INTERACTIVE VOLUME RENDERING FOR MEDICAL IMAGES

A THESIS SUBMITTED TO

THE GRADUATE SCHOOL OF INFORMATICS

OF

THE MIDDLE EAST TECHNICAL UNIVERSITY

BY

KORAY ORHUN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

THE DEPARTMENT OF INFORMATION SYSTEMS

SEPTEMBER 2004

Approval of the Graduate School of Informatics

Prof. Dr. Neşe YALABIK
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Onur DEMİRÖRS
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Assist. Prof. Dr. Erkan MUMCUOĞLU
Supervisor

Examining Committee Members

Assoc. Prof. Dr. Onur DEMİRÖRS ______________________

Assist. Prof. Dr. Uğur GÜDÜKBAY ______________________

Assoc. Prof Dr. Veysi İŞLER ______________________

Assist. Prof. Dr. Erkan MUMCUOĞLU ______________________

Assoc. Prof. Dr. Yasemin YARDIMCI ______________________

ABSTRACT

INTERACTIVE VOLUME RENDERING FOR MEDICAL IMAGES

Orhun, Koray

MS., Department of Information Systems
Supervisor: Assist. Prof. Dr. Erkan MUMCUOĞLU

September 2004, 159 pages

Volume rendering is one of the branches of scientific visualization. Its popularity has grown in the recent years, and due to the increase in the computation speed of the graphics hardware of the desktop systems, became more and more accessible. Visualizing volumetric datasets using volume rendering technique requires a large amount of trilinear interpolation operations that are computationally expensive. This situation used to restrict volume rendering methods to be used only in high-end graphics workstations or with special-purpose hardware. In this thesis, an application tool has been developed using hardware accelerated volume rendering techniques on commercial graphics processing devices. This implementation has been developed with a 3D texture based approach using bump mapping for building an illumination model with OpenGL API. The aim of this work is to propose visualization methods and tools for rendering medical image datasets at interactive rates. The methods and tool are validated and compared with a commercially available software.

Keywords: Direct volume rendering, PC graphics hardware, OpenGL, bump mapping, multi-texturing, medical imaging, Phong Illumination model, 3D texture mapping

ÖZ

TIBBİ GÖRÜNTÜLER İÇİN ETKİLEŞİMLİ HACİM KAPLAMA

Orhun, Koray

Yüksek Lisans, Bilişim Sistemleri Bölümü
Tez Yöneticisi: Yrd. Doç. Dr. Erkan MUMCUOĞLU

Eylül 2004, 159 sayfa

Hacim kaplama, bilimsel görselleştirme yöntemleri içerisinde önemli bir yöntemdir. Kişisel bilgisayarların grafik işlemcilerinde yaşanan son zamanlardaki hız artışı, bu metodun kullanımını yaygınlaştırmaktadır. Hacimsel verilerin görselleştirilmesinde kullanılan hacim kaplama yöntemi, bilgisayar işlemci zamanını çok harcayan bir hesaplama yöntemi olan üçlü doğrusal aradeğerleme işlemini çokça yapmaktadır. Bu durum, hacim kaplama yöntemi kullanarak görselleştirme işlemlerinin, ancak son teknoloji iş istasyonları veya özel amaca yönelik üretilmiş donanımlarla gerçekleştirilebilmesine olanak sağlamaktaydı. Bu tezde, günümüz kişisel bilgisayarları için üretilmekte olan grafik işlemcilerini kullanarak, donanım ile hızlandırılmış hacim kaplama konusunda bir yazılım geliştirilmiştir. Geliştirilen yazılımda, üç boyutlu dokuların kullanılması yöntemini temel alarak, vurdurarak doku eşleme (*bump mapping)* yöntemi ile OpenGL uygulama programı arayüzü üzerinden bir ışıklandırma modeli kullanılmıştır. Bu çalışmanın amacı, etkileşimli hızlarda tıbbi verileri görüntüleyebilecek bir yöntem sunmaktır. Geliştirilen yazılım, yaygın olarak kullanılan bir ticari yazılımla karşılaştırılmıştır.

Anahtar kelimeler: Hacim kaplama, kişisel bilgisayar grafik donanımı, OpenGL, vurdurarak doku eşleme, çoklu doku, tıbbi görüntüleme, Phong ışıklandırma modeli, üç boyutlu doku kaplama

To my grand-mother Câhide and grand-father Fâzıl Engin

ACKNOWLEDGEMENTS

I wish to express my deepest gratitude and special thanks to my advisor, Assist. Prof. Dr. Erkan Mumcuoğlu, for his invaluable support and encouragement. I wish to thank him very much for his endless patience and perfect guidance, not only in supervising of this work but also in his question-ask-able lectures which enhanced my vision.

I wish to extend my sincere thanks to my friend Mehmet Şaşmaz, for his endless ambition to help and his invaluable comments, suggestions and remarks for my works.

I wish to thank to Dr. Bilge Volkan Salancı, for her invaluable supports for the analysis section of this study.

I would like to thank to my friends, Barış Tosun, Fatih Nar, Ümit Yaşar Karadeniz, Ayşegül Özkul, Levent Çapçı, Utku Göker, Balkan Uraz, Diren Çakıcı and my ex-manager Serdar Ak for their invaluable supports.

My appreciations go to the company AnalyzeDirect, which send me the evaluation version of the software Analyze 5.0 for the analysis section of this work.

Finally, I would like to express my endless thanks to my family for their endless support and patience.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS AND ABBREVIATIONS

2D	Two Dimensional
3D	Three Dimensional
API	Application Programming Interface
DICOM	Digital Imaging and Communications in Medicine
CAD/CAM	Computer-aided design/computer-aided manufacturing
CPU	Central Processing Unit
CT	Computed Tomography
JOGL	Java bindings for OpenGL API
GHz	Giga Hertz
GL	Graphics Library
GPU	Graphics Processing Unit
GUI	Graphical User Interface
MB	Mega Byte
MHz	Mega Hertz
MIP	Maximum Intensity Projection
MRI	Magnetic Resonance Imaging
PC	Personal Computer
PET	Positron Emission Tomography
RGB	Red, Green, Blue
RGBA	Red, Green, Blue, Alpha
SGI	Silicon Graphics Inc.
SPECT	Single Photon Emission Computed Tomography

CHAPTER 1

INTRODUCTION

The contribution of this study is primarily in volume rendering. Using hardware acceleration in computer graphics has become very important due to fast evolution of the graphics processing devices in the recent years. The vast amount of production and development of computer graphics devices in the industry enabled the consumers to own high performance graphics computation power without a high expenditure on workstations. The main focus of our study is on the capabilities of hardware accelerated volume rendering techniques.

1.1 Motivation

The main motivation for this study is to propose useful solutions and techniques for the visualization section of the project: "*Three Dimensional Brain Image Processing*" which is directed by *Assist. Prof. Dr. Erkan Mumcuoğlu* in the *Informatics Institute* department of *Middle East Technical University.* The primary focus of this project is to register medical imaging volumetric datasets of patient's brain which were acquired with different modalities (Nuclear Medicine, Radiology, etc.) and visualize them in three dimensional spaces for the medical staff use.

The main advantage of *Nuclear Medical Imaging* studies is to present not only the morphologic information about the organ systems of the patients, but also give information about their functions. The medical images acquired by *Radiology Technology* present a higher level of detail for the anatomical structures of the patient's organ systems. However, they present less information about the functions. The main disadvantage of Nuclear Medical Imaging is that the resolution of the acquired datasets is very low with respect to the ones acquired with Radiology

Technology, and the small structures may not be localized correctly. In some situations, there is a requirement for building a correlation between the results acquired by both of the techniques for better diagnostics, and guide the surgeon before an operation by localizing the functional information with the anatomical structure. This process is called *co-registration,* which requires sophisticated hardware and software systems. Even though the results are registered, visualizing the datasets in a three dimensional environment using computer graphics has a great importance on enabling doctors to understand the structures more clearly.

There are many researches about this phenomenon, and in many developed countries such techniques are being used. In *Hacettepe University Hospital,* very sophisticated devices are used for acquiring medical imaging datasets on different modalities, however this co-registration and visualization with computer graphics processes cannot be established. *Analyze* [37] is one of the most popular software system for establishing this technique; however it requires a huge amount of expenditure.

The aim of the project "*Three Dimensional Brain Image Processing*" is to develop a system that is able to co-register and visualize multimodality medical image datasets according to the requirements, and distribute this system to the hospitals that are interested, with no fee. The motivation for our studies is to propose a solution for the visualization section of this project.

1.2 Organization of the Thesis

This Thesis has 6 Chapters:

Chapter 1: Gives the motivation and some introductory information about visualization. It also mentions different methods of computer visualizations.

Chapter 2: Gives detailed information about volume rendering and presents a pipeline for volume rendering.

Chapter 3: Summarizes different kinds of volume rendering techniques using PC graphics hardware, and gives an introductory information about computer graphics.

Chapter 4: Presents detailed information about methods of the implementation developed for this thesis.

Chapter 5: Gives the testing results of the implementation and presents some qualitative comparison with software *Analyze*.

Chapter 6: Concludes the thesis by discussing the results of the tests, and gives a pathway for the future studies.

1.3 Three Dimensional Visualization

"Forming an image is *mapping* some property of an object onto *image space.* This space is used to visualize the object, and its properties and may be used to characterize quantitatively its structure or function. *Imaging science* may be defined as the study of these mappings and the development of ways to better understand them, to improve them, and to use them productively" [1, pp. 685].

3D visualization refers to the process of transforming and displaying the three dimensional objects in a way that their nature is able to be seen. 2D display devices that visualize shaded graphics of the rendered objects or 3D display devices that enable stereoscopic or holographic type of displays are used for visualizing the outputs of 3D visualization methods. The term *visualization* not only concerns methods for displaying but also includes methods for manipulating and analyzing the displayed information, "this term implies inclusion of cognitive and interpretive elements" [1, pp. 686]. The term *3D imaging* is generally defined as acquiring digital samples of objects in a three dimensional environment. This term also includes processing, analyzing and visualizing of the sampled datasets.

1.3.1 Methods of Three Dimensional Visualization

There are various kinds of techniques used for visualizing 3D datasets. The generally used methods especially in biomedical research and clinical applications shall be defined briefly in this section.

2D Display:

2D image generation and display methods aim to generate optimal 2D images from 3D volumetric datasets by allowing the user to set the orientation of the 2D image plane for visualizing unrestricted view of important features in the sampled 3D information.

Multiplanar Reformatting is the process of constructing 2D images from the volumetric dataset by reslicing the set of data in any arbitrary spatial direction which visualize the images that lie along the non-acquired orthogonal orientations of the volume. Figure 1.1 [38] shows an example for this method of display.

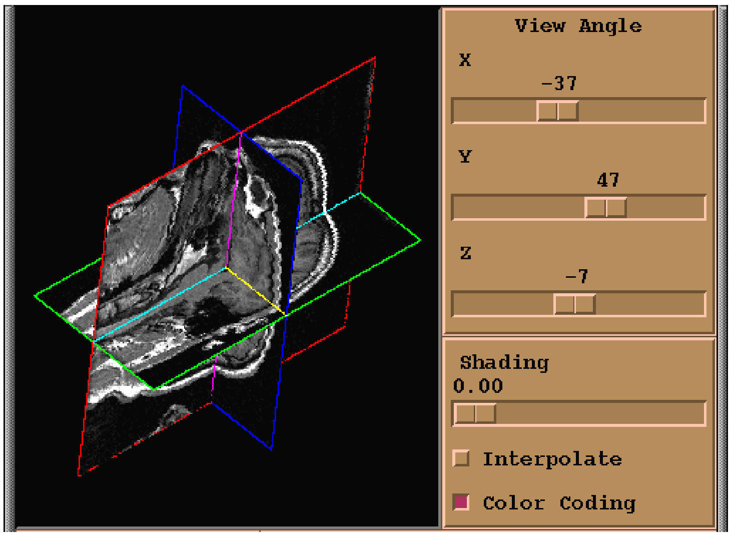


Figure 1.1 Multiplanar slicing

Oblique Sectioning [39] is the method of visualizing a desired plane in the 3D volumetric dataset that is not parallel to any orthogonal orientation in which the volume has been acquired. An example display for oblique sectioning is given in Figure 1.2 [39].

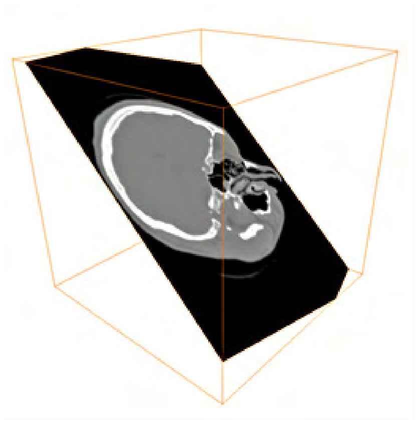


Figure 1.2 Oblique Sectioning

Curved Sectioning is the method of displaying structures that have curvilinear morphology which oblique and multiplanar methods cannot display. An example for curved sectioning is given in Figure 1.3 [1].

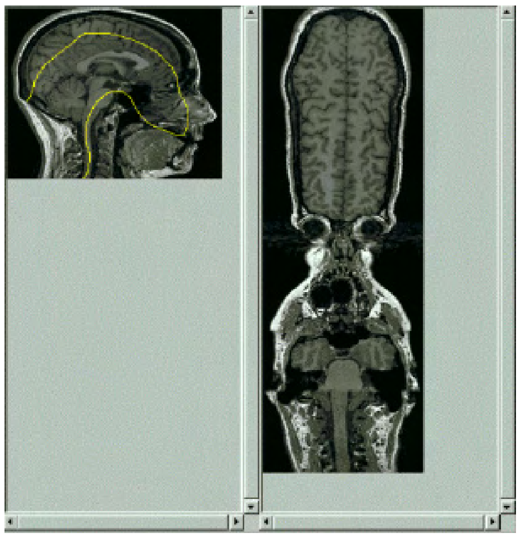


Figure 1.3 Curved Sectioning

3D Display:

"Visualization of 3D biomedical volume images has traditionally been divided into two different techniques: *Surface Rendering and Volume Rendering"* [1, pp. 688].

Surface Rendering is the visualizing techniques in which the contours of the edges present in the volumetric dataset are extracted as geometric primitives, and visualized by a mosaic of connected polygons representing the surfaces. There are some different approaches for extracting the surfaces from the volume. *Surface reconstruction from contours* [2] and *Marching Cubes* [3] algorithms are some of the popular methods used for extracting surfaces from volumes. The main advantage of this technique is that, the results of surface extraction methods are geometric primitives that are polygons. Polygon based surface representation enables the information to be transformed into analytical descriptions; so, the standard computer graphics techniques like transforming the volume or using illuminations models, are able to be applied for this method of visualizing, and other visualization packages of CAD/CAM software can be used for displaying. "The disadvantages of this technique are largely based on the need to discretely extract the contours defining the structure to be visualized. Other volume image information is lost in this process, which may be important for slice generation or value measurement. Finally, because of the discrete nature of the surface polygon placement, this technique is prone to sampling and aliasing artifacts on the rendering surface."[1, pp. 689]

Volume Rendering is one of the most powerful visualization techniques being used for displaying volumetric datasets [1]. The focus for our study is mainly on volume rendering, so it shall be defined in detail in the following chapters.

In some of the resources, the term *Surface Rendering* described here is called as *Indirect Volume Rendering,* and *Volume Rendering* is called as *Direct Volume Rendering* [4]. In this document the term *Volume Rendering* refers to *Direct Volume Rendering.*

CHAPTER 2

VOLUME RENDERING

“To visualize noninvasively human integral organs in their true form and shape has intrigued mankind for centuries. If the discovery of X-rays gave birth to radiology, the invention of computerized tomography and magnetic resonance imaging has revolutionized radiology. Three dimensional imaging is another recent development that has brought us closer to fulfilling the age-old quest of noninvasive visualization.” [5]

The importance of scientific computing has become more significant in the recent evolution period of the technology. Scientific computation applications require new techniques to process the vast amount of data and be interpreted by the scientist. This requirement has resulted in a new field in the computer graphics studies, which is called *Scientific Visualization.* The recent increases in the performance of the computing have made it possible to use the scientific visualization in many different disciplines with complex datasets that are rich in quality. Medical imaging, computational fluid dynamics simulations, meteorology, molecular modeling and geographical information systems are some of the disciplines that use the advantages of scientific visualization.

Volume rendering, which is one of the branches of scientific visualization, popularity has grown considerably in the recent years, and due to increase in the computation speed in the desktop systems, became more and more accessible.

2.1 What is Volume Rendering?

Volume rendering is a method of visualizing a three dimensional (3D) volumetric data as two dimensional (2D) image. An example for volumetric data is the sampling of an object in three dimensions. In medical imaging, Positron Emission Tomography (PET), Magnetic Resonance Imaging (MRI) datasets are examples for volumetric datasets. Volume Rendering techniques enable these three dimensional datasets to be transformed into a meaningful image, and this process is called *rendering.* In traditional computer graphics, rendering is the action of painting a picture of a scene as if the user is looking from a specific point to a specific direction in the scene. It uses geometric calculations for how shall the primitives (points, lines, polygons) will be seen in the camera (two dimensional result of the rendering process) and textures added to the objects in the scene with the addition of lighting into the rendering calculations; the realism of the rendered two dimensional image is increased. Volume rendering techniques process the three dimensional datasets and transform into a rendered result image by using lighting functions from the study of computer graphics, classify the data with image processing techniques and apply compositing by emulating alpha blending from computer graphics studies.

2.2 Where is Volume Rendering Used?

Many different disciplines and sciences use volume rendering technique. In medical imaging, the human internals captured with Magnetic Resonance Imaging (MRI), Computed Tomography (CT), Ultrasound, PET and Single Photon Emission Computed Tomography (SPECT) scanners produce vast amount of datasets which are to be analyzed by doctors or physicians. The sampled datasets are needed to be viewed in different directions, rotated, zoomed and also separately colored in order to distinguish one type of tissue from another. Volume rendering techniques help the surgical planning processes; haptics[40] and telepresence surgery technology, in which the doctor can conduct a surgery on a patient in a remote location. Volume rendering methods help the paleontologists to distinguish between a fossil and the ground that covers it, by the help of a CT scanner. Computational Fluid Dynamics science (which is used in many areas such as designing exhaust manifolds for engineers, designing wings of aero planes) is governed by a set of derived equations that consist of velocity and vorticity (a measure of the rotation of air in a horizontal plane) of a fluid’s flow. Scientists use volume rendering techniques, in order to monitor all of these values through a structure. Meteorologists and other scientists who use modeling techniques use volume rendering in viewing and analyzing their models built for inquiring the phenomena such as ocean turbulence, precipitation, solar magnetic storms, ozone layer, typhoons, acid rains and hurricanes. Volume rendering enables the viewer to examine inside of something, without removing physically the layers. Visible Human Project [41] is a helpful tool for nondestructive testing processes. CT scan techniques combined with volume rendering visualization technique, non-destructive testing can be obtained. Volume rendering is also an essential tool for microbiologists for microscopic analysis and geoscientists for oil explorations.

The uncertainty principle, which was thought by the German Physicist Werner Heisenberg, tells that: “It is impossible to measure the trajectory of an electron moving through space. The very act of observing the electron shall alter its path and contaminate the experiment.”[42] This principle is a significant problem for failure analysis in different fields. For example, in order to find a failing reason of an engine or to find out if a building structure has been damaged of not, the analyst have to give harm or sometimes even destroy the inquired structure. But in medical imaging, the harm to the patient is kept at minimum by keeping the radiation levels low.

2.3 Terminology and Overview

In a digital image, the information is stored in a two dimensional array which represents color of light intensity or transparency. Data elements kept in the array are called *pixels.* Volumetric dataset can be defined as a three dimensional digital image. The information of volumetric dataset is stored in a three dimensional array in which data elements are called *voxels.* A pixel value stores the information of a point in a two dimensional spatial coordinates, and a voxel stores the information of a point in a three dimensional spatial coordinates. In literature, voxel is defined in two different ways: in the first definition voxel is considered as a small cube; in the other definition, voxel is considered as a point which has no size but has a location in the three dimensional space. In this study, the second definition shall be used.

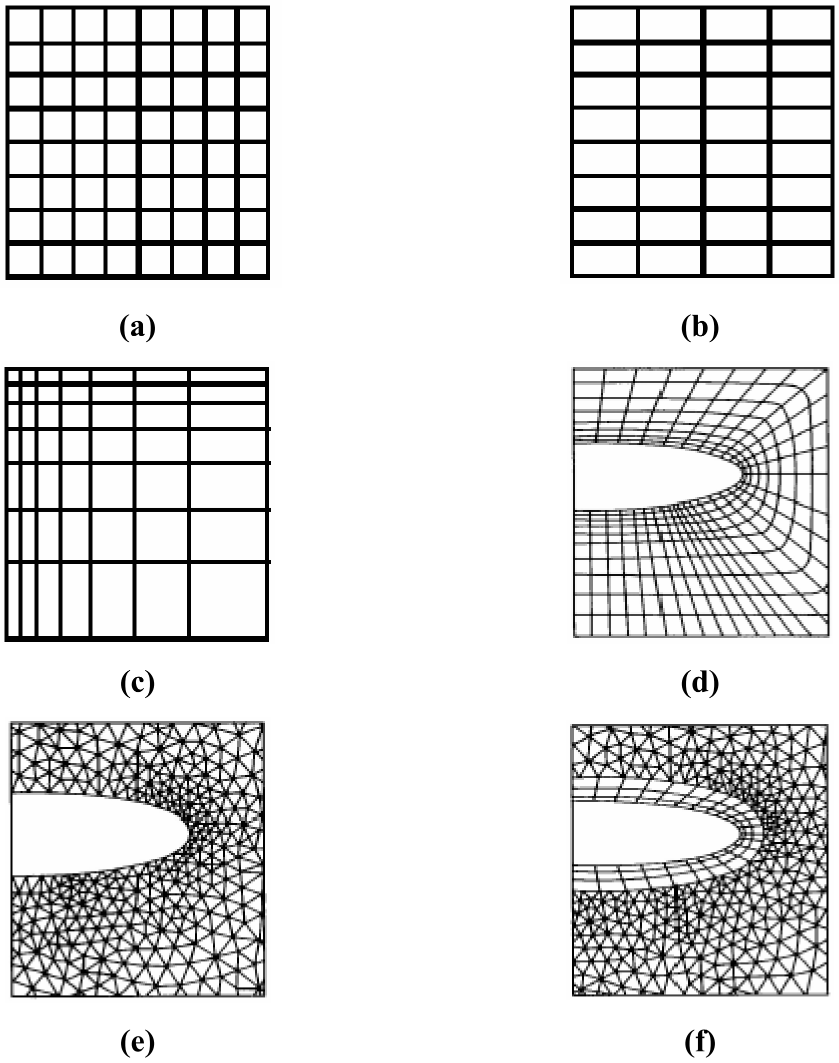
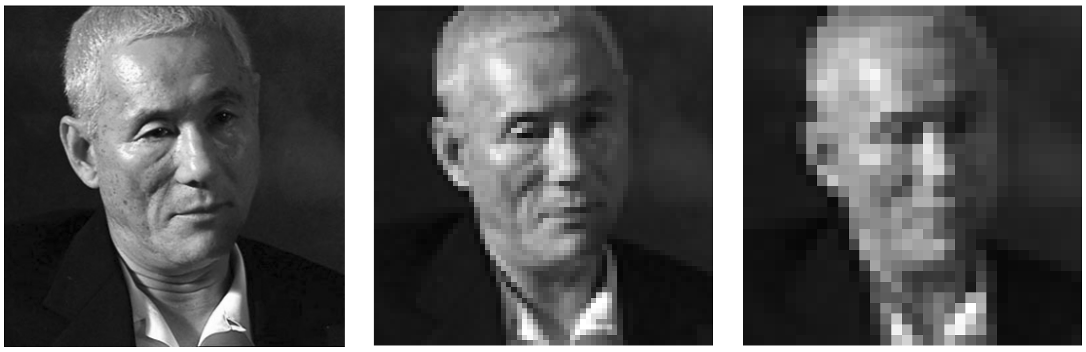


Figure 2.1 Different Types of Grids

(a) *Cartesian Grid:* Typically known as a voxel grid. Data elements are cubic and axis aligned; (b) *Regular Grid:* Similar to Cartesian grid, but cells are rectangular; (c) *Rectilinear Grid:* Similar to regular grid, but the cell dimensions vary; (d) *Structured (Curvilinear) Grid:* Hexahedra or rectangular cells warped to fill a volume, or warped around an object; (e) *Unstructured Grid:* No geometric constraints are imposed. The cells may be tetrahedral, hexahedra, prisms, pyramids, etc; (f) *Hybrid Grid:* A combination of structured and unstructured grids.

