4.18 Threshold Comparison (ε = 0.15): (a) Original Image (b) Shader Implementation (c) Difference Image

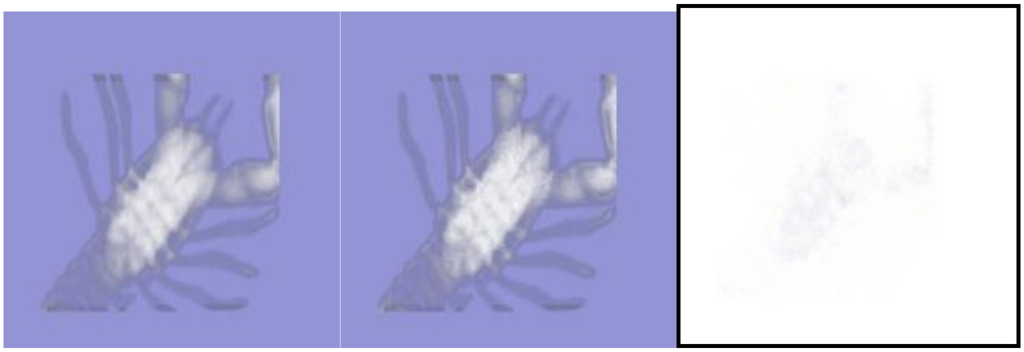
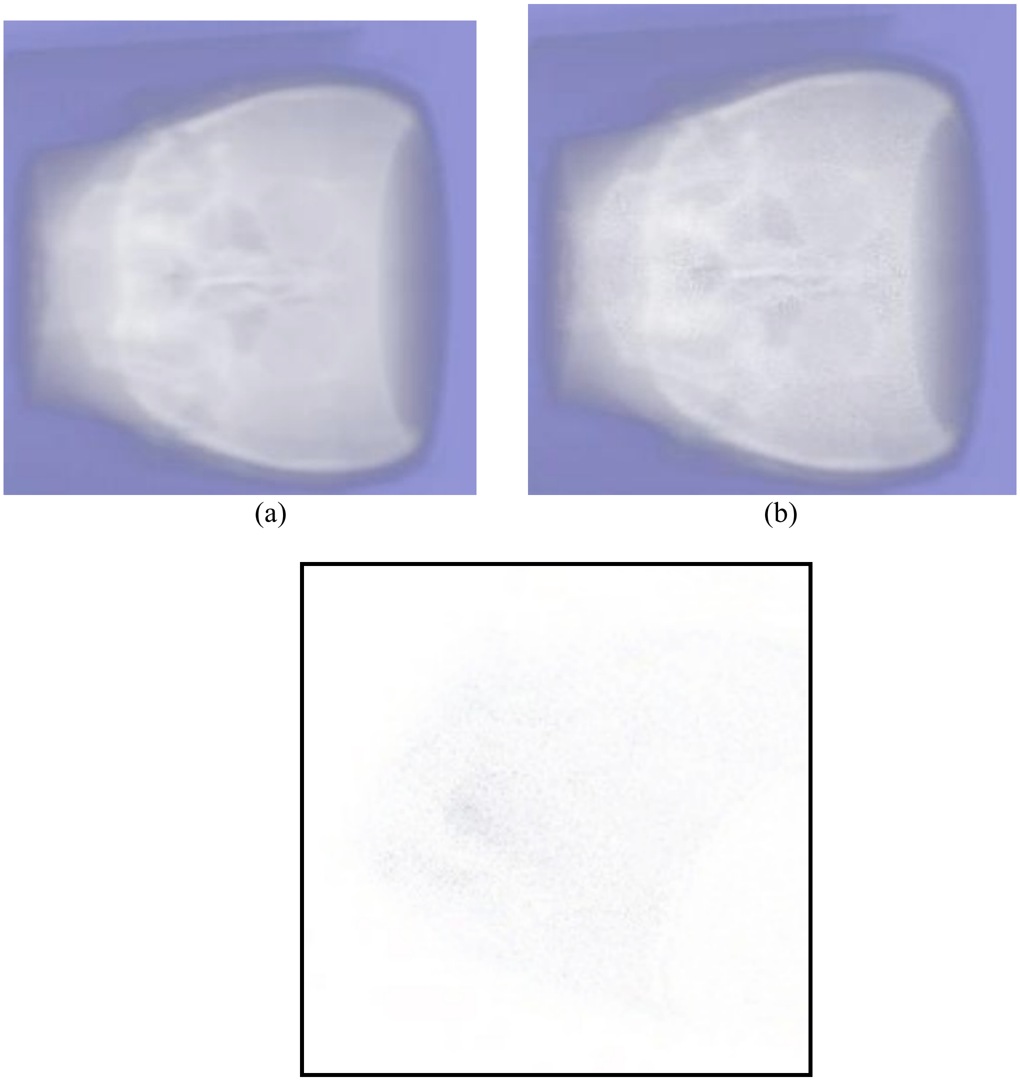

(a) (b) (c)

Figure 4.19 Threshold Comparison (ε = 0.2): (a) Original Image (b) Shader Implementation (c) Difference Image

4.2.6 Poisson Distribution Algorithm

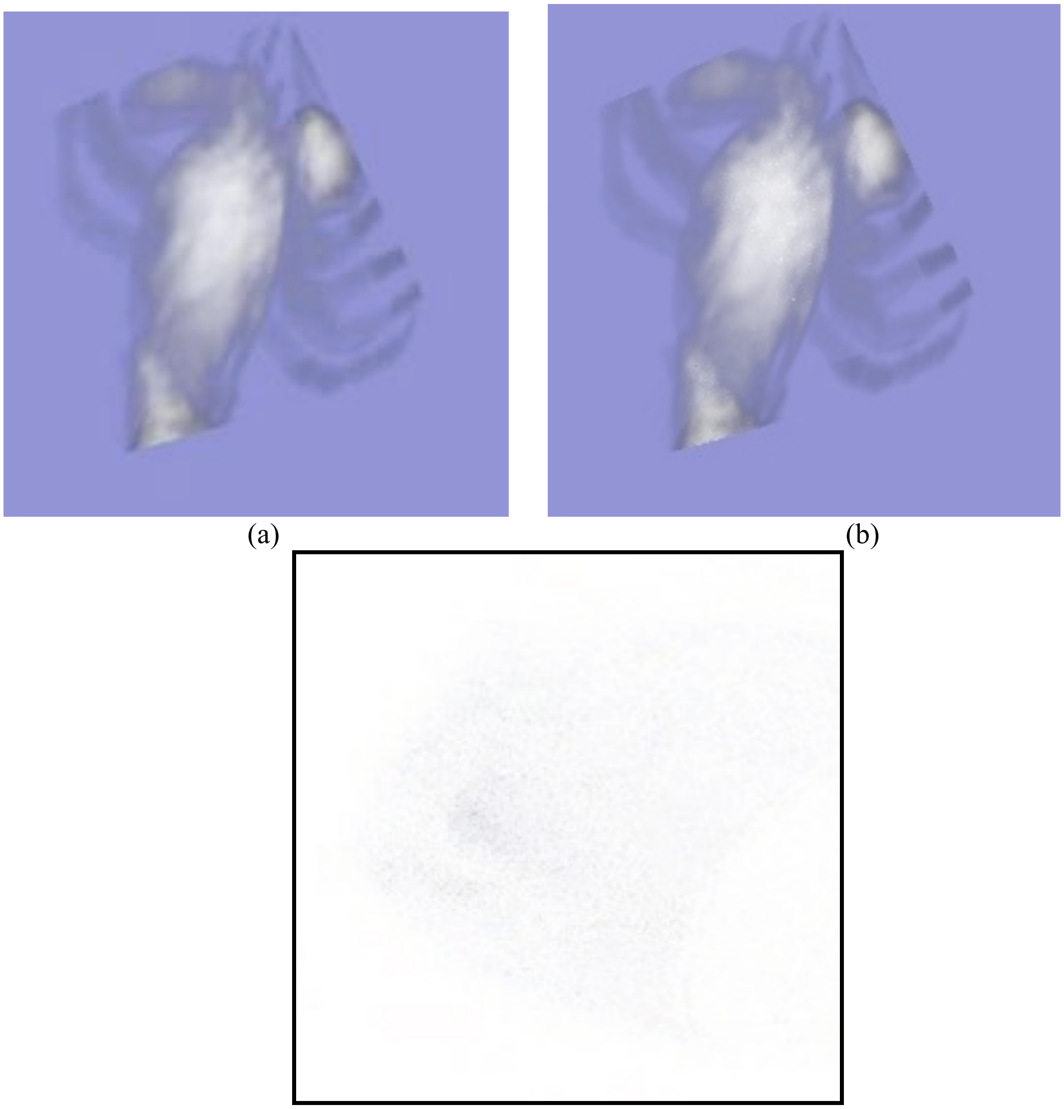
In this subsection we perform the image quality analysis of Poisson distribution algorithm implementation. As discussed in the implementation details chapter, this implementation involved changing the number of samples in a given raystep based upon the change in color and opacity values of consecutive image pixels. Figures 4.20 and 4.21 below show the images acquired using Poisson distribution algorithm with two different datasets. We achieved a framerate that was nearly equivalent to the earlier framerate of 50 frames per second for the skewed head dataset and 300 frames per second for the lobster dataset. Image quality with respect to SSIM index was found to be 0.0512 and 0.0300. Thus we can say that for the Poisson distribution implementation we did not achieve a significant performance gain while the output images were deteriorated as compared to the original images.