Measuring a property of a physical environment at a specific location is called sampling. Sampled information may be color value, light intensity, transparency, hue, density, temperature, acceleration, etc. Voxels are the sampled information which imposes a grid on the volume. Different kinds of grids may be classified as shown in the Figure 2.1 [43], [4].

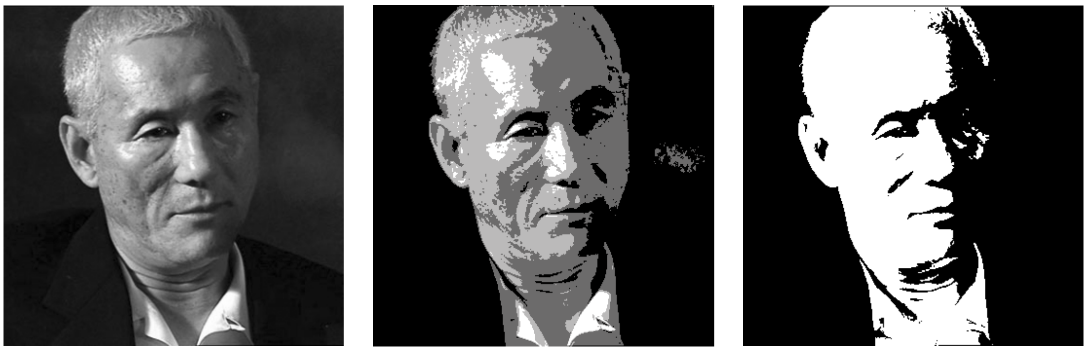
In this thesis, Cartesian type of sampled volumetric data shall be used as input. The density or amount of spacing between sampled points differs by the spatial resolution of the dataset.



(a) (b) (c)

Figure 2.2 Example for different spatial resolutions
(a) 350x350 pixels; (b) 64x64 pixels; (c) 32x32 pixels.

Quantizing is the process of storing the sampled information in the digital environment. Intensity resolution is the number of bits which is used for the storage of the sampled information. Using higher number of bits for each sampled point increases the intensity resolution. Examples for different spatial and intensity resolutions is given in Figure 2.2 and 2.3 [6, pp. 17] respectively.



(a) (b) (c)

Figure 2.3 Example for different intensity resolutions
(a) 8 bits/pixel; (b) 2 bits/pixel; (c) 1 bit/pixel.

2.4 Volume Rendering Pipeline

The aim of visualization is to enable the user understand what is happening or stored in the dataset. Volume rendering, which is a method in three dimensional computer graphics, gives user the ability of making any kind of sense of a group of voxels and view their relationships.

Volume rendering pipeline is a kind of dataflow diagram which shows the main operations required for the overall volume rendering process. Different volume rendering implementations may exclude or change the order of some operations shown in the volume rendering pipeline diagram [6, pp. 29].

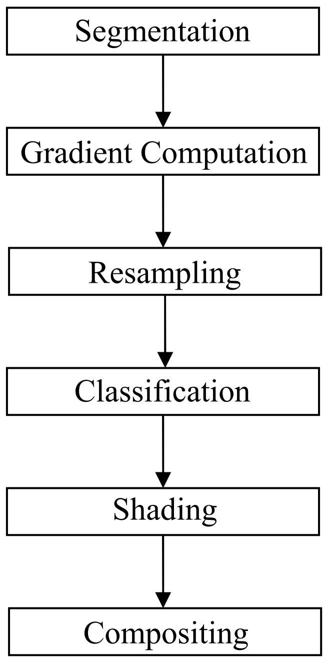


Figure 2.4 Volume Rendering Pipeline

2.4.1 Segmentation

Each volumetric dataset has certain characteristics according to its data acquisition technique, and in most of the data acquisition techniques, the sampled voxel values carry information that cannot be visualized directly; such as "density, acoustic impedance, tissue magnetization and the like" [6, pp. 29].

In order to visualize this non-visual information, we need to assign color or light intensity / transparency to each voxel in the dataset.

Segmentation is the process of labeling the voxels inside a volumetric dataset. Segmentation operation is done before the rendering phase and it categorizes / separates the whole data into structure that are formed by certain relationships between voxels. For example: a volumetric dataset acquired by MRI which contains information of a patient's head shall contain skull data in some of the voxels and brain data in some other voxels. Segmentation enables us to label each voxel either brain or skull.

Segmentation is an important pre-rendering process to achieve high quality visualization. However, in some kinds of volumetric datasets, it is a complicated process that requires many different image processing algorithms. It is not always possible to extract every different feature in a dataset automatically. Usually, segmentation algorithms are semi-automatic which require some user interaction for maximum success. "There are many researchers working on the problem of extracting, or segmenting features in a dataset. It is sometimes not possible to come up with an automatic algorithm that does the segmentation for you" [6, pp. 97]. Segmentation is usually a difficult and time consuming task; however it is sometimes essential for visualization. For example; in the patient's MRI head dataset example, it might not be possible to visualize the brain data as it is covered with a skull.

When a volumetric data is applied a segmentation process, interested features are labeled in the voxels so that, each voxel is part of a material or a feature. Thus segmentation process shall be done before rendering and other processes in the volume rendering pipeline. Classification processes and coloring transfer functions may use the output of segmentation process, and more meaningful and higher quality renderings can be obtained.

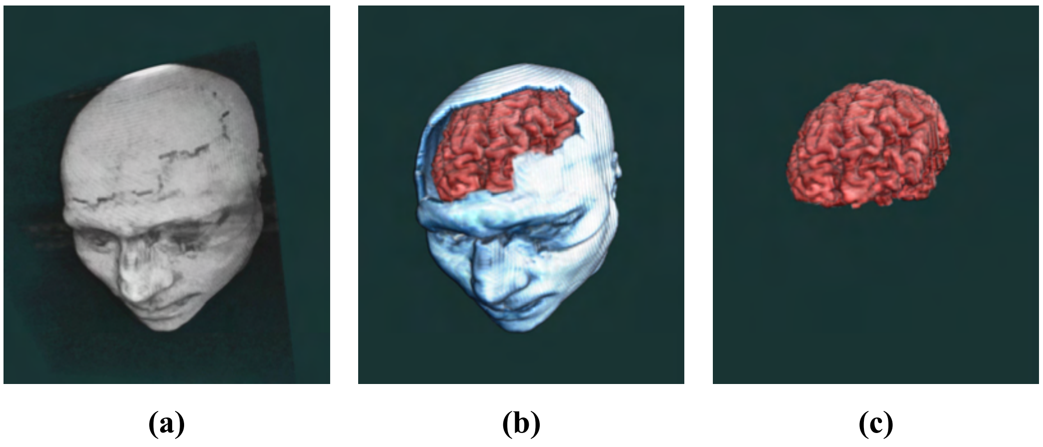


Figure 2.5 Example for segmentation.
(a) Without segmentation; (b) Segmented; (c) Only brain segment visualized.

2.4.2 Gradient Computation

Gradient computation is the process to calculate and find the boundary voxels between different materials. The gradient is a measure which tells how quickly values of the voxels change and the direction of that change. This information has an importance in volume rendering as it gives a lot of information about the structures inside the dataset. For example, two different tissues in an MRI dataset will have two different intensity values, and gradient value of the voxels which were located at the boundary between these two different tissues will be significantly high. The direction value calculated in gradient computation also gives the information about the three dimensional orientation of the boundary [6, pp. 67].

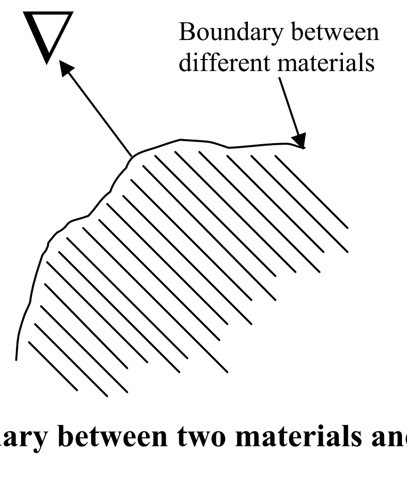


Figure 2.6 Boundary between two materials and the gradient vector.

∇ = [∇x, ∇y, ∇z] is a gradient, which is a three dimensional vector which points a direction in the three dimensional space. This direction gives us the information about the orientation of that voxel. The magnitude of this vector gives the information about how quickly values of the voxels change around that voxel which is given as [6, pp. 67]:

$$\nabla = \sqrt{(\nabla x)^2 + (\nabla y)^2 + (\nabla z)^2}$$

If the magnitude of a voxel's gradient is zero, this means that there is no change in the values of the neighborhood voxels. On the contrary, if the magnitude has a significant value, it can be told that, this voxel is located at a boundary.

It is recommended to read the *Interpolation-Resampling* section at this point, because some knowledge about interpolation is required for a better understanding of how gradient computation works.

In order to understand the logic behind the gradient computation, here is a one dimensional example [6, pp. 68]:

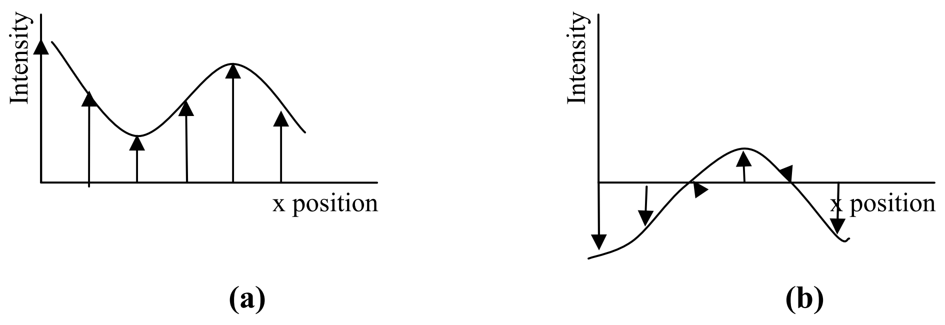


Figure 2.7 Continuous function
(a) Underlying continuous function of the discrete data;
(b) Derivative of the continuous function and the sampled points.

In this example, another step in which the derivative of the continuous function is calculated is used so that the information of how quickly the continuous function change is obtained.

There are many different methods that calculate the gradient of a dataset. The central difference gradient estimator method is one of the mostly used gradient computation methods which is fast and easy to implement, but not very high in quality.

The definition of the central difference gradient estimator is [6, pp. 69]:

$$\nabla_x = f(x-1, y, z) - f(x+1, y, z)$$

$$\nabla_y = f(x, y-1, z) - f(x, y+1, z)$$

$$\nabla_z = f(x, y, z-1) - f(x, y, z+1)$$

In this formula f(x,y,z) is the function that gives the value of the voxel at the position (x, y, z) in the volumetric dataset.

$\nabla = [\nabla_x, \nabla_y, \nabla_z]$ is the gradient vector of the point (x, y, z) which is consist of the components ∇_x, ∇_y, ∇_z. The central difference gradient estimator can also be calculated using a convolution kernel: [-1, 0, 1] (see the next section for the definition of convolution, if reader is not familiar with convolution).

Using this one dimensional kernel on each three axis, the components of the gradient vector ∇_x, ∇_y, ∇_z shall be obtained.

The central difference gradient estimator method uses six voxels to calculate the gradient vector. Other methods use different kinds of operators. Sobel operator uses 26 of the neighboring voxels, to estimate the gradient vector. 26 point neighborhood operators are usually better at estimating the gradient; however they are more expensive in computation. Sobel operator is a well known image processing operator which uses the kernel in Figure 2.8 [6, pp. 72] for three dimensional datasets

$$\begin{array}{ccc} x=-1 & x=0 & x=1 \\ \begin{bmatrix} -2 & 0 & 2 \\ -3 & 0 & 3 \\ -2 & 0 & 2 \end{bmatrix} & \begin{bmatrix} -3 & 0 & 3 \\ -6 & 0 & 6 \\ -3 & 0 & 3 \end{bmatrix} & \begin{bmatrix} -2 & 0 & 2 \\ -3 & 0 & 3 \\ -2 & 0 & 2 \end{bmatrix} \end{array}$$

Figure 2.8 Three dimensional Sobel gradient operator

3x3x3 convolution kernel shown in the figure is applied to the 26 voxels around the voxel whose gradient vector is being computed. In order to calculate the three components of the gradient vector, three different passes should be applied on each axis (x, y, z). The kernel shown in the figure is used for the z direction to get the x component of the gradient vector. In order to compute y and z components of the gradient vector, this kernel need to be rotated, according to direction of the axis.

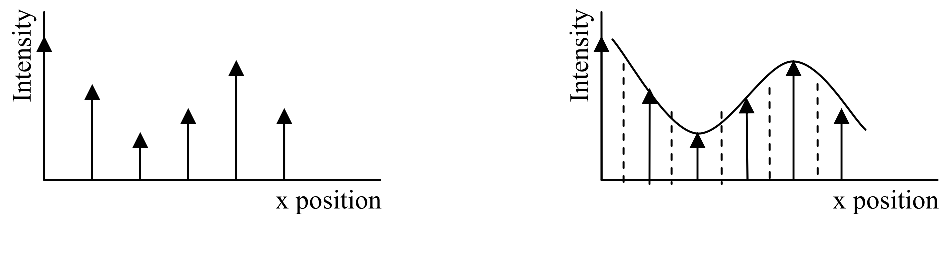
Another operator is *intermediate difference* operator for which the convolution kernel for one dimension is: [-1 1].

Convolution kernel looks similar to the central difference gradient operator, however this operator uses the voxel that gradient vector is being computed so that it can be able to register the very fast changing of values in the dataset by subtracting two neighboring voxel intensities.

Gradient is used in two phases of the volume rendering pipeline: shading stage and classification stage. In computer graphics, in order to increase the realism of the rendered scene view, many different kinds of methods are used. Shading techniques increase the render quality by using the information of the position of the light sources, material properties that has been assigned to the polygons, color of the polygon and the surface normal of the polygons. In volume rendering, the dataset does not consist of polygons but voxels. In other words, there is no surface normal as there is no surface in the data format. However, the gradient vector information for each voxel can be used as surface normal in the illumination model, and more realistic views can be rendered by computing the output according to the light position, color, material properties assigned to each voxel.

2.4.3 Interpolation - Resampling

Interpolation is computing the intermediate values between two discrete points. In the sampling process, the data stored are discrete value of a continuous function. Interpolation is meaningful if we have an idea about what the continuous function is. Figure 2.9 (a) shows a sampled one dimensional discrete points (through x axis). Figure 2.9 (b) shows the interpolated points between discrete sampled points, according to the continuous function [6, pp. 68].



(a) **(b)**

Figure 2.9 Interpolation
(a) Discrete data; (b) Interpolated according to the continuous function.

There are different methods for the computation of the interpolated values. It is usually impossible to find the continuous function for the whole dataset, especially in the three dimensional volumetric datasets. In order to calculate the interpolated values of points, computing the interpolation according to the point's neighboring discrete sampled points is meaningful. This can be done by calculating a weighted sum of the surrounding sampled points. This method can be done by using convolution algorithm with different kind of interpolation kernels. Interpolated kernels are overlays, which we place them on the top the values need to be interpolated. Interpolation kernels are centered at the points that we are interested in to find out the interpolated values. Every position that the interpolation kernels intersect a known sampled value in the dataset, the sampled value and the kernel value are multiplied. The newly interpolated value is achieved by the summation of these multiplied discrete values. Examples for different interpolation kernels are shown in Figure 2.10 [6, pp. 105].

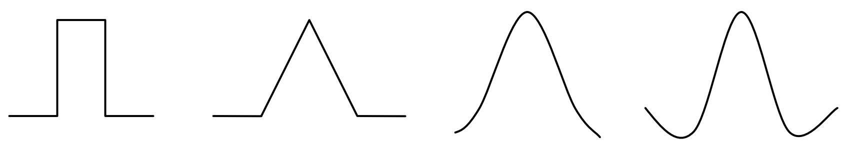


Figure 2.10 Different interpolation kernels

"The one dimensional interpolation kernels can be applied to interpolate in two and three dimensions if the kernel is separable. A two dimensional function is considered separable if it can be decomposed as follows" [6, pp. 105]:

$$f(x,y) = g(x) \cdot h(y)$$

Interpolation calculation of a three dimensional dataset can be done in three stages, where each stage is done on a different axis (x, y, z). These stages are shown in the following figure [6, pp. 106].

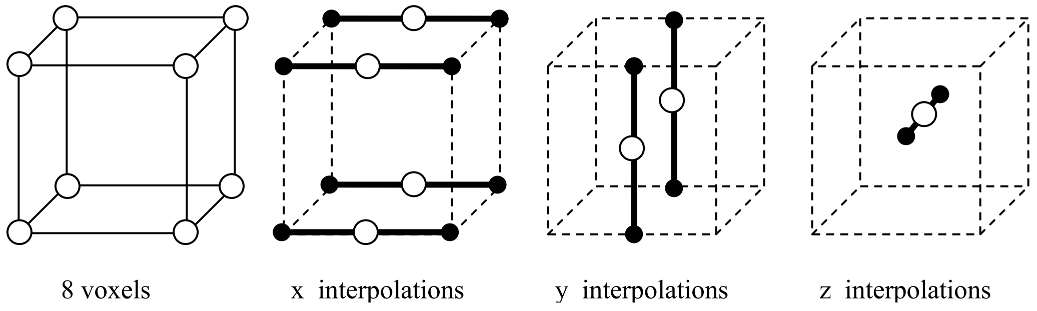


Figure 2.11 Interpolation in three dimensions

The simplest method of interpolation is nearest neighbor method. In this method nearest sampled point's value is used as interpolation point.

Linear interpolation method is one of the most popular interpolation method used in image and signal processing. It is called bilinear interpolation when it is used two dimensional signals, called trilinear interpolation when it is applied to three dimensional signals.

Linear interpolation is a computationally expensive method that nearest neighbor interpolation, because it assumes that there is a linear relationship between the points to be interpolated. It is computed by the formula [6, pp. 110]:

$$f(d) = \frac{f(x_1) - f(x_0)}{x_1 - x_0} \cdot d + f(x_0)$$

where d is the interpolated point's distance from the point x_0. An example for bilinear interpolation is shown in the following figure [6, pp. 111]:

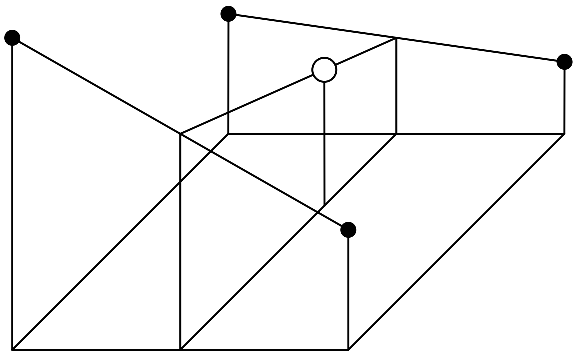


Figure 2.12 Bilinear interpolation

2.4.4 Classification

Classification stage in the volume rendering pipeline, enables the structures to be visualized without extracting surface or explicitly defining the shape in the volumetric dataset. This stage is the most powerful ability of volume rendering which makes it more useful with compared to surface rendering methods. In the surface rendering visualization techniques, surface of the structures in the volumetric dataset must be extracted as a pre processing operation before the rendering phase. In the pre-processing stage of the surface rendering [3], [2], [7], the surfaces need to be decided if present or not, and there might be some errors occurred in the surface extraction. These errors might lead to rendering some surfaces that are not existed in the dataset. Classification step of the volume rendering pipeline is not a binary decision process that decides if a surface is existed or not. The surfaces are made visible by assigning *opacity* to the voxels. Opacity is a measure between 0 and 1 which defines how much transparency shall be applied to the voxels, in other words, how much light that passes through will be absorbed by that voxel. For example, while visualizing a patient's MRI volumetric dataset, if the voxels which are at a position where the patient's brain exists are assigned as opacity 1, and the rest of the voxels in the dataset are set to 0 opacity, the rendering result will be the visualization of only the brain voxels stored in the dataset. In the classification stage of the volume rendering pipeline, the main aim is to assign opacity values to the each voxel stored in the dataset. However, this process might be very complex and might require sophisticated methods to be applied in order to extract meaningful structures from the raw data of voxels in the dataset. Segmentation is an example for extracting structures from a dataset.

The assignment of opacity value to the voxels is done by the help of information extracted from the dataset, like the intensity / color value of the voxel or the gradient magnitude. This process is called opacity transfer function. Before understanding the transfer function methods, histogram[1], which is a very useful tool for designing the transfer function, need to be known.

In almost all of the datasets, there is an inherent noise which differs from one dataset to another according to different conditions or methods in the acquisition process of the volumetric data. Histograms help to produce meaningful filters to avoid noise. While building the opacity transfer function, many different properties can be used as input [6, pp. 89]:

$$\alpha_1 = O(I_i, |\nabla_i|, \ldots)$$

where O is an example for the opacity transfer function, which has input of the value of the voxel (I_i) and the local gradient magnitude ($|\nabla_i|$).

[1] A histogram, in image processing, shows that how many times a pixel of a value appears in the image. This is same with the volumetric datasets. The vertical axis shows the number of occurrence of the voxel value (frequency) and the horizontal axis shows the values appear in the dataset. Histogram is a useful tool for determining the opacity transfer function, because having the information about the spread of the voxel intensities can help to construct the transfer function.

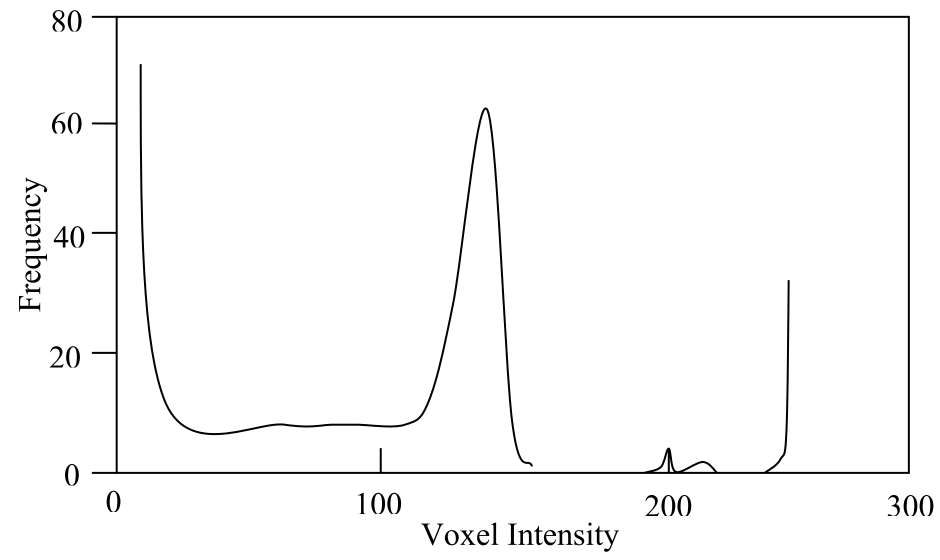


Figure 2.13 Histogram of a CT dataset.

For example: the histogram of a CT shown in the Figure 2.13 gives the information that there is a peak between 120 – 150 intensities and the intensities below the 100 seems to be noise. In order to filter out the voxels in [120,150] intensity range, we built an opacity transfer function by assigning "0" opacity value in this range and assigning high opacity values to the other voxels. The output shall be different if the filtered intensity range is changed or the gradient magnitude is also used. As gradient magnitude shows how quickly the voxel intensity values change, filtering the voxels that have less gradient magnitude values by assigning small opacity value, the voxels that happen to be on a surface will be rendered and the structures in the volumetric dataset could be viewed.

Here is another example for opacity transfer function, which helps to visualize the voxels having the intensity value of f_v and neighboring voxels that has a significant gradient value[6, pp. 94]:

Opacity value for the voxel i	**Case**
$1-\frac{1}{r\lvert\nabla_i\rvert}\lvert f_v - I_i\rvert$	If $\lvert\nabla_i\rvert > 0$ and $I_i - r\lvert\nabla_i\rvert \le f_v \le I_i + r\lvert\nabla_i\rvert$
1	If $\lvert\nabla_i\rvert = 0$ and $I_i=f_v$
0	otherwise

where,

$\lvert\nabla_i\rvert$ is the gradient magnitude at voxel i,

I_i is the intensity value of the voxel i,

r is a constant, which is "the maximum a voxel's intensity can deviate from f_v"

Classification process is mainly implemented with an interactive user interface, so that the user is able to change the parameters in real time according to the type and histogram of the volumetric dataset. Segmentation results or other kinds of labeled information and other kinds of filtering techniques might be used as input in the opacity transfer function.