(a) (b)

(c)

Figure 4.20 Poisson Distribution Algorithm: (a) Original Image (b) Shader Implementation (c) Difference Image



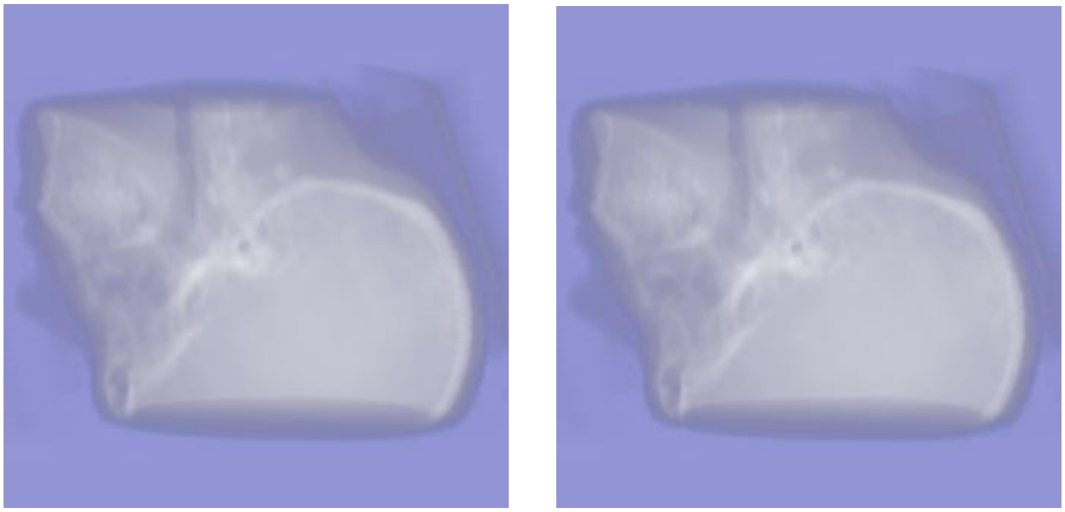
(a) (b)

(c)

Figure 4.21 Poisson Distribution Algorithm: (a) Original Image (b) Shader Implementation (c) Difference Image