2.4.5. Shading

Shading phase of the volume rendering pipeline refers to illumination and shading techniques that are well known methods in the conventional computer graphics in order to enhance the quality of the rendering to make it more realistic. Shading methods try to model the geometric scene in a way that more photo realistic effects like shadow, scattering and absorption of the light according to the properties of the material, could be obtained. In volume rendering, the primary goal is not the photo realism, but to get better and more understandable rendered views of the structural information stored in the volumetric dataset. Because volume rendering the datasets may contain information about tissues of a human body, an engine block, a fluid dynamics test, acoustics, etc, in the real world, only the surface of objects could be seen. However, volume rendering aims to visualize the inside of the object, and use the shading phase in order to visualize that as realistically as possible.

In computer graphics, illumination is defined as a model, which describes all the light striking a particular point on a surface that has particular material properties.

"An Illumination Model describes the interaction of light incident with a surface in terms of the surface properties and the nature of the incident light."[44] A shading model is a framework that an illumination model fits, in other words, shading is a model which determines when and which illumination model shall be applied to a point, and what parameters shall the illumination model use. The result of shading is the color of a point in the environment that is being rendered, according to the physics that how light shall shine on that point, and the position-angle of light and the rendering reference (user's eye) orientations in the space. The computation of this physics is achieved by the illumination model used. In order to build a model and obtain realistic results, first, how light interacts with the surface of the objects should be understood [6, pp. 67].

"The complete physics of the interaction of light with surfaces is very complex and it is usual to use various empirical approximations to the true physics in Computer Graphics. The reason for this is the vast computational demands made by a good physical illumination model. However, acceptably realistic results can be produced fairly quickly using a quite simple illumination model" [44].

In order to compute how the light shall behave when it hits a surface, there should be information about the shape of that surface. Surface normal of a point gives this information and enables the model to calculate how the reflections shall be. In the volumetric datasets, the gradient information of a voxel can be used as surface normal for that voxel.

2.4.5.1 Phong Illumination Model

Illumination models, in general, aim to simulate the behavior of light reflection on a surface according to the observer position, light source position, surface shape and material properties. For example, a black billiards ball under a single, white spot light shall be observed as a white light shinning on the surface of the ball. However, if the observer changes the position and look at the ball at a different angle, it would be seen that, the white shinning part of the ball is now black.

In other words, every point on the scene need to be calculated during the rendering process.

The Phong Illumination Model [8] deals with three types of light reflection, namely: (i) *Ambient Reflection:* The reflection of light that arrives at the surface of the object from all directions; (ii) *Diffuse Reflection:* The reflection of light from non-shiny surfaces in which the light is scattered equally in all directions; (iii) *Specular Reflection:* The reflection of light from shiny or mirror like surfaces.

The visualization of a point of an object is the intensity of light that is reflected from the surfaces of that object; and the intensity of the light in Phong shading is calculated by summing over the above three types of reflection. If the model is rendered in color, this process shall be done for each of the color components: red, green and blue.

Ambient Light:

The Phong model assumes that, the ambient light has the same intensity everywhere in the scene that is being rendered. The ambient light has no single point position, so there is no angle of ambient light with respect to the position-shape of the object being rendered.

Phong illumination model with ambient light illumination can be formulated as follows:

$$C_o = C_a\, k_a\, O_d$$

where,

C_o: Resulting color computed for rendering of a point

C_a: Color of the ambient light. (The color consists of red, green and blue intensity components, so the computation is done for each single component of color, separately.)

k_a: Material property of the surface. It is called *the ambient reflection coefficient.* This coefficient is a number between 0 and 1 which is assigned as a property of a

material in the scene. k_a for a black surface is smaller than k_a for a white surface as the color black absorbs light more than white.
O_d: The visualized object's diffuse color (assigned in the material properties of the surface)

Diffuse Reflection:

Diffuse reflection is the scattering of light in all directions. If a surface is a perfectly diffusing surface, an incoming light ray shall be reflected to every angle. Thus, in the rendering result, intensity of a point on a surface will not depend on the position of the user, but will depend on the properties of the material, color and distance of the light source, and the angle of the light ray. The color of a surface is obtained by the light absorbing property of a surface. For example: A red billiard ball is observed as red if a white light source exists, because the material absorbs the green and blue colored light rays and scatters the red light rays. If the material of the ball has no specular light reflection, in other words, it has a perfectly diffuse reflecting surface; it will appear dull-matt. Figure 2.14 [45] shows how diffuse reflection occurs.

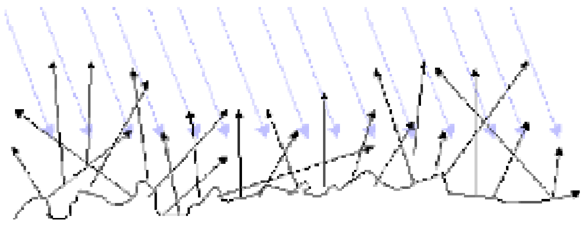


Figure 2.14 Diffuse reflection

While computing the diffuse reflection, often the distance between the light source and the surface is not taken into account. So, the light source is thought to be infinitively far away and the light intensity does not change at any distance. This is called directional lighting. In directional lighting the only parameter that will be used in computing the rendering result for a point, is the angle between the surface and the rays of the light.

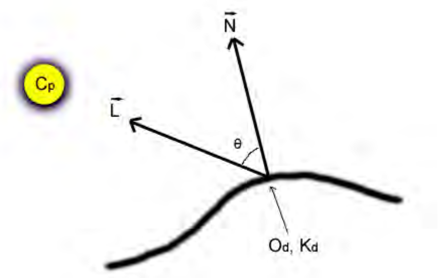


Figure 2.15 Diffuse reflection dependency of angle between light position and surface normal

In Figure 2.15, N is the surface normal of the point that is going to be shaded; L is a vector that points the light source; Θ is the angle between L and N; k_d is the *diffuse reflection coefficient* of the surface; C_p is the color of the light source.

When the diffuse reflection parameters are added to the previous equation, the Phong illumination model, the formula becomes:

$$C_o = C_a k_a O_d + C_p k_d O_d \cos\Theta$$

This equation shows that, when the angle between the light source and surface normal is 0, the diffuse reflection of the surface becomes the maximum; when it is 90 degrees, no diffuse reflection is added to the C_o (resulting color)

If the L and N vectors are normalized, the formula can be changed as follows:

$$C_o = C_a k_a O_d + C_p k_d O_d (N \cdot L)$$

where, $(N \cdot L)$ is the dot product between L and N vectors, which is equal to cosΘ.

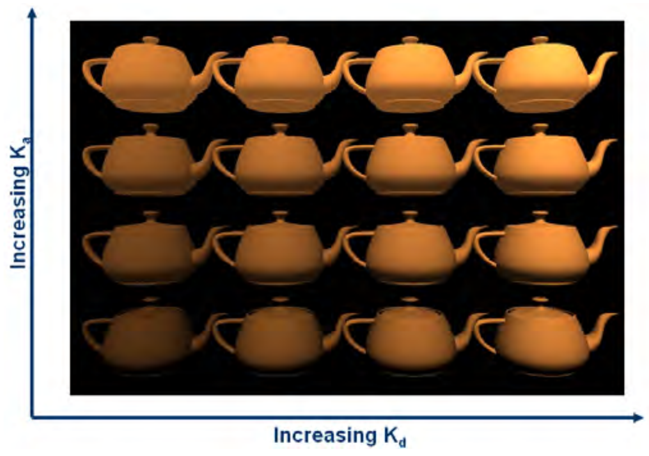


Figure 2.16 Effects of k_a, k_d changes

Figure 2.16 [46] shows the difference of diffuse and ambient reflection. When the diffuse reflection coefficient (k_d) increases, shadows occurred because of the directional light reflection which makes the object look more photo realistic.

Specular Reflection:

While rendering shiny surfaces like polished metal, a glossy plastic, specular reflection is necessary for a more photorealistic result. In shiny surfaces, a highlight or a bright spot is seen [47].

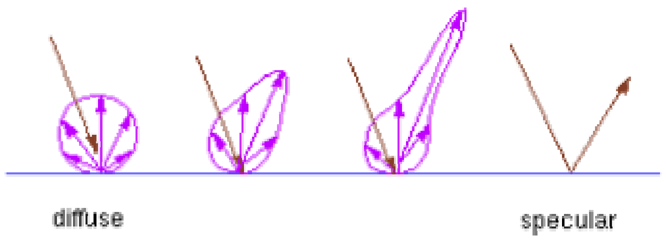


Figure 2.17 Diffuse to Specular reflection

The bright spot seen on the surface is dependent on where the surface is seen. Figure 2.18 shows that the color of the rendering result for a point also depends on the angle between the reflection direction and the position of the viewer.

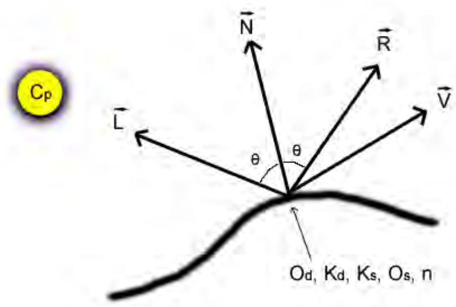


Figure 2.18 Specular Reflection

When the specular reflection is added, the Phong illumination model becomes:

$$C_o = C_a k_a O_d + C_p [k_d O_d (N \cdot L) + k_s O_s (R \cdot V)^n] \quad \textbf{Equation (2.1)}$$

where,

k_s is the *specular reflection coefficient*

O_s is the specular reflection color

R is the normalized reflection vector,

which is the mirror of vector L about the normal N

V is the vector from the point to be shaded to the viewer

$(R \cdot V)$ is the dot product between R and V vectors,

which is equal to cos(angle between the vectors R and V)

n is the *specular reflection exponent*

The presence of the vector V shows that the result of rendering shall be dependent upon the position of the viewer.

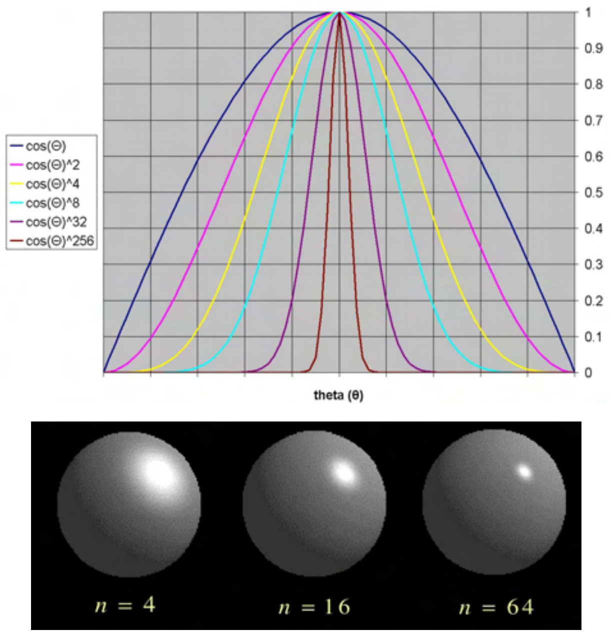


Figure 2.19 Effects of specular reflection exponent changes

The specular reflection exponent n is used in order to increase the sharpness of the edges of the highlighting dots because of specular reflection. Figure 2.19 [46] shows results for different specular reflection exponent values.

2.4.5.2 Shading Methods

"Gouraud and Phong shading models both use the illumination model of Phong that was given in Equation (2.1) or some close derivative. The difference lies when and where the illumination model is applied" [6, pp. 79].

While rendering a geometric model that consists of polygons, the material properties and surface normals of the objects are assigned on the vertex points of the polygons (here texture mapping is not taken into account). One way of rendering is to use *Flat Shading* which assumes the same surface normal for every point that exists on the polygon. However, this would lead to discontinuities on the surface between the polygons, and a non smooth rendering result shall be obtained as shown in Figure 2.20 (a) [48] and Figure 2.22 (a) [48]. The smoothness could be achieved by increasing the number of polygons of the geometric model (Figure 2.20 (b)). However, this would increase the computation time of the rendering process a lot.

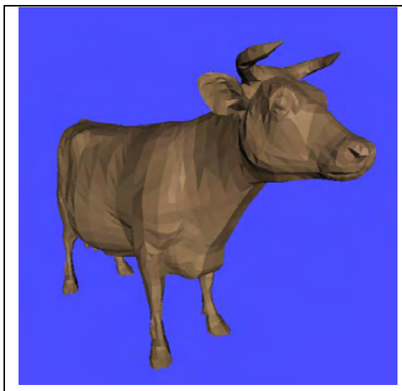
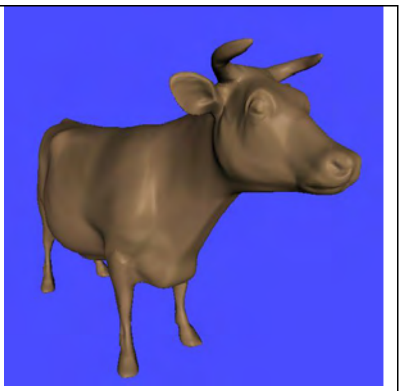
	
(a)	**(b)**

Figure 2.20 Flat Shading
Model used in the image (b) consist of 16 times more polygons than the model used in the image (a)

Gouraud shading model solves the discontinuity problem by interpolating the non vertex point colors across the edges. Usually linear interpolation method is used, but other kinds of interpolation techniques can also be applied. First the resulting color values for the vertex points are calculated; afterwards the remaining points (pixels of the rendering result) are calculated by interpolation for the each red, green and blue component of color. Figure 2.21 [48] is an example for this method.

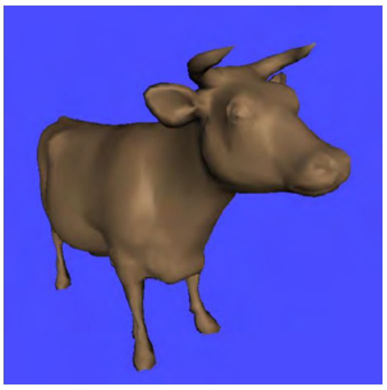


Figure 2.21 Gouraud Shading

Phong shading method's difference from Gouraud shading is that; in Phong shading, color values of the non vertex points for the resulting rendered image are computed by interpolating the normals of the vertices. In other words, first the surface normal of the non vertex point is computed by interpolating the vertex normals of the polygon; afterwards, the Phong illumination model is applied on that point to compute the resulting color value.

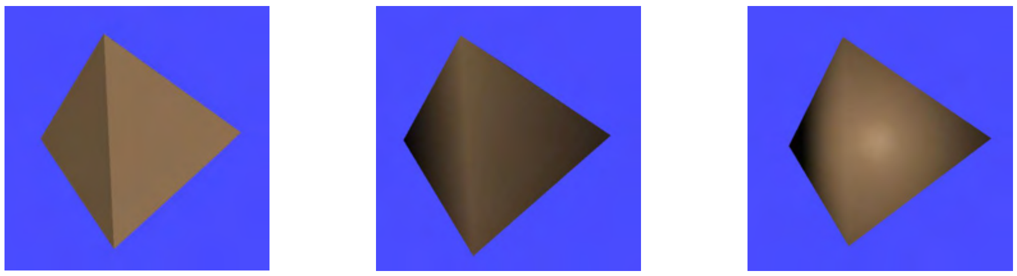

(a) (b) (c)

Figure 2.22 Different Shading Methods
(a) Flat Shading; (b) Gouraud Shading; (c) Phong Shading.

As seen in the above figure, Phong shading model has an advantage over Gouraud shading model in computing a more accurate specular shading result. However, in order to apply the Phong shading, the normals should be interpolated. Although the normals at the vertex points have been normalized, the new interpolated vectors shall not be normalized in general. The normalization computation process requires a high computation time.

2.4.6. Compositing

The term compositing is the method of combining two or more images [49].

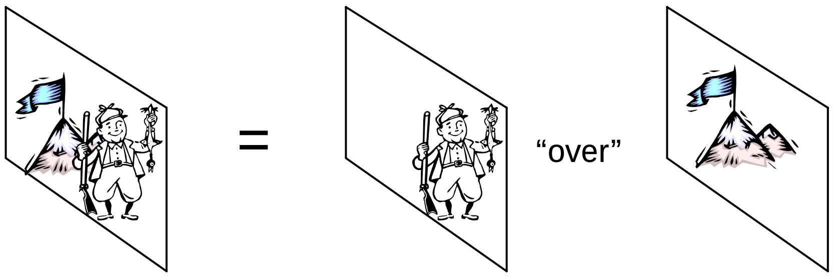


Figure 2.23 Intervisibility of two images

The result of rendering is a digital two dimensional image that consists of pixels. Each pixel can carry only one color value, but may represent hundreds of values that present along the ray of that pixel. Compositing is the accumulating of these values into one. A pixel value may contain the translucency information as well.

The alpha value is generally used for defining the opacity property of that pixel. While combining two different pixels into one, the values of red, green, blue and alpha (RGBA) can be combined with more than ten different ways [6, pp. 121]. However, not all of the combining techniques are meaningful for the volume rendering purposes. In computer graphics, the term compositing is also called as blending, and these different blending techniques shall be given in the blending topic of OpenGL.

Compositing is the last phase of rendering the volumetric dataset in the volume rendering pipeline. There are two basic methods of compositing; back-to-front, front-to-back; and the main difference of these methods is the direction that is taken along the ray.

2.4.6.1 Front-To-Back Compositing

In order to compute the resulting color of each pixel for the result image of rendering, front-to-back compositing methods draw a ray that starts from the pixel (viewer) and goes through the volumetric dataset. The casted ray may pass through the space between the voxels as shown in Figure 2.24. First of all, an interpolation method is to be used to calculate the color and alpha values of the newly sampled points a and b. This computation is done according to the shading model that is chosen to be used. This can be a simple direction independent method or a more complicated method like Phong shading.

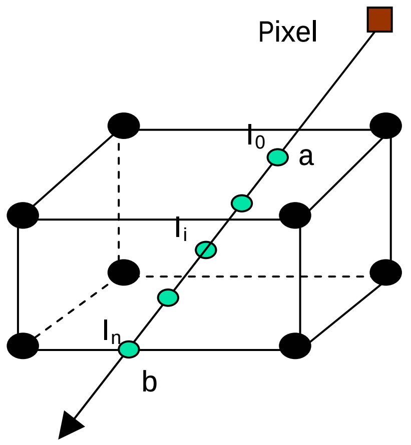


Figure 2.24 Front-to-back compositing

After the interpolation process, the value of pixel can be calculated with this often-used front-to-back compositing equation [6, pp. 125]:

$$I(a,b) = \sum_{i=0}^{n} I_i \prod_{j=0}^{i-1} (1-\alpha_j)$$

where;

I (a,b) is the total intensity accumulated between the points a and b. Intensity here is not the same as color. The relationship between color and intensity is given as: “I = Color * Opacity (α)”. α is the opacity of the point, which is (1-Transparency).

This equation can be rewritten like this [6, pp. 127]:

$$\sum_{i=0}^{n} I_i \prod_{j=0}^{i-1} (1-\alpha_j) = I_0 + I_1(1-\alpha_0) + I_2(1-\alpha_0)(1-\alpha_1) + ... + I_n(1-\alpha_0)...(1-\alpha_{n-1})$$

This equation shows an “over” relation like this: “I_0 over I_1 over I_2 …. I_{n-1} over I_n“ “This operator was first introduced by Porter and Duff for digital imaging in their 1984 SIGGRAPH paper. Thus compositing means applying the over operator on all sample points on one ray” [6, pp. 127].

The pseudo code of this equation’s implementation can be:

```
I[0 .. n] is the array of intensity values of the points
T[0 .. n] is the array of transparency values of the points
float Transparency = 1.0;
float Intensity = I[0]; // this is the variable which will store the result intensity value, initially assigned as the intensity of the first point.

for ( i = 1; i <= n; i++)
{
      Transparency *=  T[i-1];
      Intensity += Transparency * I[i];
      if  ( Transparency is 0.0 )
            Brake;
}
```

The implementation shows that until the transparency value is zero or very close to zero, the loop can be stopped before computing all of the points which would require a high computation time.

2.4.6.2 Back-To-Front Compositing

Back-to-front compositing method, compute the intensity value, starting from the most far point to the nearest point with respect to the user.

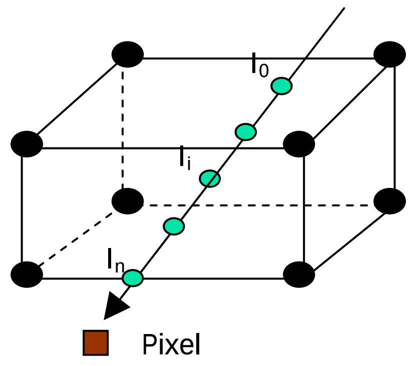


Figure 2.25 Back-to-front compositing

It is formulated as [6, pp. 128]:

$$I(a,b) = \sum_{i=0}^{n} I_i \prod_{j=i+1}^{n} (1-\alpha_j)$$

The equation for back-to-front compositing is very similar to the equation given in the front-to-back compositing method. The difference shows itself in the implementation [6, pp. 129]:

`I[0 .. n]` is the array of intensity values of the points

`T[0 .. n]` is the array of transparency values of the points

`float Intensity = I[0];` // this is the variable which will store the result intensity value, initially assigned as the intensity of the first point.

```
for ( i = 1; i <= n; i++)
{
      Intensity = Intensity * T[i] + I[i];
}
```

This implementation has an advantage over front-to-back implementation example, because there is no variable kept for the accumulated transparency, and that shall decrease the computation time. However, in back-to-front implementation example, there is no control that can end the loop before processing all of the points.

2.4.6.3 Maximum Intensity Projection (MIP)

Maximum intensity projection compositing technique is a simple method that finds the maximum intensity value on the ray.

The pseudo code for MIP can be:

`I[0 .. n]` is the array of intensity values of the points

`float maxIntensity = I[0];` // this is the variable which will store the maximum intensity value, initially assigned as the intensity of the first point.

```
for ( i = 0; i <= n; i++)
{
      if ( maxIntensity < I[i] )
      maxIntensity = I[i];
}
```

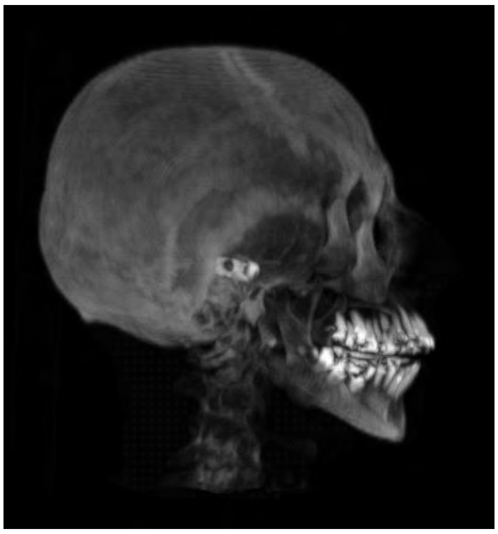


Figure 2.26 Maximum intensity projection of a human head

An example for maximum intensity projection is given in Figure 2.26 [50]

2.4.6.4 X-ray Projection

X-ray projection method is another method of compositing, in which the values across the ray are added.

The pseudo code for X-ray projection can be:

`I[0 .. n]` is the array of intensity values of the points

`float Intensity = I[0]; //` this is the variable which will store the X-ray intensity value, initially assigned as the intensity of the first point.

```
for ( i = 0; i <= n; i++)
{
      Intensity += I[i];
}
```

However, the intensity value calculated with this loop should be normalized, in case some values might exceed the maximum value that the rendered image pixels can have.

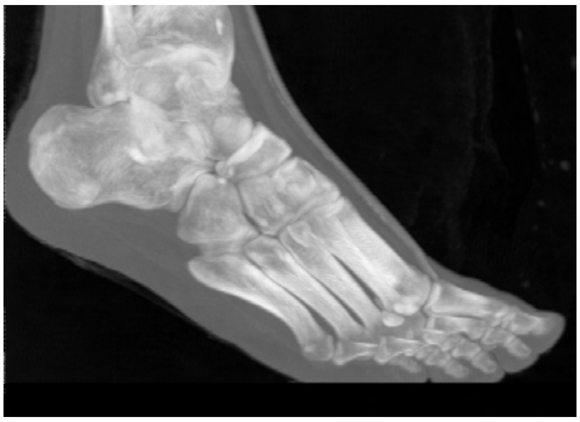


Figure 2.27 X-ray projection of a human feet

An example for X-ray projection is given in Figure 2.27 [51].

2.5 Volume Rendering Techniques

Volume rendering pipeline described here, consist of some general operations that the volumetric data is processed for rendering. There are many different approaches for implementing volume rendering applications. The sequence of the data flow and applied processes described in the volume rendering pipeline may vary from one technique to another. Some of the different approaches for implementing volume rendering are:

Image-order volume rendering
Object-order volume rendering
Shear-warp method
Texture mapping used for volume rendering
Constructing special purpose hardware for volume rendering

2.5.1 Image-Order Volume Rendering

In image-order approach to volume rendering, the color values of each pixel on the resulting rendered image plane are determined. Ray casting is an example for image-order method, which casts rays from the pixels of the image plane to the volume. The accumulation of the resampled points that are on the ray passes though the volume is done by front-to-back order approach [9], [10].

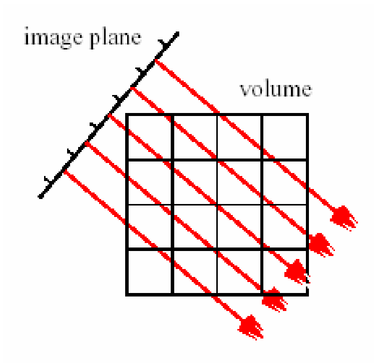


Figure 2.28 Volume Raycasting

There are many approaches in image-order volume rendering, to increase the performance and capability of ray casting method. Some of them are:

Image-Space Coherency: The image plane, which is going to be rendered, would have some sort of coherency between the pixels. In other words, if pixels which have a common neighboring pixel, have the same color value; the probability of common neighbor pixel's value is the same color, is very high. So the ray casting is done for not every pixel on the image plane, and the empty pixels are interpolated afterwards. This is called *image-space coherency* [9].

Object-Space Coherency: The sampling rate of ray casting has a great effect on the computation time of the algorithm. Usually the volumetric datasets contain some regions that has uniform or similar color values. *Object-space coherency* technique tries to increase the performance of ray casting algorithm, by initially sampling the points through the ray, at a low frequency. Then the sampled values are examined, and if two consecutive sampled points have a large difference in color value, new samples are taken between them. This aims to approach increase the performance of ray casting, without decreasing the detail of the rendering [11].

Template Based Ray Casting: If the projection method used for rendering is orthographic viewing, a coherency between the rays can be obtained, because, even though they are from different origin, they have the same slope. The method *template-based ray casting* pre-compute and store templates of the points to be sampled. The computation time required for casting the ray into the volume is decreased by applying the ray templates pre-generated before rendering [12].

C-Buffer: While rendering a volumetric scene interactively, the difference between the consequent frames are usually small. The *C-Buffer* use this feature to increase the frame rate of the rendering process. While the image plane is computed, the coordinate of first non-empty sampled point is stored in the pixel. This information is used for estimating the initial position of a ray in the subsequent frame. For example, if a rotation is to be computed, pixel values of the following frame are calculated by transforming the C-Buffer information according to the rotation, and the coordinates that might become masked are eliminated [13].

Empty Cell Skipping Methods: The volumetric datasets usually contain large spaces of fully transparent voxels. There are many approaches to avoid sampling such regions and increase the computation performance in ray casting. *Hierarchical spatial enumeration* method [14] preprocess the volumetric dataset and create a hierarchically indexed, a binary pyramid for the volume. When a ray is sent to the volume, it passes through the first level of the pyramid. If a non empty cell is reached, more detailed cells stored in the lower level of the pyramid are used. *Space leaping* is another method for passing the empty cells (transparent voxels) of the volumetric data [15], [16], [17]. In this method, the volumetric dataset is pre processed and the voxels which are fully transparent are labeled with a value that shows the distance of the nearest non empty cell; so that sampling distance can be increased safely.

2.5.2 Object-Order Volume Rendering

Object-order volume rendering methods determine how the volumetric data sample affects the pixels of the image plane. An object-order algorithm computes through sampled points in the dataset, and project it onto the image plane [10].

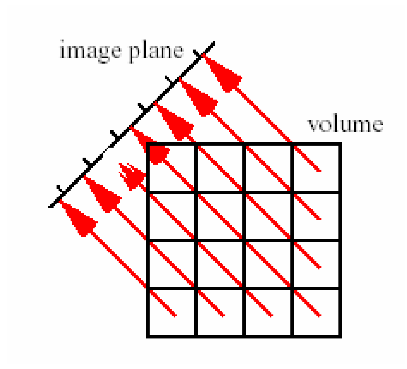


Figure 2.29 Object-order volume rendering

Object order volume rendering methods can be classified as *splatting* and *scan line cell drawing* algorithms.

Splatting algorithm [18] developed to improve the performance of volume rendering at the price of less accurate rendering result. This technique is rather complicated and it will not be defined in detailed here. It approximates a projection called *Gaussian splat*, which depends on the color and opacity of the voxel. The projection is made by splatting every voxel onto the image plane by compositing on top of each other.

Scanline cell drawing [19] methods treat each of the voxels of the dataset as geometric surfaces (like hexahedron, tetrahedron, a square or a plane perpendicular to the image plane) and split the resulting scan line according to the distance of the voxels from the image plane.

2.5.3 Shear-Warp Method

Shear-warp method [20] is considered to be the fastest volume rendering algorithm (software based). The slices in the volumetric dataset are applied a shear transformation as shown in the Figure 2.30. The shear transformation changes all viewing rays parallel to the axis of the volume array which is the transformed volume (called *sheared object space*). This enables the image plane and the volume to be traversed simultaneously. An intermediate image is created as the result of compositing and a two dimensional transformation is applied to the intermediate image in order to obtain the final rendered image [20].

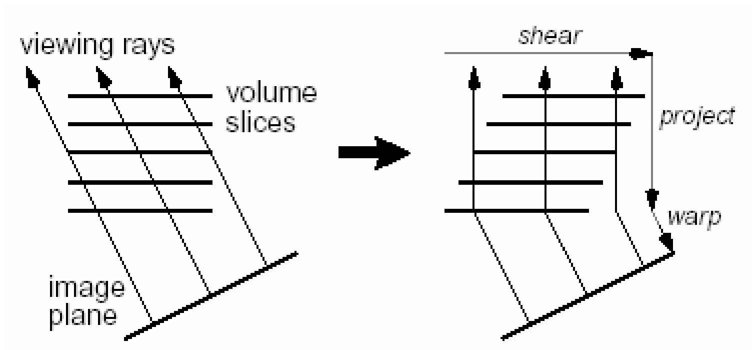


Figure 2.30 Shear-Warp algorithm mechanism (for Parallel Projection)

2.5.4 Volume Rendering Using Texture Mapping

Due to the latest vast amount of increases in the performance of GPU hardware, some techniques were developed for implementing volume rendering using the graphics hardware which enables more intractability by higher frame rates of rendering. This subject shall be explained in detail in *Chapter 3*.

2.5.5 Special Purpose Hardware for Volume Rendering

Due to the high computation time required for volume rendering, many researchers build special purpose hardware architectures for volume rendering. VOGUE [21], VIRIM [22], VIZARD II [23], EM Cube [24] are some of these architectures.

CHAPTER 3

VOLUME RENDERING USING PC GRAPHICS HARDWARE

The developments in the gaming and entertainment market lead to a fast evolution of consumer graphics hardware in the recent years. Some of the hardware developer companies like NVIDIA [52] and ATI [53] have produced state of art consumer graphics chips, and these chips offer a level of programmability with a high performance on a cheap personal computer that was only possible to be performed in high price traditional workstations. This success in the production of hardware not only increased the reachablity of performance but also employed the use of some rendering algorithms that previously could not be used for real time rendering.

Volume rendering algorithms have high computational demands. The major problem faced while using PC graphics hardware is the amount of texture memory required for storing the volumetric dataset is usually large, and texture fetching operations cause all of the dataset to be transferred over the bus for each frame to be rendered. The newly developed graphics cards present larger texture memories, with increased programmability and flexibility of the Graphics Processing Unit (GPU) including transfer functions, shading, and filtering. These developments gave a new environment for the researches to develop new techniques for implementing interactive volume rendering with a high performance and quality.

3.1 Texture Based Volume Rendering Methods

Texture based volume visualization technique is composed of these phases: First, the volumetric data is sampled as planes and these samples are sent to the texture memory of the graphics hardware, in order to be used for mapping the polygons as textures. Then planes which are placed parallel to the image plane (result image to be viewed), are mapped with the texture. These planes are rendered as polygons and they are clipped by the limits of the texture volume. The resulting slices of polygons are blended together by back to front order, and while each polygon is rendered, its pixel value is blended into the frame buffer with the appropriate transparency, and the volume is visualized. [54]

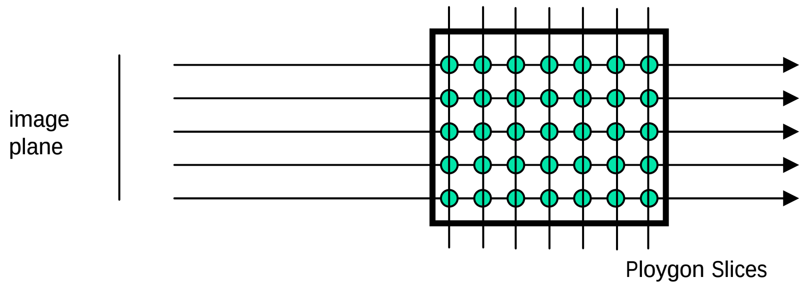


Figure 3.1 Polygonal slices that are mapped with textures.

3.1.1 Volume Rendering Using 2D Textures

Volume rendering using 2D texture mapping is supported by most of the graphics hardware. Volumetric dataset is sampled into two dimensional array or digital images, according to the number of slices to be used while rendering. These images are sent to the texture memory for mapping the polygonal slices as shown in the Figure 3.2 [55].

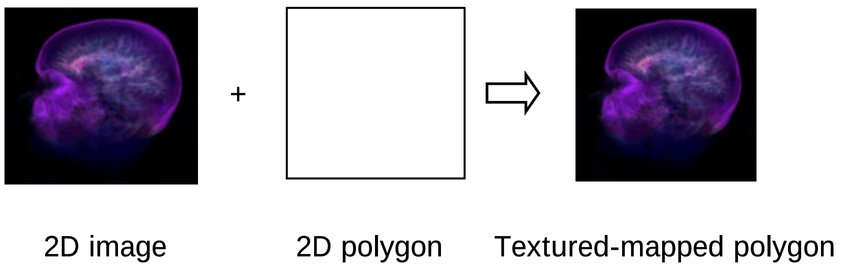


Figure 3.2 Texture Mapping

Using the texture mapping, the performance of graphics hardware allows interactive rendering of the scene. This means that, without resampling the volumetric data and sending the images of slices to the texture memory, the scene can be rotated by applying a transformation to the polygons or changing the eye position as shown in Figure 3.3 [55].

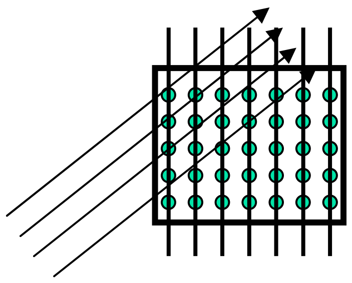


Figure 3.3 Rotated view

However, when the slice planes become parallel to the view of direction, the user cannot see anything in the rendering result, as shown in Figure 3.4 [55].

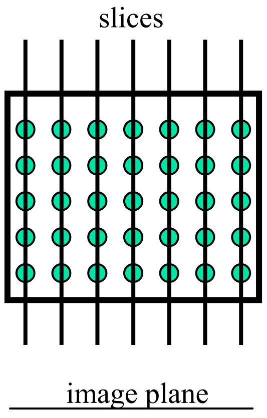


Figure 3.4 No rendering result

In order to solve this problem, during the first stage that the volumetric dataset is sampled as two dimensional slices of images, three different set of slices of images are sampled along each x, y and z axis as shown in Figure 3.5 [55].

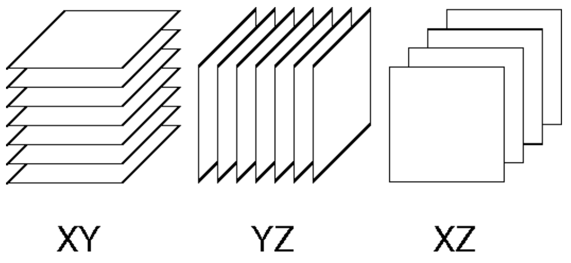


Figure 3.5 Slice sets parallel to the three coordinate planes

After sending the each three set of slices to the texture memory, the set of most suitable that is the most perpendicular slices of 2D texture set according to the viewpoint and view direction is chosen to be rendered. When the position of the viewpoint changes with respect to the volume, 2D texture set that is most closely aligned with the view direction is used. Each slice of polygon that was mapped with texture is rendered from back to front using an appropriate blending method [56].

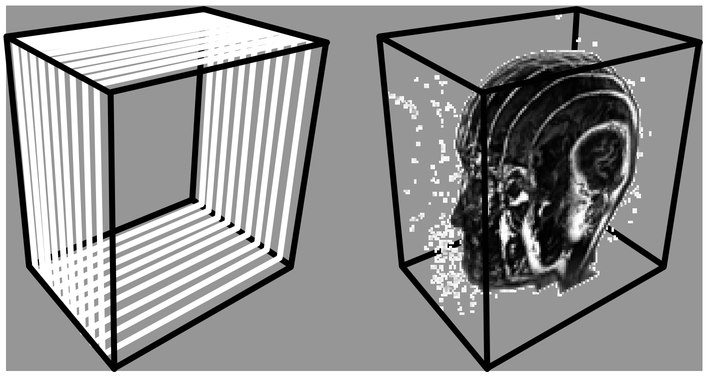


Figure 3.6 2D texture mapped slices

Disadvantages of this method are: The sampled images for the slices from the volumetric dataset, occupy three times time more space in the texture memory as three sets of textures have to be produced. The sampling rate of the resulting render image changes according to the perpendicularity of view direction to the slices shown in Figure 3.7 [55].

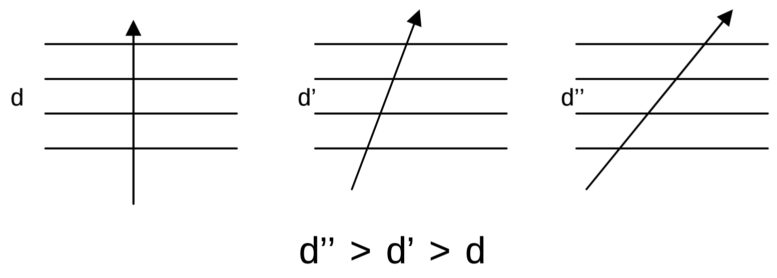


Figure 3.7 Sampling artifact

When the slice set use changes during the rotation of the model, there is a change occur in the intensity of the results as the sampling rate according to the view direction changes. This is called *popping effect* shown in Figure 3.8 [55].

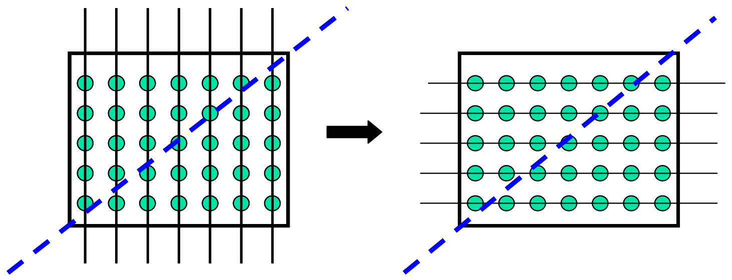


Figure 3.8 Artifact during the change of the slice set.

3.1.2 Volume Rendering Using 3D Textures

Recent developments in the graphics hardware enable to process three dimensional textures. 3D texture supported hardware enables the interpolation of three dimensional texture coordinates to the vertices of the polygons, so that the texture samples are reconstructed by trilinear interpolation [25]. In other words, there is no need to keep the volumetric data as three times itself [26] as it was done in two dimensional texture mapped volume rendering, because trilinear interpolation enables the three dimensional texture to be mapped on surfaces of the polygons. This means that arbitrary slicing through the volumetric texture data can be achieved [55] as seen in Figure 3.9.

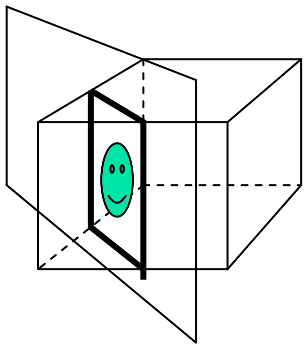


Figure 3.9 3D texture, arbitrary slicing capability

In 3D texture based volume rendering, all of the volumetric data is loaded to the texture memory at once. Transformation operations required for rotating, scaling can be directly done on the mapped texture, as the mapped slice is computed by the graphics hardware with trilinear interpolation, so that changing the texture coordinates on the vertices of the polygons will change the mapped result on the polygon slice [56].

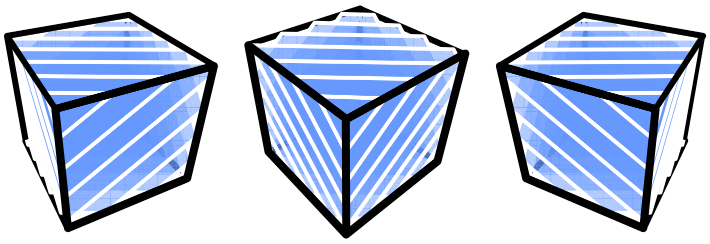


Figure 3.10 Viewing direction aligned slicing

This enables a view aligned slicing capability which overcomes the problem of changing sampling ratio according the viewing orientation faced in the 2D texture mapping volume rendering technique [56].

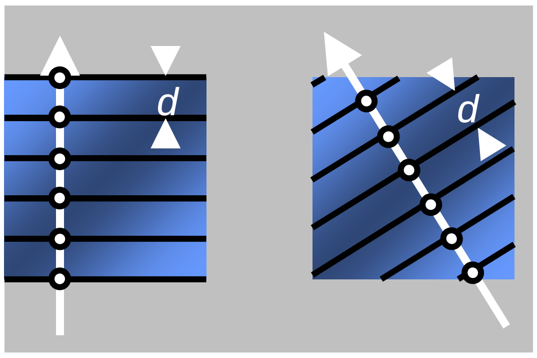


Figure 3.11 Consistent sampling rate.

The sampling ratio can be changed only by changing the number of slices that will be rendered without making any extra computations on the volumetric data. This gives a capability to shift between performance and quality of rendering result during interactively rendering the scene.

Volume rendering by 3D texture mapping also gives the capability to use the planar clipping mechanism of graphics APIs which enables the clipping operations to be done by the graphics hardware.

The implementation built for this work has been done by this method, and the details for the capabilities of this method shall be given in the following chapters.

3.1.3 Sampling Frequency

Both in 2D and 3D texture mapped volume rendering, sampling frequency is an important variable on the quality and performance of rendering process. Sampling frequency is the obtained by the number of slices that are mapped with the texture. There are some factors to be considered while choosing the number of slices to be used in rendering [57]:

Performance: Implementations using hardware accelerated volume rendering method may have two modes for rendering, interactive and detail modes. In the interactive mode, less number of slices can be used in order to increase the frame rate of the rendering process, but that shall lead to a low frequency of sampling which may lead to decrease in the quality of the rendering. Detail mode can render more number of slices and obtain higher quality render results with more detail but with lower frame rates of rendering.

Volume size: While rendering a cubic volume with a view position at the front side, a good rendering quality can be achieved by using number of slices equal to number of voxel count on the view direction axis. However, the dimension of the volume may differ from an axis to another, so making an approximation in the number of slices to be rendered according to the major axis is a good method.

Transparency Issue: It is said that, increasing the number of slices which would increase the sampling rate to be rendered, increases the quality of the final render result. This is not true after a rate that exceeds the original sampling rate of the volumetric dataset. Increase in the sampling rate more than the sampling rate of the original dataset does not give more details. Moreover, over operator used while rendering back to front is not a linear operator. In other words, when more than one sample is taken for a voxel that is semi transparent, the opacity for that voxel would increase at the render result, as it is sampled for more than one and the transparency is decreased. If the sampling rate is to be changed, the alpha values should be rescaled because of this issue.

Introductory information about *Computer Graphics* is given in Appendix B, "*1. Fundamentals of 3D Computer Graphics"* chapter. Brief information about *OpenGL API,* and some detailed information about syntax and commands of OpenGL is given in Appendix B, *"2.OpenGL"* chapter.

CHAPTER 4

IMPLEMENTATION

The main purpose of the implementation is to present an environment that is able to read and visualize volumetric datasets. The motivation for building this implementation is to use the graphics hardware of an ordinary desktop computer that is being used today, and implement a volume rendering application for this hardware and give an interactive environment with some visualization techniques for the user.

Java [59] programming language was used for the implementation because Java platform gives an advantage to run the implementation on various operating systems. Development and testing phases for the implementation has been done on Microsoft Windows XP, and Windows 2000 operating systems. Java Development Kit version 1.4.2, which was the latest development kit provided by the Sun Microsystems at the start of the project, have been used.

4.1 Software Libraries

Since the motivation for this implementation is to build an application that uses PC graphics hardware for rendering volumetric datasets, an application programming interface (API) that gives an interface to communicate with the graphics processing unit had to be used. The first intention was to use Java3D [58] API which provides a set of object oriented interfaces that support a high level programming model that is able to give interface for the developers to develop an implementation that works on both of the major low level graphics APIs, Microsoft DirectX and OpenGL. The first prototypes built for understanding the computer graphics implementation techniques, has been done using this API. Java3D API gave an easy and faster implementation capability as it provides an object oriented and descriptive interface. However, first prototyping experiences showed that using a procedural interface like OpenGL would give more capabilities while rendering a scene, because every step for rendering can be controlled and modified by the programmer, which cannot be a case for a high level interface like Java3D. Actually, at the start of the project, the programming requirements for building a volume rendering implementation was not very clear. Choosing a lower level API would require more time for understanding and developing the software, but would be more flexible for the changing programming requirements. Because of these, OpenGL API is chosen for the implementation.

"OpenGL is supported on every major operating system, it works with every major windowing system, and it is callable from most programming languages. It offers complete independence from network protocols and topologies. All OpenGL applications produce consistent visual display results on any OpenGL API-compliant hardware, regardless of operating system or windowing system." [60] In order to use OpenGL API in Java platform, a project called Java for OpenGL (JOGL) has been initiated. "The JOGL Project hosts a reference implementation of the Java bindings for OpenGL API, and is designed to provide hardware-supported 3D graphics to applications written in Java. It is part of a suite of open-source technologies initiated by the Game Technology Group at Sun Microsystems. JOGL provides full access to the APIs in the OpenGL 1.5 specification as well as nearly all vendor extensions, and integrates with the AWT and Swing widget sets." [61]

JOGL User Guide [62] explains the properties for this API as follows:

"JOGL is a Java programming language binding for the OpenGL 3D graphics API. It supports integration with the Java platform's AWT and Swing widget sets while providing a minimal API that handles many of the issues associated with building multithreaded OpenGL applications. JOGL provides access to the latest OpenGL routines (OpenGL 1.4 with vendor extensions) as well as platform-independent access to hardware-accelerated off screen rendering. JOGL also provides some of the most popular features introduced by other Java bindings for OpenGL like GL4Java, LWJGL and Magician, including a composable pipeline model which can provide faster debugging for Java-based OpenGL applications than the analogous C program. JOGL was designed for the most recent version of the Java platform and for this reason supports only J2SE 1.4 and later. It also only supports true color (15 bits per pixel and higher) rendering; it does not support color-indexed modes. Several complex and leading-edge OpenGL demonstrations have been successfully ported from C/C++ to JOGL without needing direct access to any of these APIs. However, all of these classes and concepts are accessible at the Java programming language level in implementation packages, and in fact the JOGL binding is itself written almost completely in the Java programming language."

In order to read volumetric datasets produced for medical imaging systems, a library called NeatMed [63] has been used. NeatMed medical imaging API was created by Vision Systems Laboratory [27] with the purpose to facilitate the development of medical imaging applications. It provides an access to medical imaging volumetric datasets that were encoded according to two industry standards DICOM [64] or Analyze [37].

4.2 Properties of the Implementation

This section shall provide information about the capabilities of the implementation. Information about user interface and some methods of the implementation are given in Appendix A, *File Menu, GL Window,* and *View* chapters.

4.2.1 Visualizing Volumetric Datasets

Volumetric datasets are read and the intensity values for the voxels are kept in an array. Header part of the files provides information about the number of voxels present at each axis and the size information about the voxels. In some datasets, the distance that a voxel's height, width and depth represent may differ. This information is used for scaling each axis of the model while rendering, in order to obtain correct proportion of sizes.