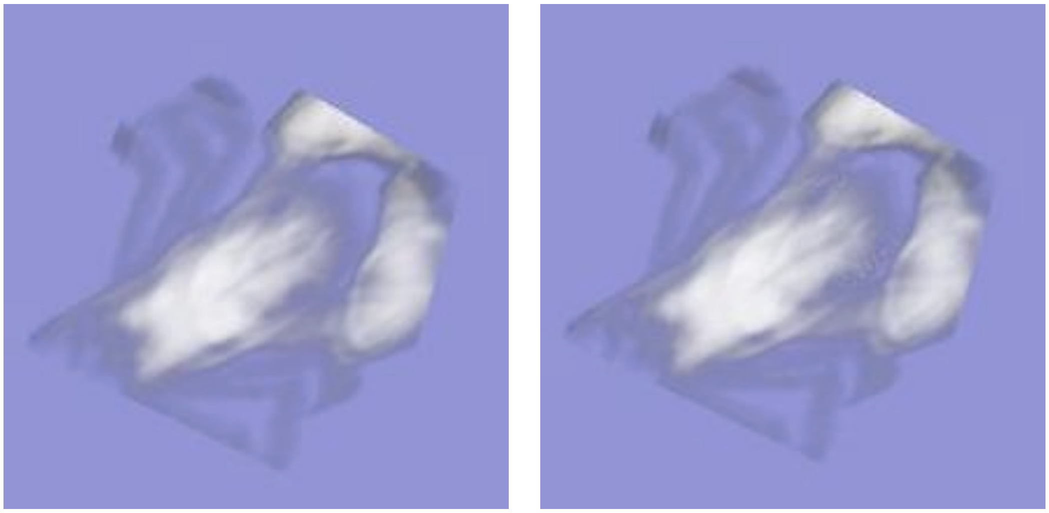
4.2.7 Experiments in LAB Color Space

This subsection talks about the performance and image quality comparisons, when the volumes were generated using LAB color space. Figure 4.22 and Figure 4.23 show the comparison of images acquired in RGB and LAB color space when the step size was linearly changed with a factor of 1.5 and with a scaling factor of 1.2 and 0.5 respectively. We found that there was a significant difference between the image quality in LAB color space and in RGB color space with LAB color space resulting in better images. But at the same time performance in the form of frames per second was seen to have decreased from 60 fps to 48 fps in case of linear changes and from 50 fps to 36fps in case of scale changes. An improvement in image quality with an SSIM index of 0.0001 was a result of better comparison between color values at different steps, while the lower performance can be thought of as result of higher computations (in the form of cube root computations) at each raystep in the volume data.



(a) (b)

Figure 4.22 RGB vs. LAB Comparison (Linear Changes): (a) RGB Image (b) LAB Image



(a) (b)

Figure 4.23 RGB vs. LAB Comparison (Scale Changes): (a) RGB Image (b) LAB Image

Chapter 5: Conclusion and Future Work

In this thesis we have presented a discussion of GPU based ray casting techniques and their advantages. We have incorporated a GPU based ray casting technique for our work, which involved using texture based volume rendering. Our work can mainly be divided into two stages. The first stage involved developing a GPU based volume renderer for medical images. Our system accepted DICOM images acquired from imaging scanners as inputs and converted them to 3D textures using OpenGL implementation. Once the volume data was stored in these textures we developed vertex and fragment shader programs to implement the ray-casting algorithm. Our vertex shader computed the viewpoint for the volume and passed texture information to the fragment shader. Our fragment shader then computed the size of the raystep and performed the ray-casting algorithm for the predefined number of raysteps. This gave us a volume renderer independent of any library or toolkit for reading the DICOM images.