First step to visualize the volume is to send the volumetric dataset array to the texture memory as a three dimensional texture. Then map this texture to the polygons and render the scene. Figure 4.1 shows a set of polygons mapped with the volumetric data texture.

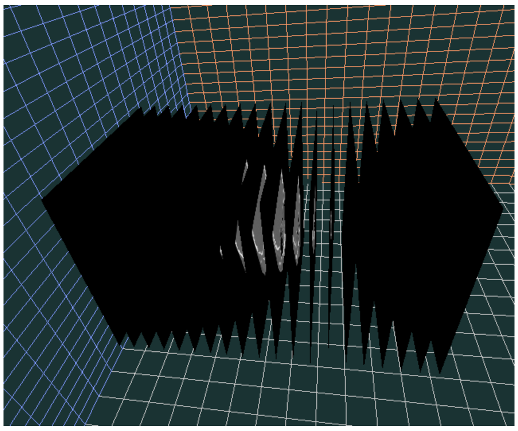


Figure 4.1 Texture mapped polygons

A raw medical imaging dataset does not carry information about translucency of the voxels. As it is seen Figure 4.1, the black regions of the dataset, which represents no intensity value are also seen because no transparency option has been set yet. Compositing methods described in the previous sections can implemented in OpenGL by using blending commands.

The application gives the user the ability to control the number of slices to be rendered during the rendering process. Keyboard control for this property is:

D + *PageUp*: increase the density of the slices
D + *PageDown*: decrease the density of the slices

The key "*F9*" sets the rendering mode into *automatic density* state in which the density of slices is increased a bit in every frame of rendering. In other words, when a key or mouse is pressed, the density of the slices becomes to the smallest number (this number is relative to the dataset) so that the rendering process occurs at the highest frame rate and the response time of the application to the user requests becomes lower. If no key and mouse is pressed, the detail of the rendering increases as the number of slices increase a bit in every rendered frame. Figure 4.2 shows a sequence of captured image for this command.

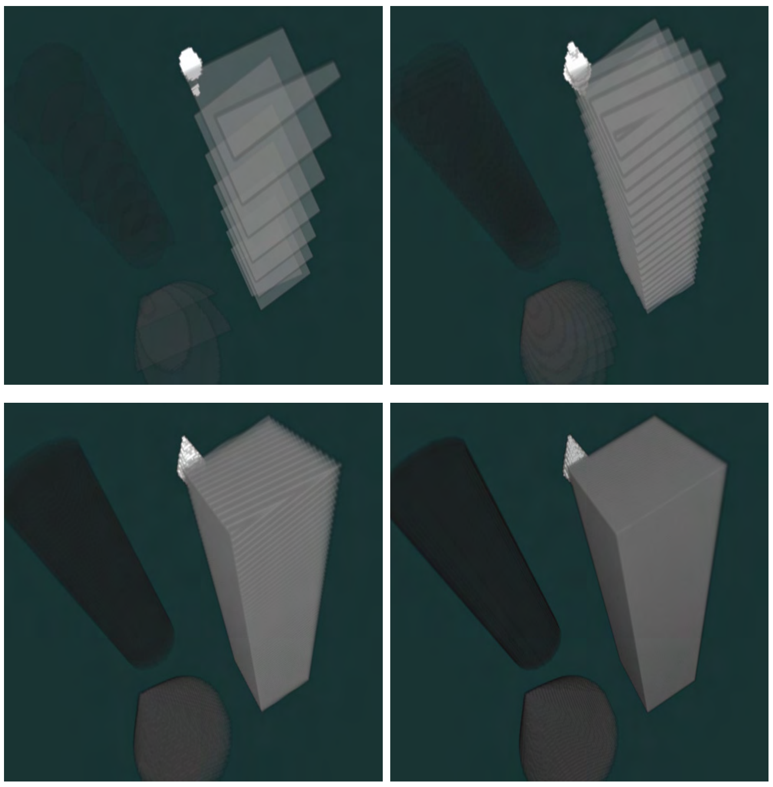


Figure 4.2 Different densities of slices

The application also enables the user to change the distance between slices during visualizing the model with the keyboard control:

P + *PageUp*: increase the distance between the slices
P + *PageDown*: decrease the distance between the slices

When the rendering projection model is set to perspective mode as described in the OpenGL section, the result of changing the distance between slices created a different perspective results as shown in the Figure 4.3.

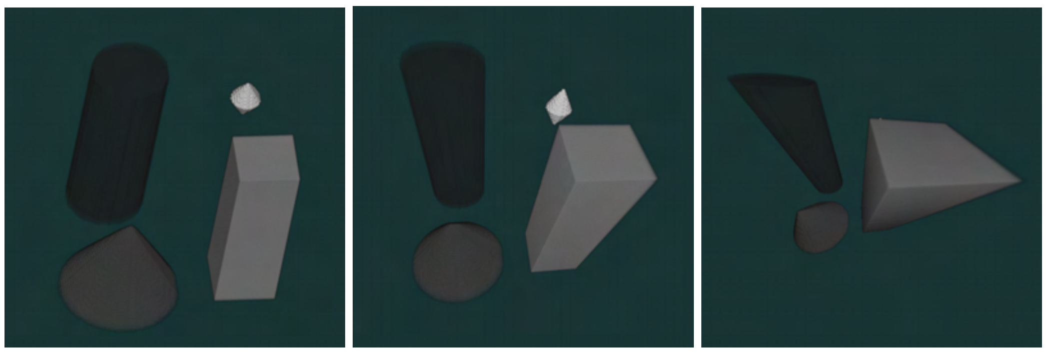
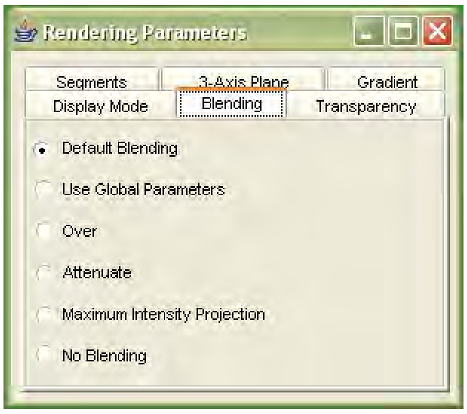


Figure 4.3 Different distances between slices

The first result seen in Figure 4.3 is the result of rendering the volume, with slices that has no distance between them. This caused a result of orthographic projection. When distance between the slices increase, a perspective effect is seen, as shown in the following images. However, as the distance between slices increase, the parts of the model that are far from the camera are rendered as darker as seen in the third image of Figure 4.3. This happens because as the distance increases, the number of slices between the camera and the far points should also increase in order to obtain a smooth result for the nearer objects. As a future work, a solution for this case can be: decreasing the number of slices that appear far from the camera by using a ratio with respect to the distance between camera and the slice. In other words, decreasing the density of slices when the distance from the camera increases, shall create better perspective results.

A perfectly transparent surface shows the object behind of it, because it does not reflect any light from its surface. A translucent material shows the objects behind, but those objects appearing are affected by the translucent material in the front, because some of the light that hits a translucent material is reflected. In volume rendering, viewing the translucency of the voxels are achieved by compositing methods described. OpenGL does not support a direct interface for rendering the partially opaque surfaces. However, compositing techniques can be applied in OpenGL with blending. Appendix B-1 gives detailed information about blending function of OpenGL.



[✓ Default Blending]
Use Global Parameters
Over
Attenuate
Maximum Intensity Projection
No Blending

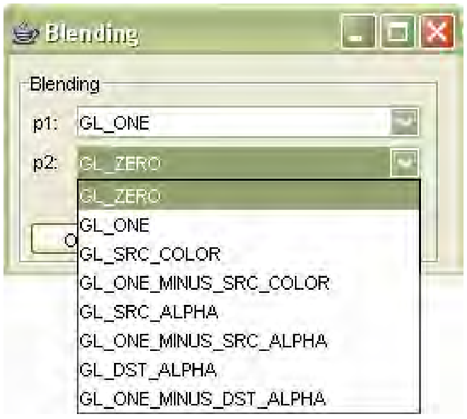
Figure 4.4 Rendering Parameters, Blending window

Figure 4.4 shows the different blending options for volume rendering in the implementation. *No Blending* option disables the blending; Figure 4.1 has been captured with this option. *Default Blending* option has been done, because some of the *display modes* which will be explained later, require their own blending options. Selecting this option disables the blending parameters set by the user.

Blending options are set in OpenGL by the command:

glBlendFunc(*sourceFactor*, *destinationFactor*)

This function is used by supplying destination and source factors. (The properties of these factors are explained in Appendix B-1) *Use Global Parameters* option uses the different methods of blending than specified by the user on the *Blending Window* shown on Figure 4.5.



Blending
p1: GL_ONE
p2: GL_ZERO

GL_ZERO
GL_ONE
GL_SRC_COLOR
GL_ONE_MINUS_SRC_COLOR
GL_SRC_ALPHA
GL_ONE_MINUS_SRC_ALPHA
GL_DST_ALPHA
GL_ONE_MINUS_DST_ALPHA

Figure 4.5 Blending Window

This option has been implemented for testing different kinds of blending techniques. Parameter selected in the combobox *p1,* is used as source factor, and *p2* parameter is used as destination factor. Not all the combinations that user may select in this option are useful for using in volume rendering. *Over, Attenuate* and *Maximum Intensity Projection* options in the *Rendering Parameters* window present predefined methods of bending for volume rendering.

The Over operator [28] is the default blending option used while the volumetric dataset is first loaded into the application. The slices of volumes which are built by mapping 3D textures on to polygon are drawn from back to the front order. The over method of blending, approximates the flow of light passing through translucent materials. The transparency of each voxel is determined by the alpha values assigned. By default, the application uses the intensity values as alpha values of the voxel. The pixel of the textures mapped on the slices with higher alpha values hides the other pixels behind them. Over method is implemented in OpenGL like this:

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)
```

This function works like this:

Source: (A_s, A_s, A_s, A_s)

which is *GL_SRC_ALPHA*

Destination: $(1, 1, 1, 1)-(A_s, A_s, A_s, A_s)$

which is *GL_ONE_MINUS_SRC_ALPHA*

Result is computed for each red, green and blue components as follows:

(color of the source)(alpha of the source)+(color of the destination)(1 – alpha of the source)

Details of OpenGL blending computation method is explained in Appendix B-1. Figure 4.6 shows an example for blending with over operator result.

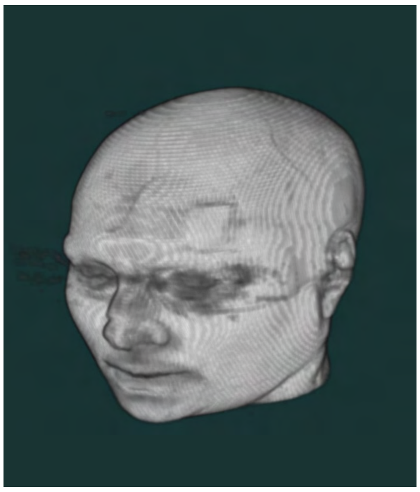
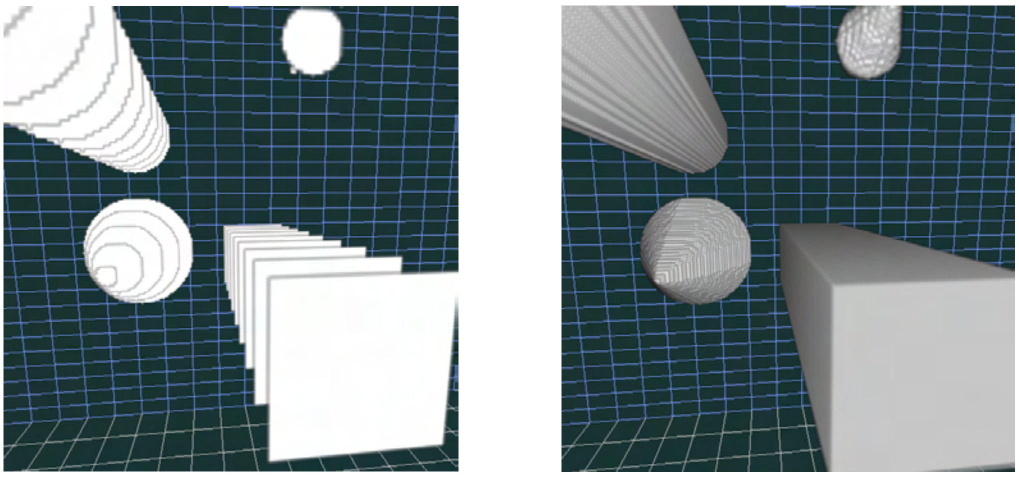


Figure 4.6 Over operator



(a) **(b)**

Figure 4.7 Lighting appearance effect in Blending with Over Operator (a) Rendered with less number of slices with respect to the rendering result (b)

The slices are drawn to the frame buffer with back to front order and blending with over operator the value of each pixel of the rendering result is determined by the translucencies of the voxels that are mapped on the slice polygons. In other words, a voxel color value is multiplied with its alpha (GL_SRC_ALPHA) value then it is multiplied with one minus alpha value of the voxel that is nearer to the user (GL_ONE_MINUS_SRC_ALPHA) and this goes on for each primitive along the ray on the each pixel of result image. However, as seen on the rendering result of the image placed in the right side of Figure 4.7, there is an effect like lighting even though the values and the transparencies are same for each voxel in the dataset. This effect is not a result of the OpenGL global lighting settings. This effect occurred because of the texture mapping settings defined by the command *glTexParameter*.

This parameter enables the pixels of the textures that are mapped to be rendered as nearest or linearly. If nearest setting had been chosen, the value of the texture element that is nearest to the center of the pixel being textured would have been seen in the result. However, this caused a non smooth look for the surfaces of the volume. Linear setting of the texture returns the weighted average of the four texture elements that are closest to the center of the pixel being textured. This setting is used in the implementation. Figure 4.7 (a) shows this effect. That image has been rendered with a very low density of slices and the edges of the squares seen as white on the image are place that the intensity value changes from 0 to 1.0. However, they are not mapped like that because of the linear mode of the texture. This effect has an disadvantage that the color of the surfaces look darker in the result, however it created a very smooth surfaces and also a lighting like effect for the volume. This effect is occurred because, when the angle of the surface with the image plane increases, number of voxels that are calculated as weighted sum (the grey edges of the white squares as seen on the left image) also increases, so the surface looks darker. Because there is also a smooth pass on the translucencies of the voxels on the surface. As seen on the right image, because of the perspective effect, while going ahead from the image plane, the angle of the surface gets larger, so it also gets darker.

The attenuate operator works with the same logic with X-ray. The intensity of a pixel on the render result shows the opacity density of the voxels along the ray of that pixel. The alpha values of the voxels in the volume appear to attenuate light shining through the viewer. In other words, the final brightness at each pixel of the result is the total density of the alpha values of the voxels along the ray. OpenGL command for attenuation is as follows [65]:

```
glBlendFunc(GL_CONSTANT_ALPHA_EXT, GL_ONE)
glBlendColorEXT(1.f, 1.f, 1.f, 1.f / number_of_slices)
```

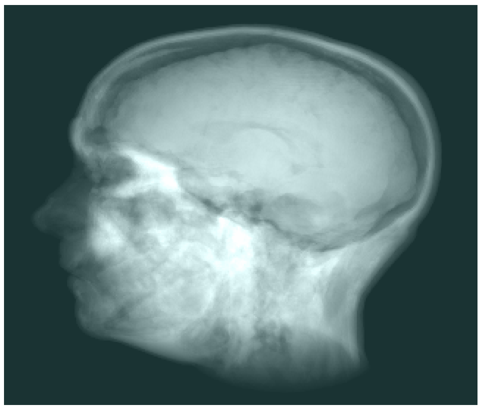


Figure 4.8 Attenuate Operator

The result of rendering shall differ as number of slices used is changed. Application lets the user to change the number of slices used, interactively during the rendering process.

The Maximum Intensity Projection (MIP) is visualizing the brightest voxel value along the light ray for each pixel of the output image. "MIP is a contrast enhancing operator; structures with higher alpha values tend to stand out against the surrounding data. MIP can be implemented with OpenGL using the blend min-max extension" [66].

glBlendEquationEXT(GL_MAX_EXT)

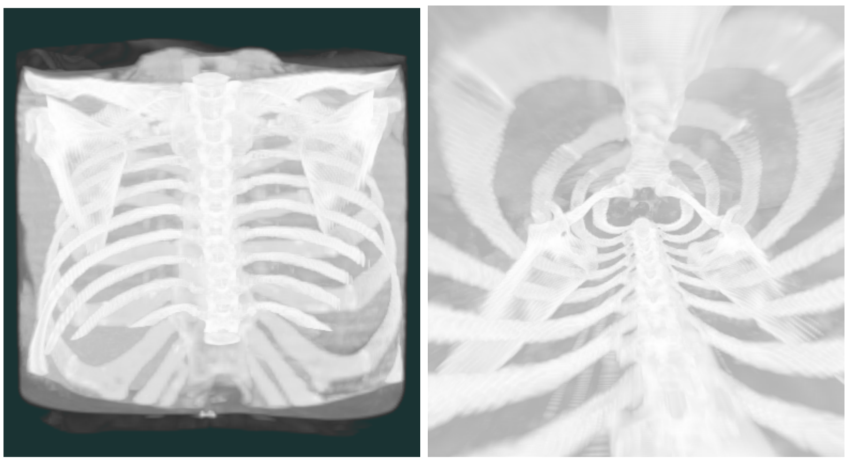
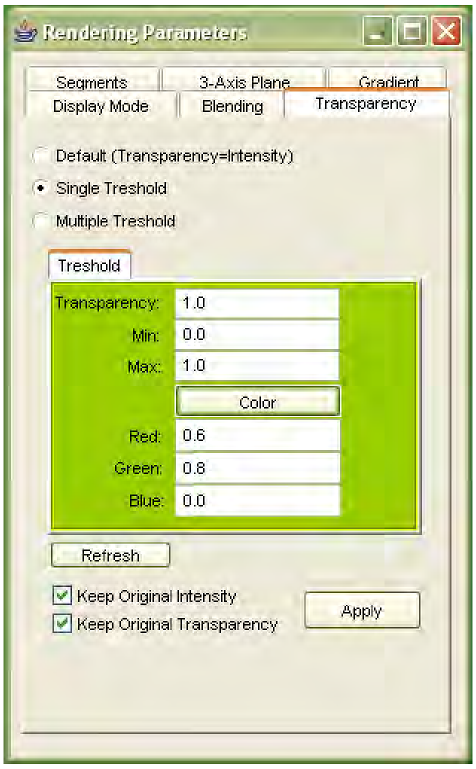


Figure 4.9 Maximum Intensity Projection



Default (Transparency=Intensity) [✓ Single Treshold] Multiple Treshold

Transparency: 1.0
Min: 0.0
Max: 1.0
Red: 0.6
Green: 0.8
Blue: 0.0

[✓] Keep Original Intensity
[✓] Keep Original Transparency

Figure 4.10
Transparency options, single threshold mode

Results for this operator can be seen in the Figure 4.9. The image on the right side is a rendering result of a flight into the same dataset rendered on the left side, with the same rendering options, inside the volume with an exaggerated perspective.

When a volumetric dataset is loaded to the application, as default the translucencies of the voxels are set to same values with the intensity values. The *transparency* part in the *Rendering Options* GUI enables the user to change the transparency parameters. Intensity values of the voxels are scaled between 0 and 1. For example, if a voxel intensity value is *0.7*, the application assigns its color components of red, green, blue and the transparency component alpha as *0.7*. *Transparency* GUI seen in the Figure 4.10, enables the user to select minimum and maximum intensity values to be rendered. The specified values of minimum and maximum thresholds are used for filtering the dataset while visualizing. Figure 4.11 shows a result for threshold filtering.

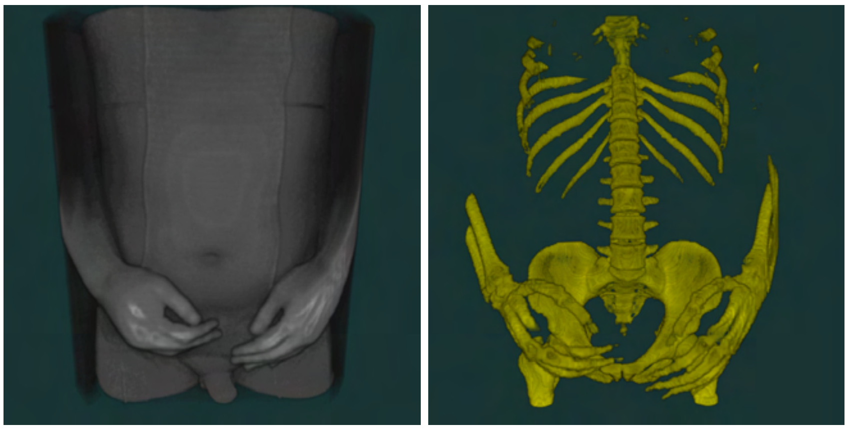


Figure 4.11 Threshold filtering of a dataset.

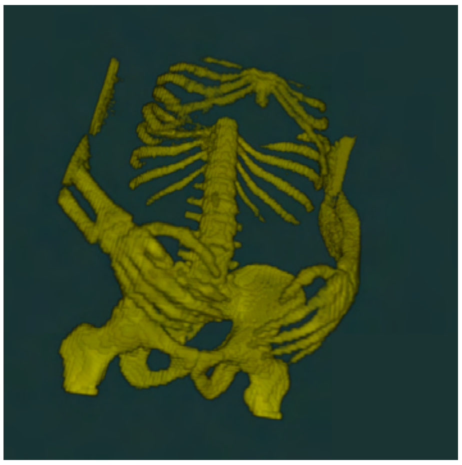
The image left on Figure 4.11 is the result of rendering a CT dataset without any filtering. The image on the left is a result of rendering the same dataset with filtering the intensity values.

The *transparency* GUI enables the user to assign color values for the voxels. If *keep original intensity value* checkbox is checked, each color component red, green and blue is multiplied with the assigned color components. If *keep original intensity value* checkbox is not checked, the assigned color is directly used as the color of the voxels. This creates a brighter view as all the voxels are rendered with same color, however the render results show that the surfaces look less smooth with an artificial look as seen in Figure 4.12.

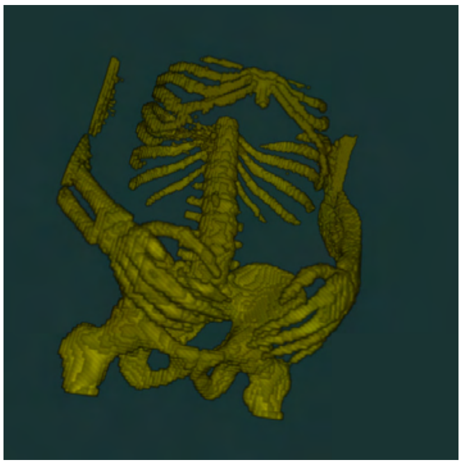
The checkbox *keep original transparency,* in the *transparency* GIU enables user to manipulate the translucency values of the voxels. If *keep original transparency* is checked, the transparency value that is specified by the user is multiplied with the alpha values of the voxels. If it is not checked, the alpha values are changed to the value that user specified. The results for this method of rendering showed that, the smoothness of the surfaces is gone since each voxel that passes the filter is rendered with full opaque (if the transparency value is assigned as the maximum 1.0).

If both *keep original transparency* and *keep original intensity value* checkboxes are unchecked, a bright rendering result is established; because, all of the voxels are assigned with same color and their alpha values are kept also the same. The rendering results showed that the smoothness of the surfaces are gone and an artificial result is obtained since all of the voxels are seen with the same color and transparency. All combinations of the rendering results for this method can be seen in Figure 4.12.

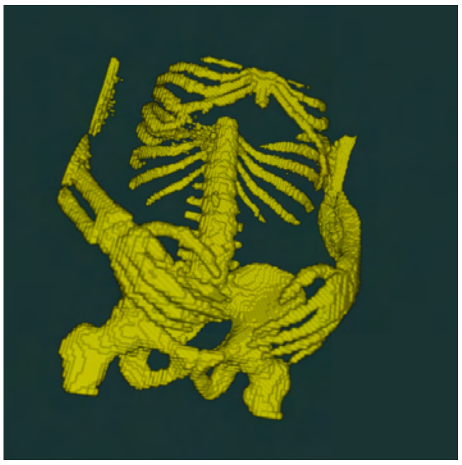
A GUI for selecting automatic threshold values or by the interaction of user from an histogram table is a future work to be done for this section.



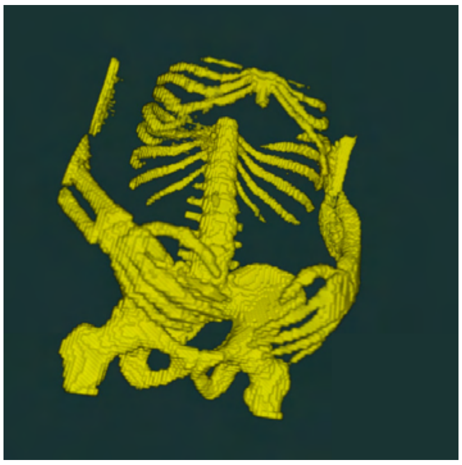
(a)



(b)



(c)



(d)

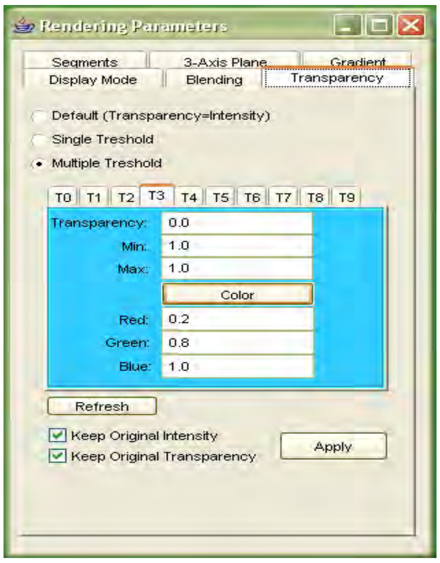
Figure 4.12 Results for different combinations of intensity and transparency

(a) Both *keep original transparency* and *keep original transparency* are selected (default option);

(b) *keep original transparency* is not selected, and *keep original transparency* is selected;

(c) *keep original transparency* is selected and *keep original transparency* is not selected;

(d) Image on the down right side: both *keep original transparency* and *keep original transparency* are not selected.


Rendering Parameters

Segments | 3-Axis Plane | Gradient

Display Mode | Blending | Transparency

Default (Transparency=Intensity)
Single Treshold
[✓ Multiple Treshold]

T0 | T1 | T2 | [✓ T3] | T4 | T5 | T6 | T7 | T8 | T9

Transparency:	0.0
Min:	1.0
Max:	1.0
	Color
Red:	0.2
Green:	0.8
Blue:	1.0

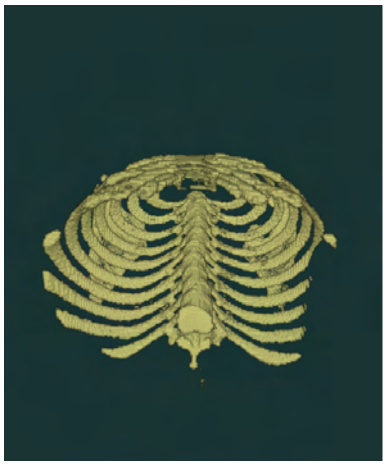
Refresh

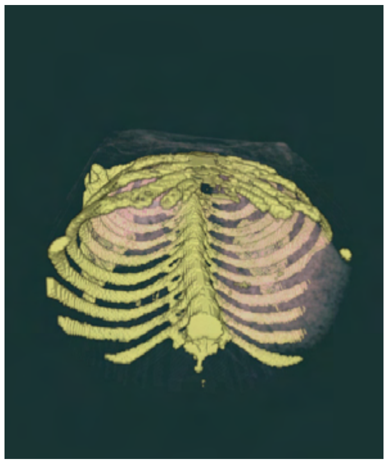
☑ Keep Original Intensity
☑ Keep Original Transparency

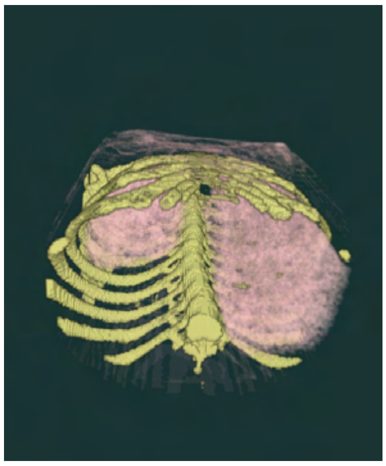
Apply

Figure 4.13 Transparency options, multiple threshold mode

Figure 4.13 shows the GUI for assigning more than one threshold filters for rendering the volume. This part works with the same logic of the single threshold section, but here the user can specify one than more intervals of thresholds to be rendered in different transparencies and colors. Figure 4.14 shows some results for this rendering option.


(a)


(b)


(c)

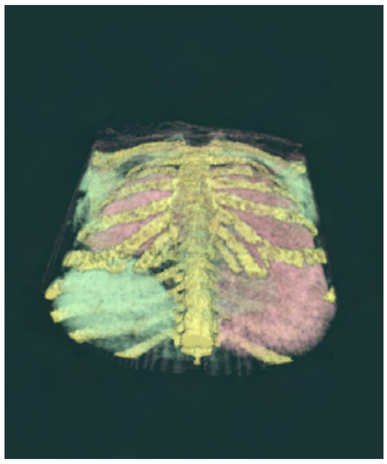

(d)

Figure 4.14 Rendering with multi transparency options.

Figure 4.14 shows a dataset with different transparency options. The model is rotated about x axis in (d).

4.2.1.1 Gradient

Gradient vector can be used for computing the reflection of the light rays meeting the surfaces of the geometric primitives. The rendering results shown so far do not use *Phong* shading, but they use the *Gouraud* shading. OpenGL does not directly support Phong shading [29]. The volumetric data has been stored as 3D texture and it is mapped on slices of polygons in order to be visualized. The global lighting options compute the lighting of the polygons according to the normal vectors assigned to the vertices of the polygons, in this case, the vertices of the slices. However, there is a requirement to compute the lighting according to each voxel in volume rendering. In this case, each normal vector of the pixels of the textures on the slice polygons are required to be calculated and used for the computation of lighting.

First obstacle is how to compute the gradient vectors. There are many methods that can be implemented for solving this issue. Methods used in the implementation can be seen in Figure 4.15.

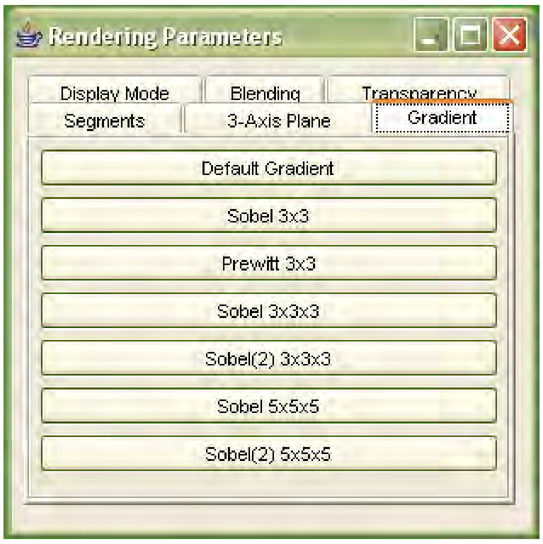


Figure 4.15 Gradient computation methods

In the *Default Gradient* mode, which is calculated while first loading the dataset, *central difference* method is used.

In the other methods, convolution method is used for calculating the gradient vector. The kernels used for these computations are given as follows:

`Sobel 3x3:`

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

`Prewitt 3x3:`

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

`Sobel 3x3x3:`

$$\begin{bmatrix} -2 & 0 & 2 \\ -3 & 0 & 3 \\ -2 & 0 & 2 \end{bmatrix}$$

$$\begin{bmatrix} -3 & 0 & 3 \\ -6 & 0 & 6 \\ -3 & 0 & 3 \end{bmatrix}$$

$$\begin{bmatrix} -2 & 0 & 2 \\ -3 & 0 & 3 \\ -2 & 0 & 2 \end{bmatrix}$$

`Sobel 3x3x3 (2):`

$$\begin{bmatrix} -1 & 0 & 1 \\ -3 & 0 & 3 \\ -1 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} -3 & 0 & 3 \\ -6 & 0 & 6 \\ -3 & 0 & 3 \end{bmatrix}$$

$$\begin{bmatrix} -1 & 0 & 1 \\ -3 & 0 & 3 \\ -1 & 0 & 1 \end{bmatrix}$$

`Sobel 5x5x5:`

$$\begin{bmatrix}-4 & -3 & 0 & 3 & 4\\ -5 & -4 & 0 & 4 & 5\\ -6 & -5 & 0 & 5 & 6\\ -5 & -4 & 0 & 4 & 5\\ -4 & -3 & 0 & 3 & 4\end{bmatrix}$$

$$\begin{bmatrix}-3 & -2 & 0 & 2 & 3\\ -4 & -3 & 0 & 3 & 4\\ -5 & -4 & 0 & 4 & 5\\ -4 & -3 & 0 & 3 & 4\\ -3 & -2 & 0 & 2 & 3\end{bmatrix}$$

$$\begin{bmatrix}-2 & -1 & 0 & 1 & 2\\ -3 & -2 & 0 & 2 & 3\\ -4 & -3 & 0 & 3 & 4\\ -3 & -2 & 0 & 2 & 3\\ -2 & -1 & 0 & 1 & 2\end{bmatrix}$$

$$\begin{bmatrix}-3 & -2 & 0 & 2 & 3\\ -4 & -3 & 0 & 3 & 4\\ -5 & -4 & 0 & 4 & 5\\ -4 & -3 & 0 & 3 & 4\\ -3 & -2 & 0 & 2 & 3\end{bmatrix}$$

$$\begin{bmatrix}-4 & -3 & 0 & 3 & 4\\ -5 & -4 & 0 & 4 & 5\\ -6 & -5 & 0 & 5 & 6\\ -5 & -4 & 0 & 4 & 5\\ -4 & -3 & 0 & 3 & 4\end{bmatrix}$$

`Sobel 5x5x5 (2):`

$$\begin{bmatrix}-1 & -2 & 0 & 2 & 1\\ -2 & -3 & 0 & 3 & 2\\ -3 & -4 & 0 & 4 & 3\\ -2 & -3 & 0 & 3 & 2\\ -1 & -2 & 0 & 2 & 1\end{bmatrix}$$

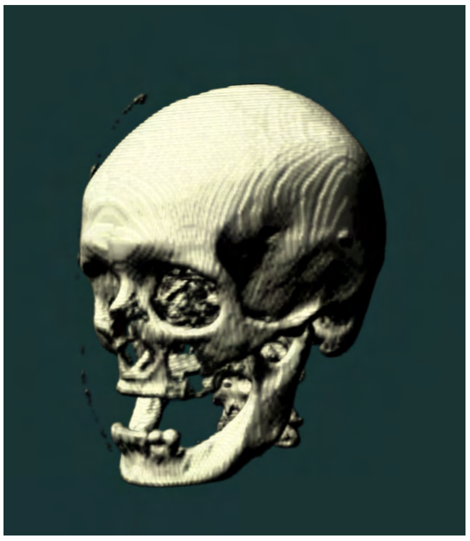
$$\begin{bmatrix}-2 & -3 & 0 & 3 & 2\\ -3 & -4 & 0 & 4 & 3\\ -4 & -5 & 0 & 5 & 4\\ -3 & -4 & 0 & 4 & 3\\ -2 & -3 & 0 & 3 & 2\end{bmatrix}$$

$$\begin{bmatrix}-3 & -4 & 0 & 4 & 3\\ -4 & -5 & 0 & 5 & 4\\ -5 & -6 & 0 & 6 & 5\\ -4 & -5 & 0 & 5 & 4\\ -3 & -4 & 0 & 4 & 3\end{bmatrix}$$

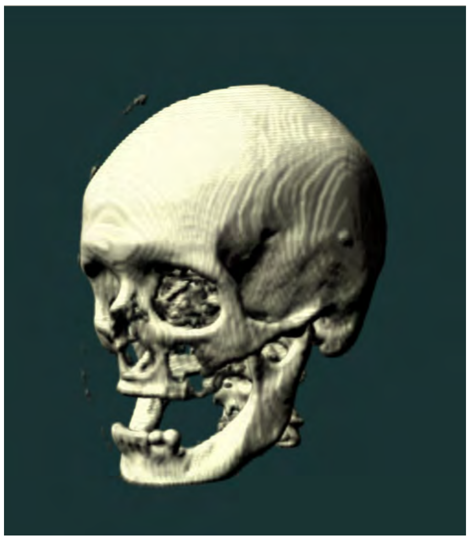
$$\begin{bmatrix}-2 & -3 & 0 & 3 & 2\\ -3 & -4 & 0 & 4 & 3\\ -4 & -5 & 0 & 5 & 4\\ -3 & -4 & 0 & 4 & 3\\ -2 & -3 & 0 & 3 & 2\end{bmatrix}$$

$$\begin{bmatrix}-1 & -2 & 0 & 2 & 1\\ -2 & -3 & 0 & 3 & 2\\ -3 & -4 & 0 & 4 & 3\\ -2 & -3 & 0 & 3 & 2\\ -1 & -2 & 0 & 2 & 1\end{bmatrix}$$

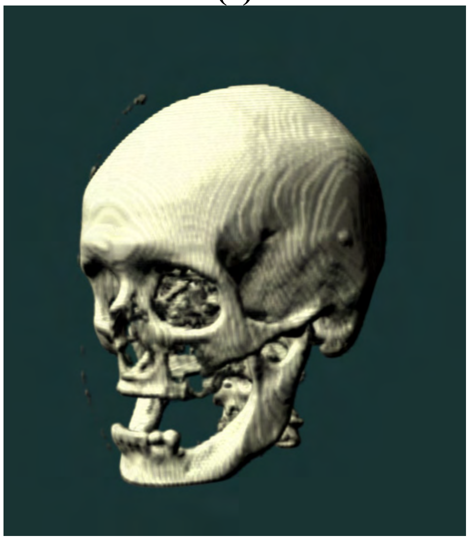
Figure 4.16 give rendering results of different operators on skull head data.



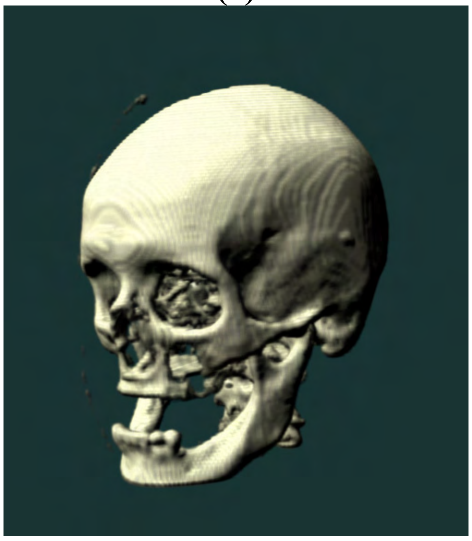
(a)



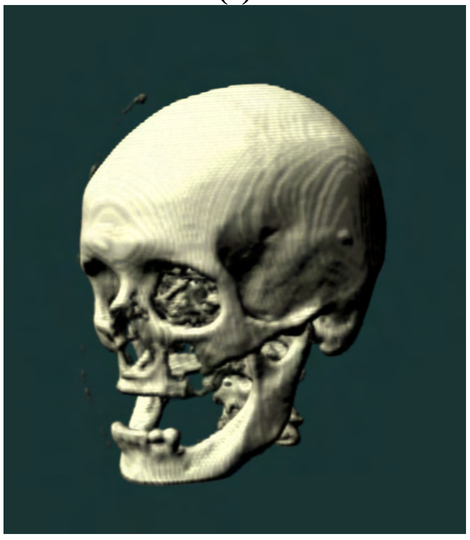
(b)



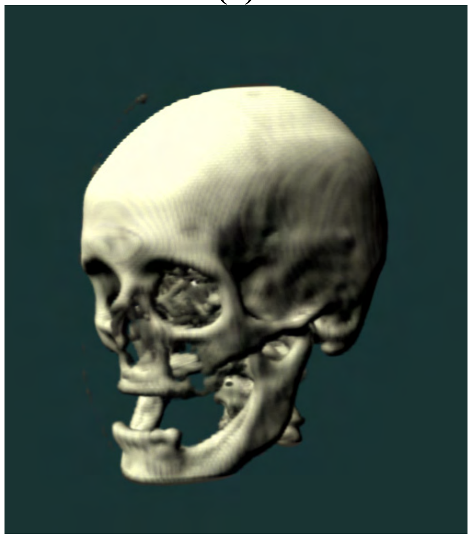
(c)



(d)

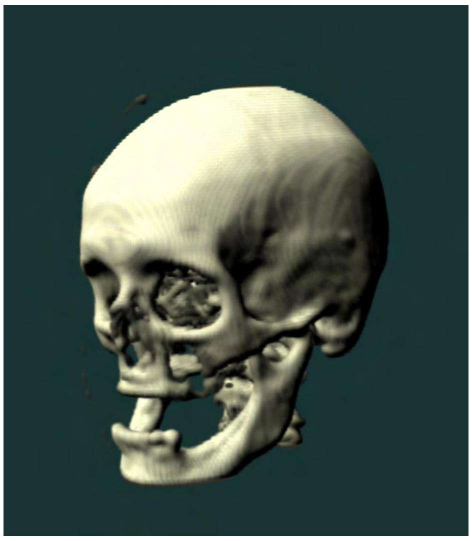


(e)



(f)

Figure 4.16 Rendering with different gradient computation methods



(g)

Figure 4.16 Rendering with different gradient computation methods (a) Central Difference; (b) Sobel 3x3; (c) Prewitt 3x3; (d) Sobel 3x3x3; (e) Sobel 3x3x3 (2); (f) Sobel 5x5x5; (g) Sobel 5x5x5 (2).

When the size of kernel used in computing gradient increases, the surface of model looks smoother as seen in the Figure 4.16.

The gradient vector for each voxel is calculated with one of these methods; however, the problem has not been solved yet. OpenGL does not support a direct illumination model that is applied on each of the pixels of the textures mapped on the slices. In order to enable lighting in volume rendering, a common method which is used for giving more realism to the two dimensional surfaces without using additional polygons, called *bump mapping* [30] has been used. Details about bump mapping can be seen in the Appendix B-1, C-1.

In order to establish a per voxel lighting effect in OpenGL, *Dot3 bump mapping* [69] technique has been used by using the *Dot3* and *multi texturing* extensions of OpenGL [68]. “The difference between "real" bump mapping and dot3 bump mapping is instead of penetrating the surface normal and binormal at each rendered pixel of a surface in dot3 bump mapping a normal map is used.”[67] Normal map is a texture and in our case its a three dimensional texture consist of voxels. The normal map of the volumetric dataset is computed by the gradient operations, and the values of each axis of the normal vectors are stored as red, green and blue color components as seen in Figure 4.17.

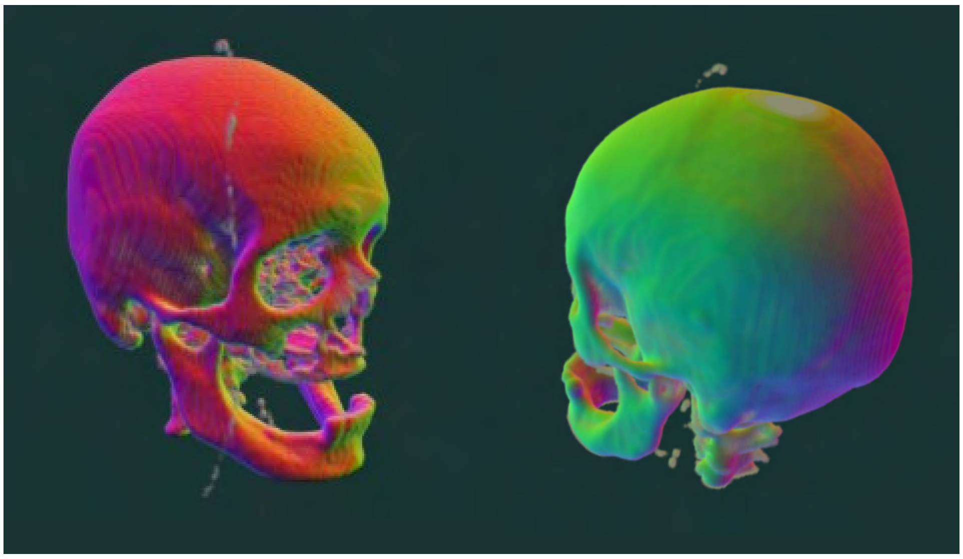


Figure 4.17 Normal Map of the Volumetric Dataset

Now, we remind the equation in diffuse lighting chapter:

$$C_o = C_a k_a O_d + C_p k_d O_d (N \cdot L)$$

The result color of a point is dependent on the dot product (cosine of the angles between them) of the *light* and *surface normal* vectors. The *Dot3* extension of OpenGL enables the dot product of the matrixes done by the GPU. The voxel color values red, green and blue are taken as surface normal and multiplied with the vector of the components red, green and blue of the polygon color. The result is then multiplied with the original volume voxels color values using the OpenGL *multitexture* extension, and the result is a diffuse lighting per pixel, which can be said as *Phong shading.* In order to change the light direction, the light vector should be changed. This can be applied by changing the color of the polygons. The following equation is established by this method:

$$C_o = C_p k_d O_d (N \cdot L)$$

The user can change the color of each red, green and blue components; in other words, change the direction of the light by keyboard controls:

Ctrl + X + PageUp: increase red intensity of the slices, or move light along x axis

Ctrl + Y + PageUp: increase green intensity of the slices, or move light along y axis

Ctrl + Z + PageUp: increase blue intensity of the slices, or move light along z axis

The source code of this operation is given in Appendix C-1. Some rendering results for per-voxel lighting with this method can be seen in Figure 4.18.

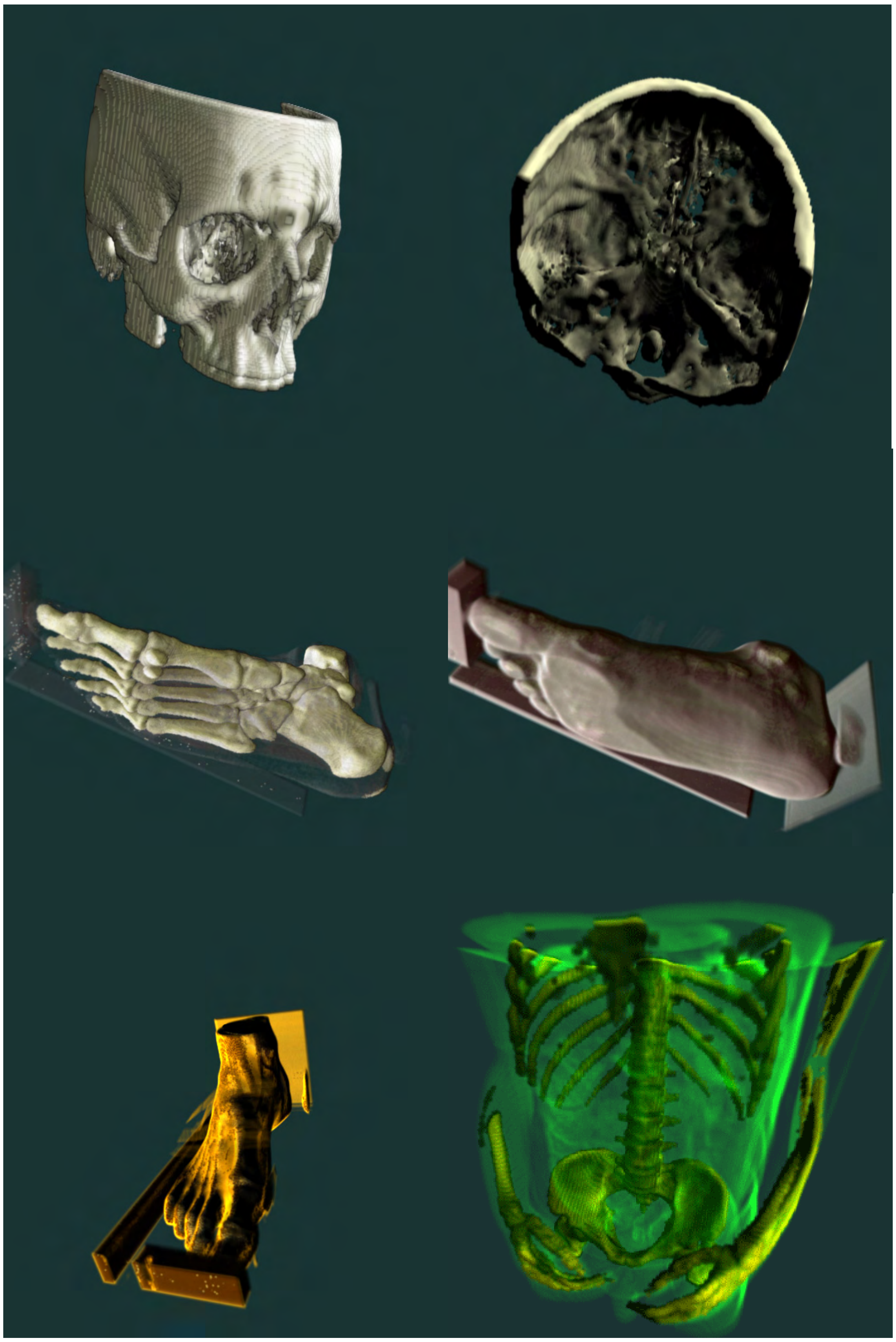
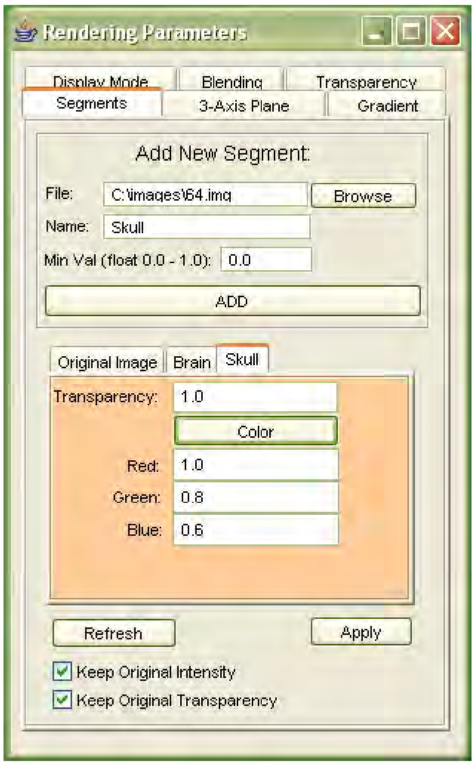


Figure 4.18 Examples for rendering results with per-voxel lighting

4.2.1.2 Segments

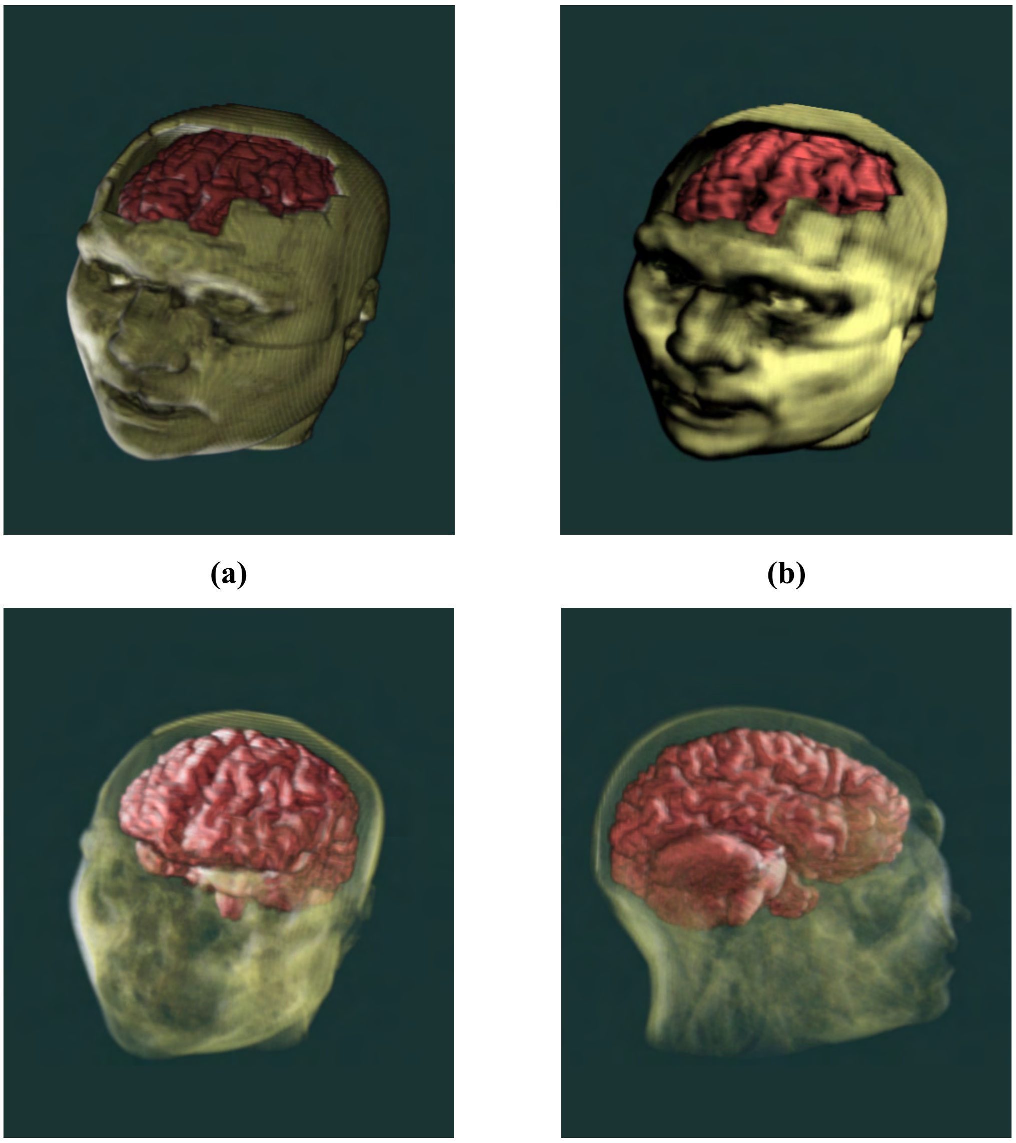
As mentioned in the volume rendering pipeline, segmentation is an important stage especially for medical imaging visualization. There are various methods built for segmenting the volumetric datasets; this application built for the purpose of visualizing the volumetric datasets and does not segment datasets. *Segments* section of the implementation aims to simulate the result of visualizing a segmented dataset. Stored segmented dataset are loaded as new layers of data and each segment's visualization options can be manipulated separately by the user as shown in Figure 4.19.



File: C:\images\64.img
Name: Skull
Min Val (float 0.0 - 1.0): 0.0
Transparency: 1.0
Red: 1.0
Green: 0.8
Blue: 0.6
☑ Keep Original Intensity
☑ Keep Original Transparency

Figure 4.19 Segments GUI

Each loaded segments can be applied the same options that described in the *Threshold* section. Some rendering results for segmented data visualization are shown below.



(c) (d)

Figure 4.20 Segmented data visualization

(a) Bump mapped added mode; (b) Bump mapped; (c),(d) Bump mapped added mode with different transparency adjustments

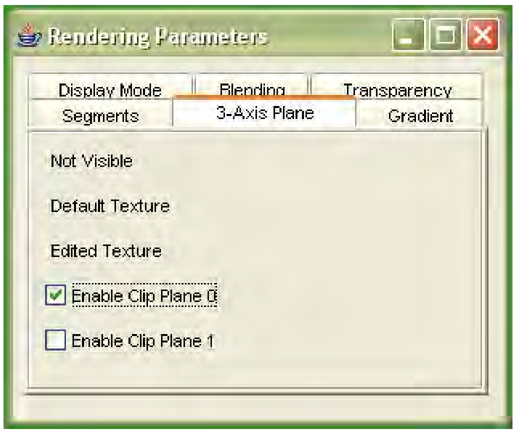
Except the image on the right top side, these rendering results are obtained by the *display mode* which uses the given equation in the diffuse lighting section:

$$C_o = C_a\, k_a\, O_d + C_p\, k_d\, O_d\, (N \cdot L)$$

OpenGL source code for this formula is given in the Appendix C-2.

4.2.1.3 Clip Plane and 3-Axis Plane

This section of the application uses some of the OpenGL methods to enhance the visualization capabilities.

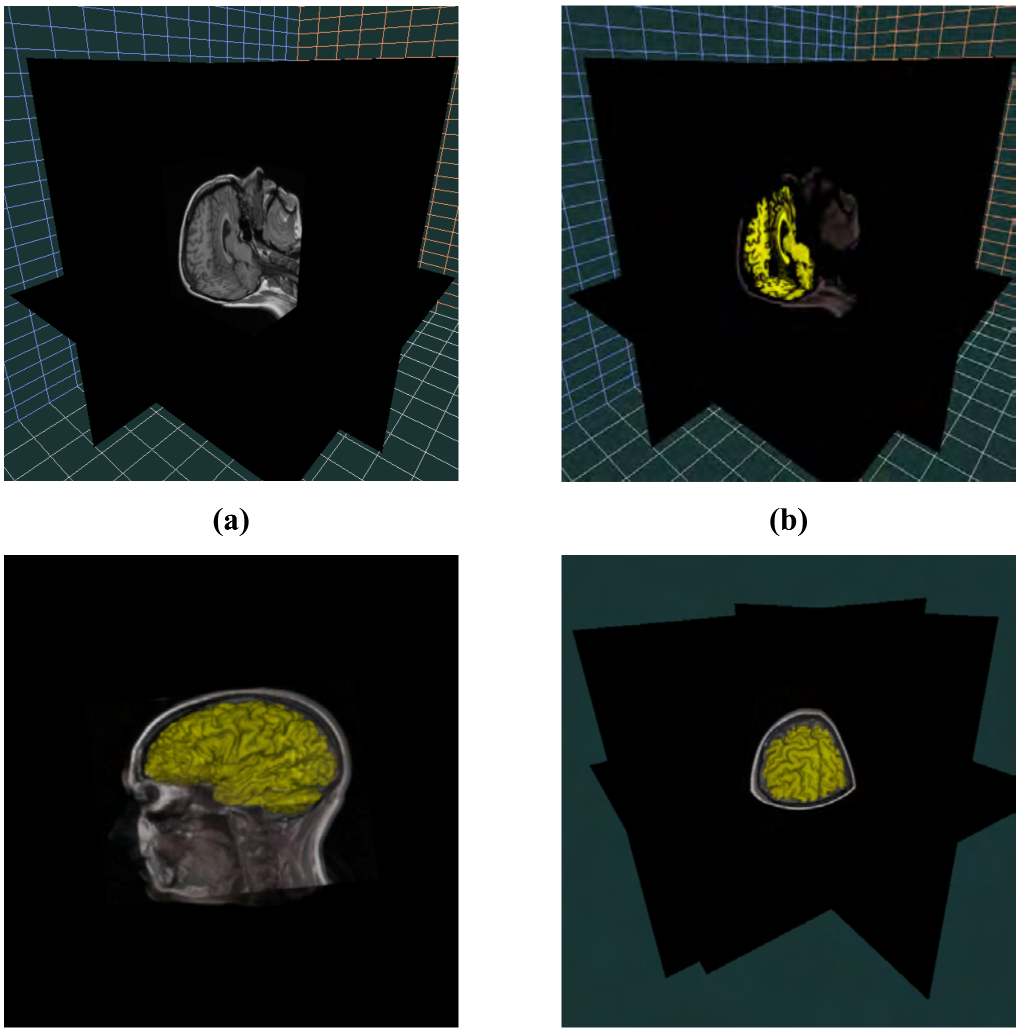


☑ Enable Clip Plane 0

☐ Enable Clip Plane 1

Figure 4.21 Clip Plane, 3-Axis Plane GIU

Trilinear interpolation capability of *3D Texture Mapping* enables the GPU to calculate the all the texels to be mapped on a polygon. Using this ability 3-axis plane command has been implemented. The user is able to move the polygons that are not applied the blend function, and see the two dimensional slices on every location. *Default Texture* command lets the user to visualize the first loaded original volume; this might be useful when some parts of the data are set as transparent as shown on Figure 4.22.



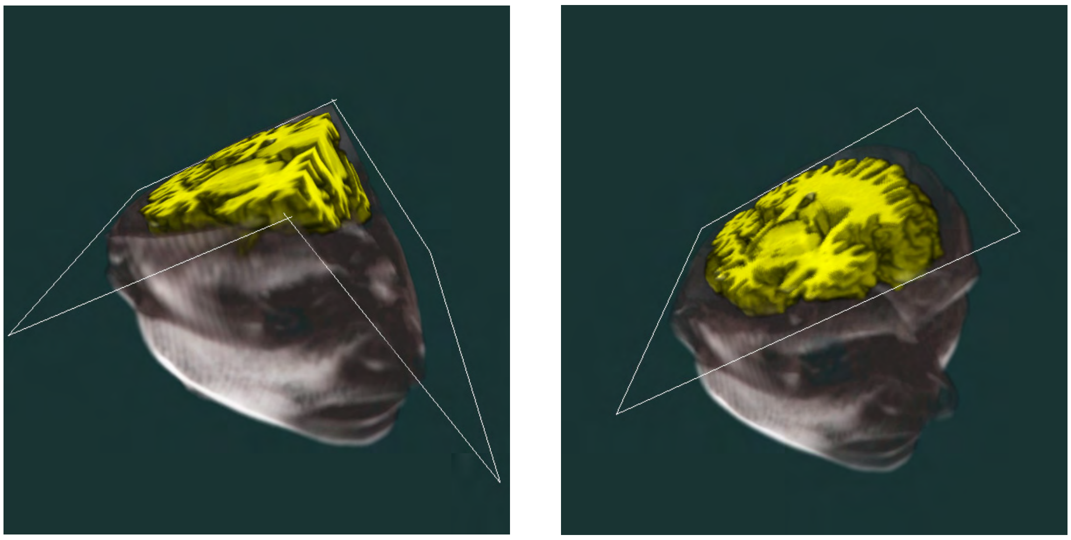
(a) (b)

(c) (d)

Figure 4.22 3-Axis plane

(a) Default texture; (b) Edited texture; (c), (d) Default texture with edited volume.

OpenGL command *glClipPlane* can be used as another interactive visualization method in volume rendering. Figure 4.23 shows some rendering results for this method.

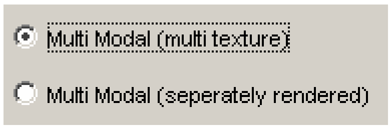


(a) (b)

Figure 4.23 Clip Plane
(a) Clipped with 2 polygons; (b) Clipped with 1 polygon.

4.2.1.4 Multi Modality

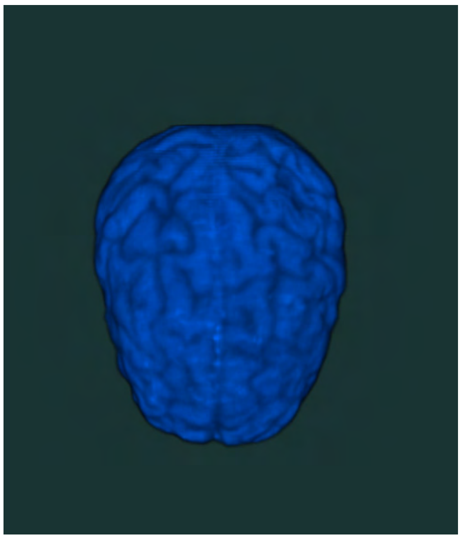
Multi modality visualization section of the software displays two different volumetric datasets in a single scene. The datasets from different modalities are required to be registered before the visualizing process. The implementation uses two different display modes for visualizing multi modality datasets as seen in Figure 4.24.

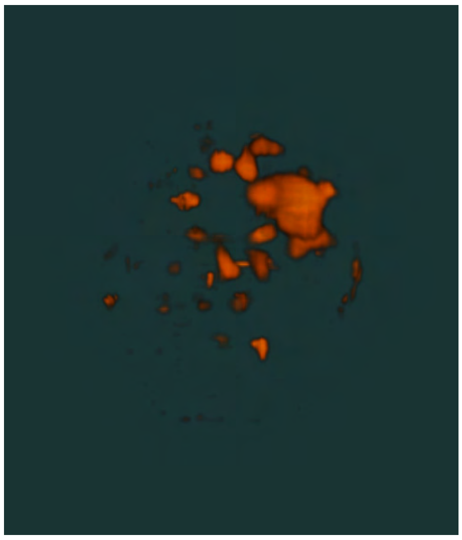


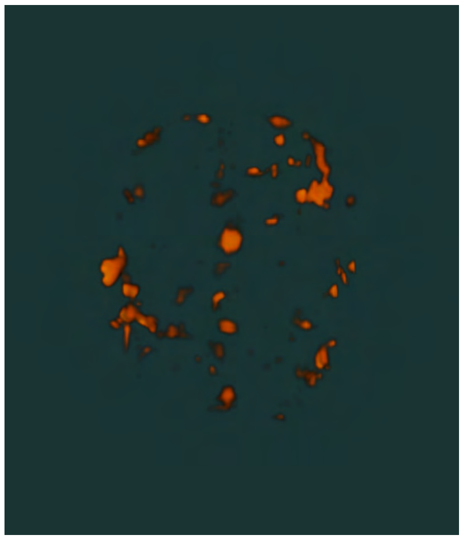
[✓ Multi Modal (multi texture)]
Multi Modal (seperately rendered)

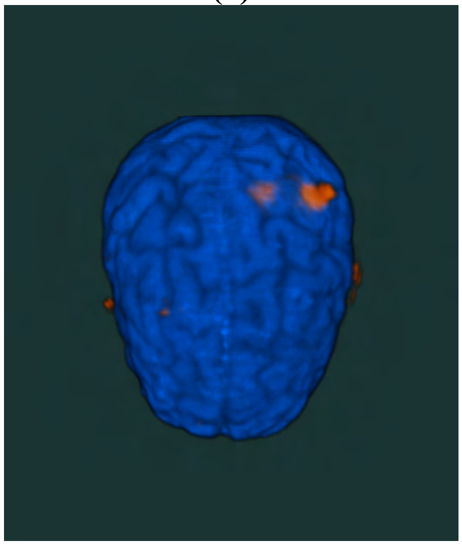
Figure 4.24 Multi Modality display modes

Multi texture option uses the multi-texturing capability of OpenGL. This mode has been implemented with the *add* state of multi texturing which adds each of the color components of the each texels that overlay. This option shall be named as *added multi texture* mode. In the *separately rendered* option, each datasets are rendered separately. Figure 4.25 shows the rendering results for displaying MR and SPECT (processed) datasets of an epilepsy patient.