The second stage of our implementation involved trying out different numerical integration techniques for compositing the chromaticity and opacity values along the direction of the ray. The implementations involved computing these values for one level and comparing them with the values of the current level. The other type of implementation involved computing the integrated values of color and opacity at different ray samples. For more interesting changes in the volume data we decreased the size of the ray while for less interesting changes in the volume data the ray size was increased. In order to change the size of the ray step we incorporated linear as well as scaling increments and decrements. One other stream of our implementation involved finding the number of samples in a given raystep for more accurate volumes. This was done using the Poisson distribution algorithm technique where the difference between the color values of pixels in the consecutive raysteps was used as an input to Poisson distribution. Our system also incorporated some user interactivity in the form of allowing the user to change the type of the shaders during runtime, zooming in and out of the volume, a pause and resume facility for the application while it was rotating, a print screen function which could save the current image in a PPM file format, etc. Our system could achieve an interactive frame rate from 10 FPS to almost above 150 FPS depending upon the size and complexity of the volume data being considered. Based on the performance we achieved for different integration techniques and the image qualities; we can say that linear changes in the size of a raystep can produce better results with a significant increase in frames per second as compared to scale changes. With scale changes resulting in deterioration of image quality with a very high performance gain, we found out that Poisson distribution implementation resulted in no performance gain yet a deteriorated image quality. The techniques were tested on a range of very small and trivial datasets to very large and complex datasets. Based on these results we can also say that the level values as well as integrated value comparison given to linear changes in the stepsizes can be applied to all kinds of dataset to achieve a significant performance gain. The implementation algorithm was also tested for LAB color space and based on the comparison results with RGB color space we can conclude that the implementation in LAB color space is useful where visual comparison between images is required.

We expect our system to work for more complex datasets containing computationally expensive texture information. The integration techniques that we have currently implemented compare the chromaticity and opacity values of the volume data at different ray samples. The same technique can be applied to compare the gradient, intensity and other such factors that might affect the computational complexity of an algorithm. Some sort of user interactivity can still be implemented into the system in the form of run-time change in the datasets, cropping of volumes, some lighting sources, etc.

Bibliography

[CCF94] CABRAL B., CAM N., FORAN J.: Accelerated Volume Rendering and Tomographic Reconstruction using Texture Mapping Hardware. In VVS ’94: Proceedings of the 1994 Symposium on Volume Visualization 1994

[CN94] Timothy J. Cullip, Ulrich Neumann Accelerating Volume Reconstruction With 3D Texture Hardware Technical Report for University of North Carolina, Chapel Hill 1994

[DCH88] Robert A. Drebin, Loren Carpenter, Pat Hanrahan Volume rendering SIGGRAPH '88 Proceedings 1988

[DH92] John Danskin, Pat Hanrahan Fast algorithms for volume ray tracing VVS '92 Proceedings of the 1992 workshop on Volume visualization

[DR07] Philip J Davis, Philip Rabinowitz Methods of Numerical Integration Dover Publications, 2007

[EHK* 06] Klaus Engel, Markus Hadwiger, Joe M. Kniss, Christof Rezk-Salama, Daniel Weiskopf Real Time Volume Graphics A. K. Peters, 2006

[EKE01] Klaus Engel, Martin Kraus, Thomas Ertl High-quality pre-integrated volume rendering using hardware-accelerated pixel shading HWWS '01 Proceedings of the ACM SIGGRAPH

[GL80] G.P. Lepage, VEGAS: An Adaptive Multi-dimensional Integration Program, Cornell preprint CLNS 80-447, March 1980

[GPS05] R.N.J. Graham, R.W. Perriss, A.F. Scarsbrook DICOM demystified: A review of digital file formats and their use in radiological practice Clinical Radiology 2005

[HL04] Mark Harris, David Luebke GPGPU Course SIGGRAPH 2004

[HN08] Hubert Nguyen GPU Gems 3 Addison-Wesley, 2008

[JB10] Jon Jansen , Louis Bavoil, Fourier opacity mapping, Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games, 2010

[JV09] Yun Jang, Ugo Varetto Interactive Volume Rendering of Functional Representations in Quantum Chemistry, IEEE Transactions on Visualization and Computer Graphics, 2009

[K69] Donald E. Knuth Seminumerical Algorithms. The Art of Computer Programming, Volume 2. Addison Wesley, 1969

[K97] D. E. Knuth. The Art of Computer Programming, Volume 2: Seminumerical Algorithms, Third Edition. Addison-Wesley, 1997

[KE05] Kristian Edie Use of GPU Functionality in Volume Rendering, Masters Thesis Norwegian University of Science and Technology, 2005

[KK99] Kreeger, K.A. Kaufman, A.E. Mixing translucent polygons with volumes IEEE Visualization ’99 proceedings 1999

[KKH02] Joe Kniss, Gordon Kindlmann, Charles Hansen Multidimensional Transfer Functions for Interactive Volume Rendering IEEE TCVG 2002

[KM04] Kenneth Moreland. Fast High Accuracy Volume Rendering. PhD thesis, University of New Mexico, 2004.

[KOR08] John Kloetzli, Marc Olano, and Penny Rheingans. Interactive volume isosurface rendering using BT volumes. In SI3D ’08: Proceedings of the 2008 symposium on Interactive 3D graphics and games, 2008

[KW03] J. Kruger, R. Westermann Acceleration Techniques for GPU-based Volume Rendering VIS '03 Proceedings of the 14th IEEE Visualization 2003

[KH84] James T. Kajiya , Brian P Von Herzen, Ray tracing volume densities, Proceedings of the 11th annual conference on Computer graphics and interactive techniques, 1984

[LC87] William E. Lorensen , Harvey E. Cline, Marching cubes: A high resolution 3D surface construction algorithm, Proceedings of the 14th annual conference on Computer graphics and interactive techniques, 1987

[LFH08] Hongwei Li, Chi-Wing Fu, Andrew Hanson Visualizing Multiwavelength Astrophysical Data IEEE Transactions on Visualization and Computer Graphics 2008

[LMK03] Wei Li, Klaus Mueller, Arie Kaufman Empty Space Skipping and Occlusion Clipping for Texture-based Volume Rendering VIS '03 Proceedings of the 14th IEEE Visualization 2003

[ML90] Marc Levoy Efficient Ray Tracing of Volume Data ACM Transactions on Graphics (TOG) 1990

[MW10] Magnus Wrenninge Volume Rendering Course SIGGRAPH 2010

[NA92] Kevin Novins, James Arvo Controlled precision volume integration VVS '92 Proceedings of the 1992 workshop on Volume visualization

[PS02] Paul Suetens Fundamentals of Medical Imaging Cambridge University Press, 2002

[PF90] W.H. Press, G.R. Farrar Recursive Stratified Sampling for Multidimensional Monte Carlo Integration, Computers in Physics 1990

[RGW* 03] Stefan Roettger, Stefan Guthe, Daniel Weiskopf, Thomas Ertl, Wolfgang Strasser Smart Hardware Accelerated Volume Rendering IEEE TCVG Symposium on Visualization 2003

[RPSC99] Harvey Ray, Hanspeter Pfister, Deborah Silver, Todd A. Cook Ray Casting Architectures for Volume Visualization IEEE TCVG 1999

[SSKE05] Simon Stegmaier , Magnus Strengert , Thomas Klein , Thomas Ertl A Simple and Flexible Volume Rendering Framework for Graphics-hardware–based Raycasting Volume Graphics 2005

[TW80] Turner Whitted An improved illumination model for shaded display Magazine Communications of the ACM 1980

[WA91] Westover, Lee Alan (July, 1991). "SPLATTING: A Parallel, Feed-Dorward Volume Rendering Algorithm"

[WE98] Rüdiger Westermann, Thomas Ertl Efficiently using graphics hardware in volume rendering applications SIGGRAPH '98 Proceedings

[WH03] Warren J. Hehre A Guide to Molecular Mechanics and Quantum Chemical Calculations, Wavefunction Inc. 2003

[WBSS04] Zhou Wang, Bovik, A.C., Sheikh, H.R., Simoncelli, E.P. Image quality assessment: from error visibility to structural similarity IEEE Image Processing 2004