(a)


(b)


(c)


(d)

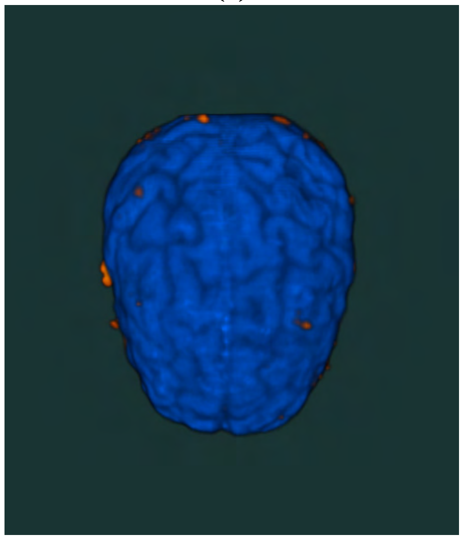
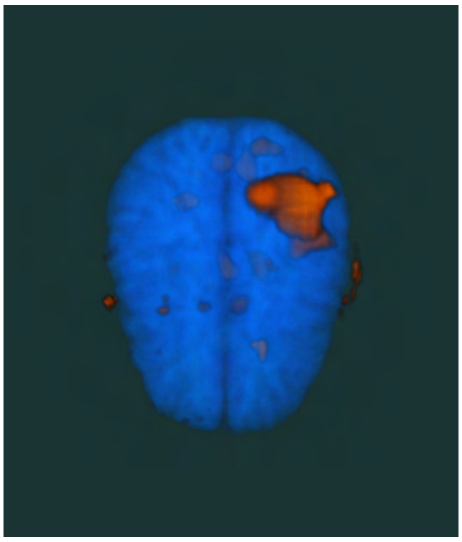
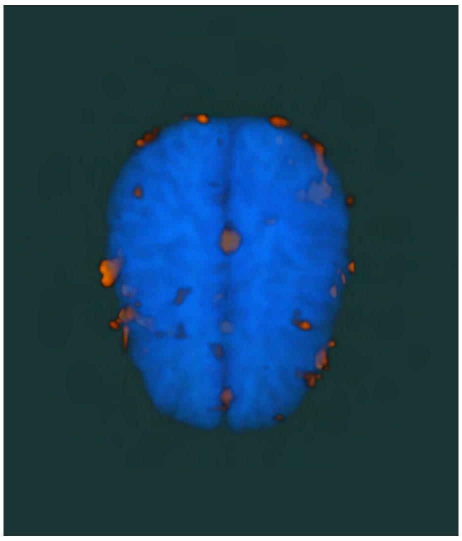
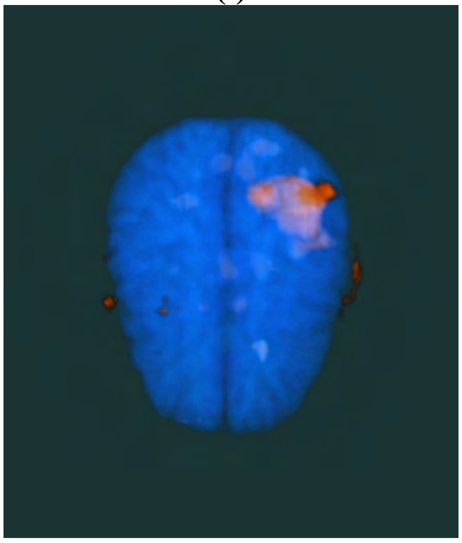
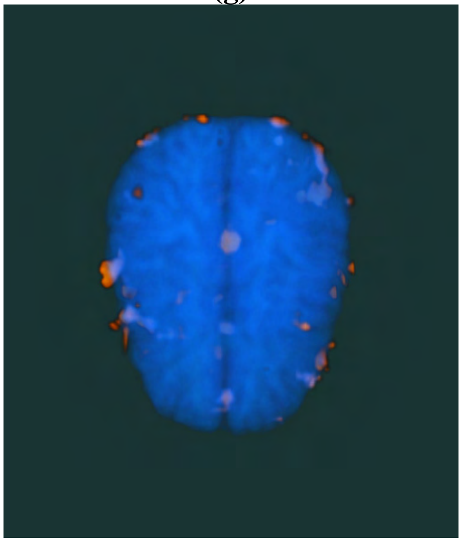

(e)

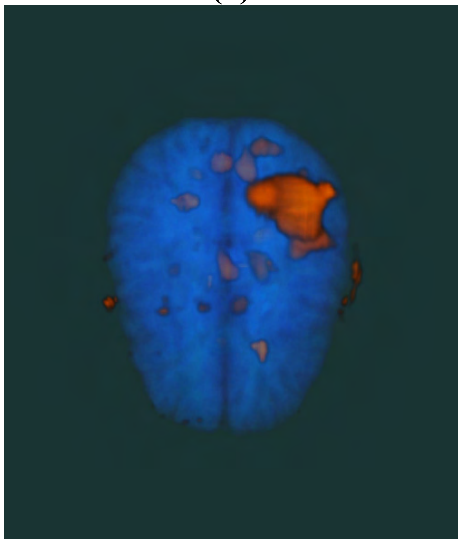
Figure 4.25 Multi modality rendering results of an epilepsy patient.


(f)


(g)


(h)


(i)


(j)

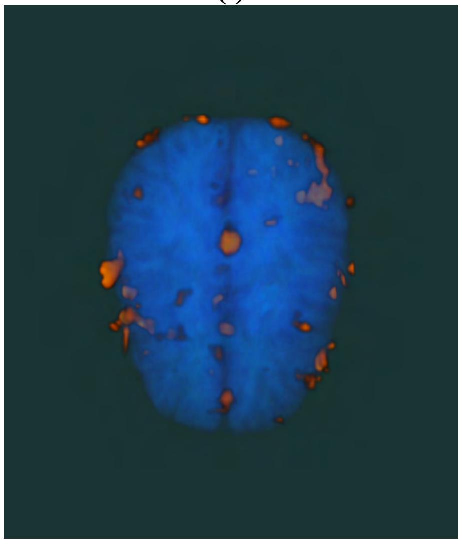
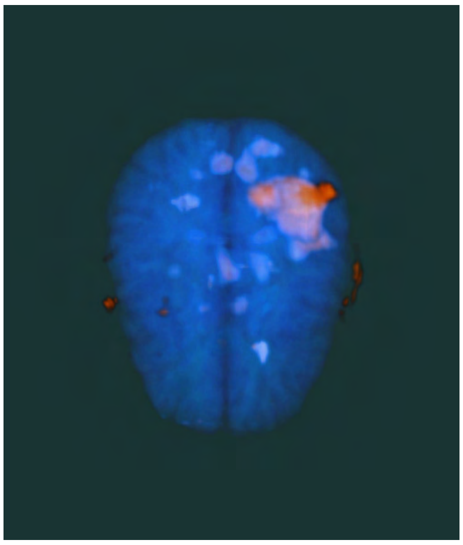

(k)

Figure 4.25 Multi modality rendering results of an epilepsy patient.


(l)

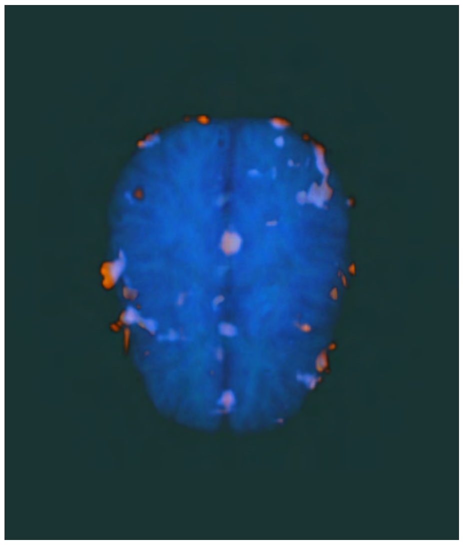

(m)

Figure 4.25 Multi modality rendering results of an epilepsy patient.

(a) An MR dataset

(b) SPECT1: Subtracted (ictal[1]- interictal[2]) SPECT of patient 1

(c) SPECT2: Subtracted (ictal- interictal) SPECT of patient 2

(d) MR full opaque, SPECT1 full opaque

(e) MR full opaque, SPECT2 full opaque

(f) MR %30 opaque, SPECT1 full opaque, with *separately rendered* mode

(g) MR %30 opaque, SPECT2 full opaque, with *separately rendered* mode

(h) MR %30 opaque, SPECT1 full opaque, with *added multi texture* mode

(i) MR %30 opaque, SPECT2 full opaque, with *added multi texture* mode

(j) MR %10 opaque, SPECT1 full opaque, with *separately rendered* mode

(k) MR %10 opaque, SPECT2 full opaque, with *separately rendered* mode

(l) MR %10 opaque, SPECT1 full opaque, with *added multi texture* mode

(m)MR %10 opaque, SPECT2 full opaque, with *added multi texture* mode

(n)

In Figure 4.25 (d) and (e), both of the rendering modes *separately rendered* and *added multi texture* produce same rendering results, because both MR and SPECT datasets are set to full opaque.

[1] *Ictal:* SPECT acquired during an epileptic seizure.
[2] *Interictal:* SPECT acquired in between seizures.

Rendering frame rates and the texture memory requirement for both *separately rendered* and *added multi texture* modes are same. *Separately rendered* mode renders every voxel that presents in each of the modalities (MR and SPECT) with their original color. In *added multi texture* mode, resulting value of a voxel seen in the rendered image is calculated by adding each color components of the voxels from each modalities (MR and SPECT). This method enables the user to detect the intersecting non-transparent parts of the datasets, since the resulting color changes on these voxels.

CHAPTER 5

ANALYSIS OF THE IMPLEMENTATION

This part provides some qualitative comparisons of the outputs and performance test results for the implementation.

5.1 Qualitative Comparisons

The outputs for the implementation have been compared with those of the medical imaging software *Analyze 5.0* [37] which were developed by Mayo Clinic [70] and has been accepted worldwide for medical image analysis. The aim of this section is to make qualitative comparisons. (Information about Analyze software is given in Appendix D).

The version 5.0 of the software Analyze has various kinds of properties-abilities developed for medical imaging. Analyze, *Volume Render* module has been used for capturing the results. There are many methods that have been implemented for each process of the volume rendering pipeline in Analyze software. Difference of this application from our implementation is that, no hardware acceleration method is used. The Analyze software output images used for comparison are the results which are produced by the most similar methods of volume rendering to our implementation methods.

The methods that are compared are as follows:

Depth Shading [31]:

The value of each pixel of the output is a function of depth. The distance of the first renderable voxel along the ray casted from the image plane is rendered with its depth value. So the voxels that appear more far from the image plane are rendered as darker and the nearer are rendered as brighter. Analyze 5.0 has an option for this method that computes the results according to the gradient of the surface. This method has not been implemented in our implementation; however the results created with the blending over operator of OpenGL presented similar results to the Analyze depth shading with gradient estimator. So, even though the methods that are being used are not very similar, the results shall be compared.

Gradient Shading [32]:

Gradient vector of each voxel is computed and the dot product of the gradient vector with the light vector which is independently specified. This has the same logic with the method that is used in *Bump Mapping with Dot3 extension of OpenGL* in our implementation. Analyze 5.0 support specular reflection computations for this method of rendering. Our implementation only supports diffuse reflection, and specular reflection shall be a future work to be implemented.

Volume Compositing:

This method uses the same technique with *Gradient Shading*, and enables the user to specify different alpha and color values for the volumetric data according to intensity values of each voxels. This ability is achieved by the *Threshold* section. Analyze interface enables the user to specify the values with a different technique we have used. So, the results of similar requests of user shall be compared.

Maximum Intensity Projection:

This method is implemented and has been specified in the *2.4.6.3 Maximum Intensity Projection Chapter*. It is achieved by maximum intensity projection blending technique with our implementation.

Summed Voxel Projection:

This section computes the average intensity values of the voxels which are present along the casted rays from the image plane to the volume. *Attenuate* method mentioned in the blending section achieves this option in our application.

5.1.1 Rendering Results

The main drawback we have faced while using hardware acceleration in volume rendering is the limited texture memory. *NVIDIA GeForce Ti4200 128 MB Graphics Card* has been used for the following results. In this medium, our implementation failed to compute *Dot3 Bump Mapping* with the datasets having more than 128x128x128 size of voxels. So, all the datasets had to be resampled to this size in order to achieve more meaningful comparisons.

Analyze 5.0 is a sophisticated software that presents many different options for each rendering methods. Following rendering results of Analyze 5.0 Volume Render module are captured with options that we think are the most similar to the methods of rendering used in our implementation.

Figure 5.1, 5.2, 5.3, 5.4, 5.5 gives rendering results of Analyze 5.0 Volume Render module and our implementation side by side for the same datasets with the methods of depth shading, gradient shading, volume compositing, maximum intensity projection and summed voxel projection respectively. The rendering results seen on the left side are rendered with Analyze 5.0 Volume Render module and the images seen on the right side are rendered with our implementation.

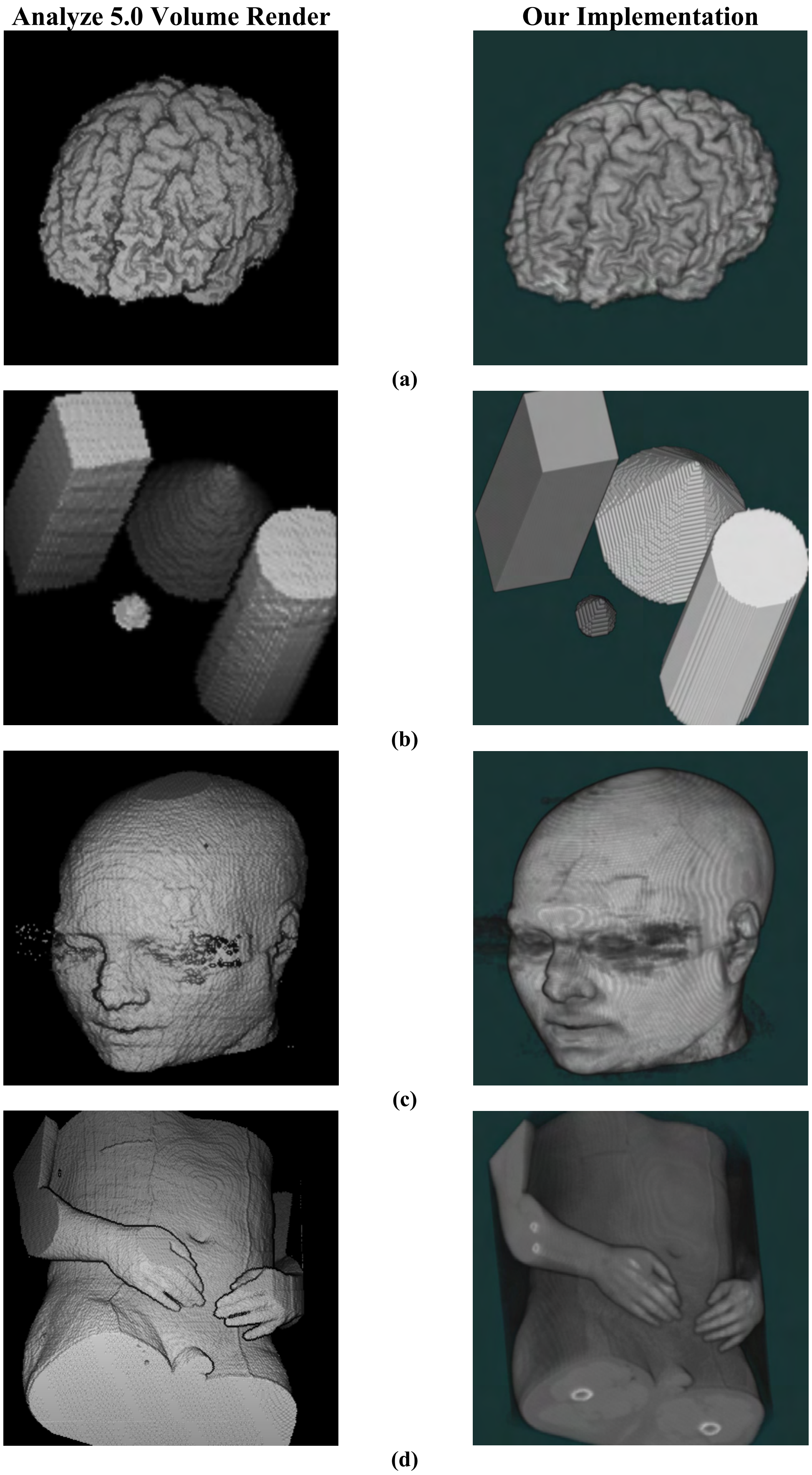
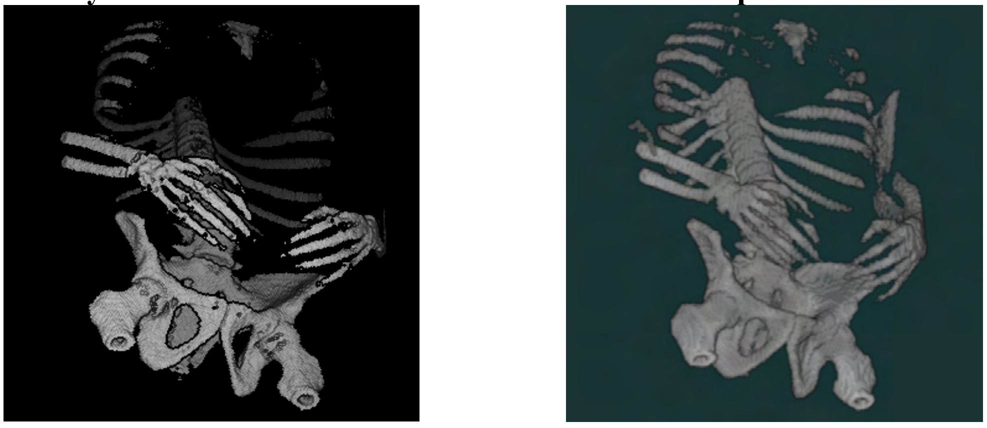


Figure 5.1 Depth Shading Comparisons

Analyze 5.0 Volume Render **Our Implementation**

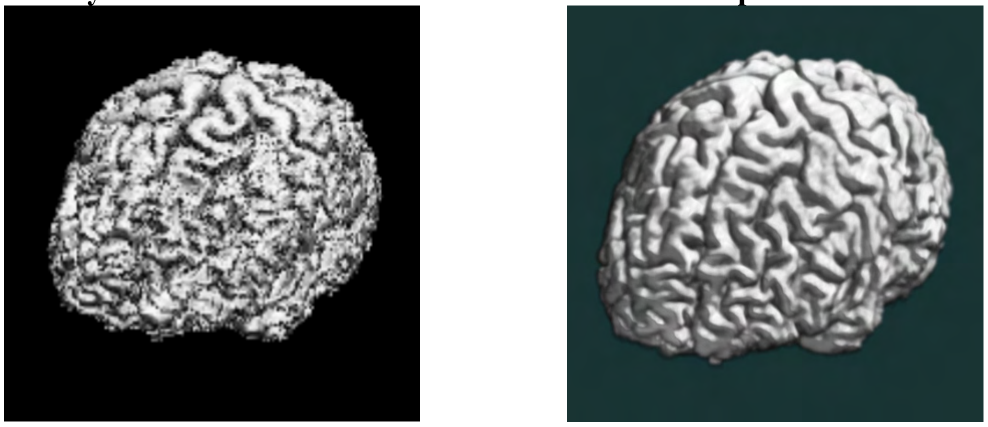


(e)

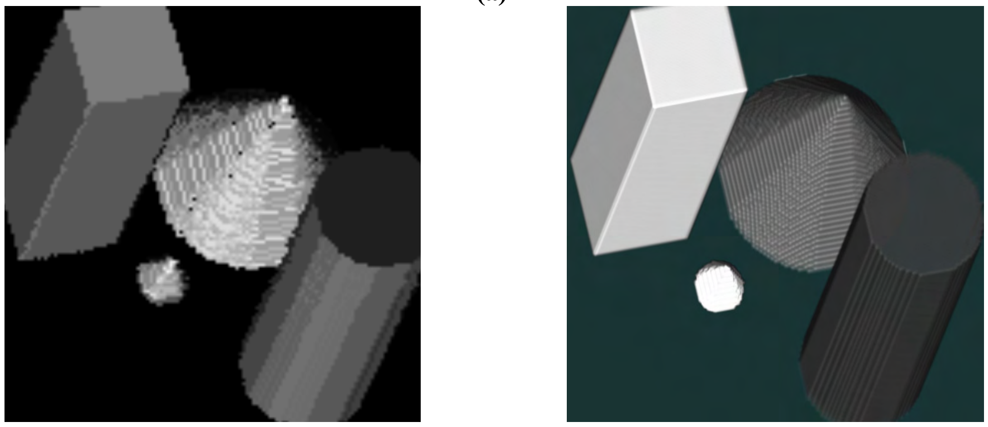
Figure 5.1 Depth Shading Comparisons

(a) Brain dataset; (b) Artificially built test geometries; (c) Head dataset; (d) Human body dataset; (e) Same human body dataset of (d) with a different threshold filter function. *The images on the left side are rendered with Analyze 5.0 Volume Render module, the images on the right side are rendered with our implementation.*

Analyze 5.0 Volume Render **Our Implementation**



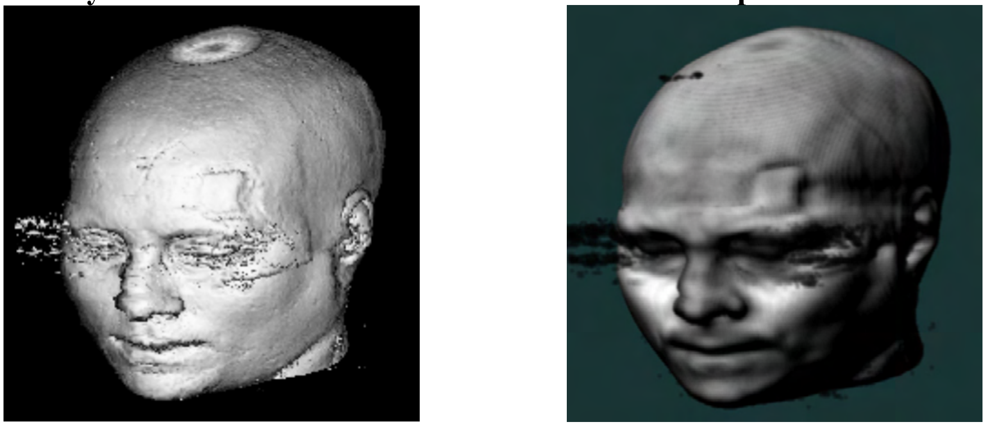
(a)



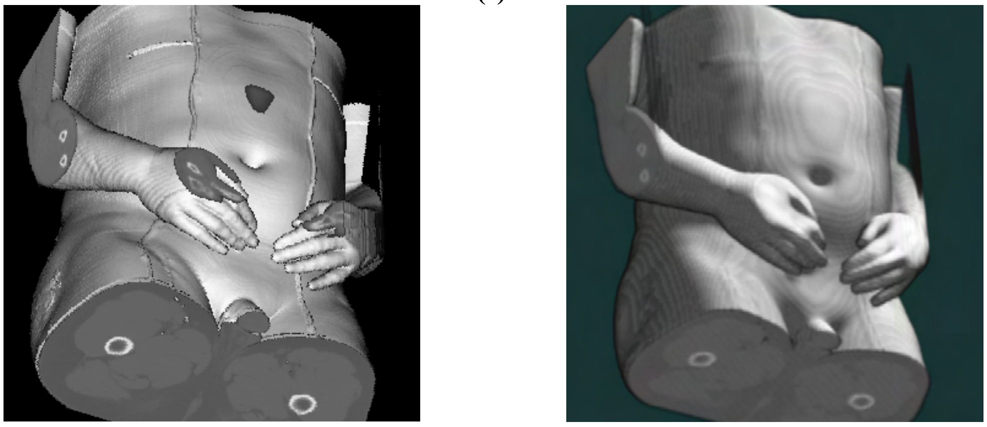
(b)

Figure 5.2 Gradient Shading Comparisons

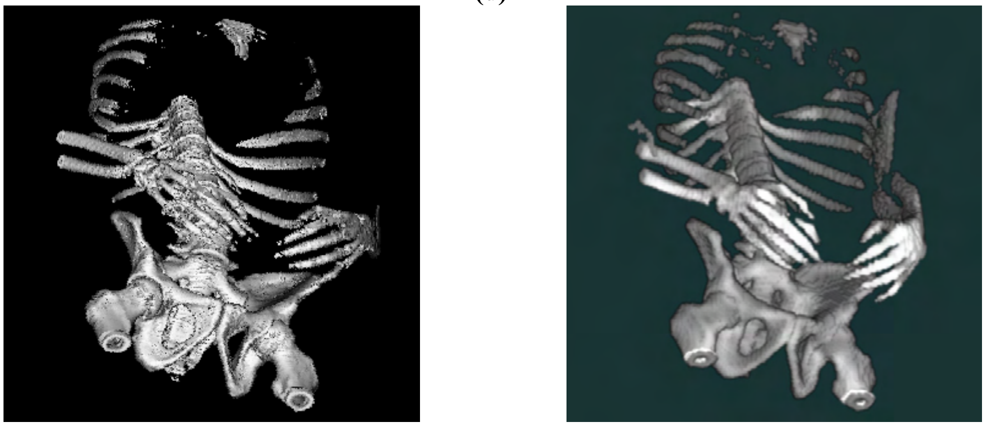
Analyze 5.0 Volume Render Our Implementation



(c)



(d)



(e)

Figure 5.2 Gradient Shading Comparisons

(a) Brain dataset; (b) Artificially built test geometries; (c) Head dataset; (d) Human body dataset; (e) Same human body dataset of (d) with a different threshold filter function. *The images on the left side are rendered with Analyze 5.0 Volume Render module, the images on the right side are rendered with our implementation.*

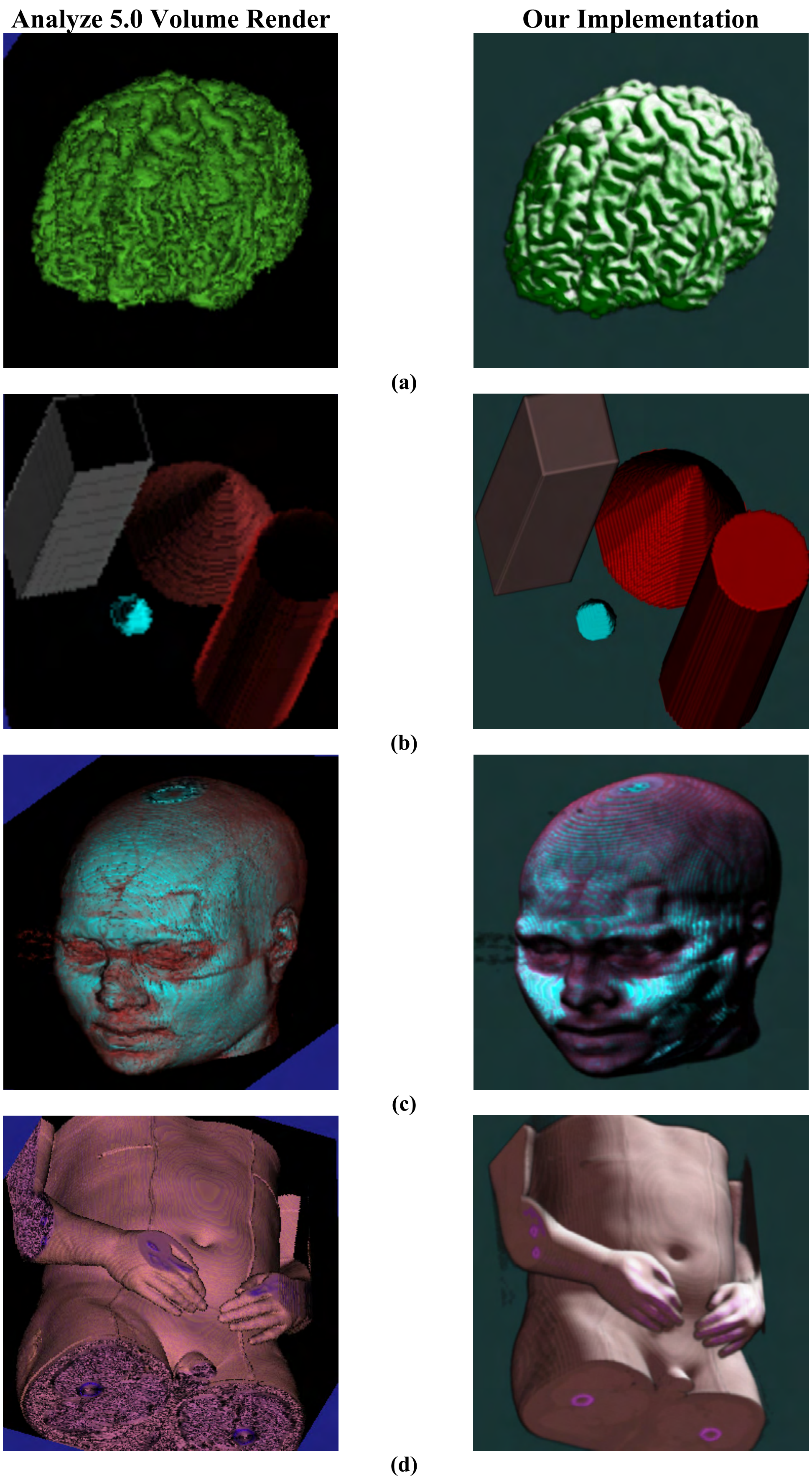
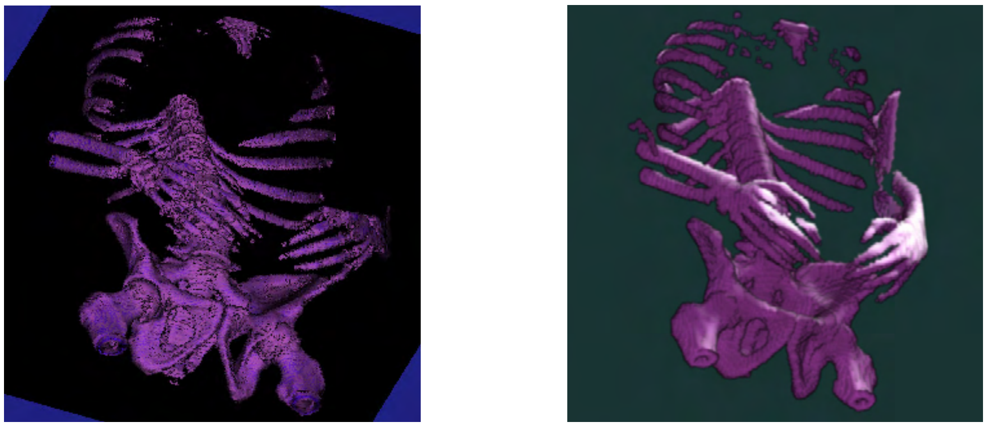


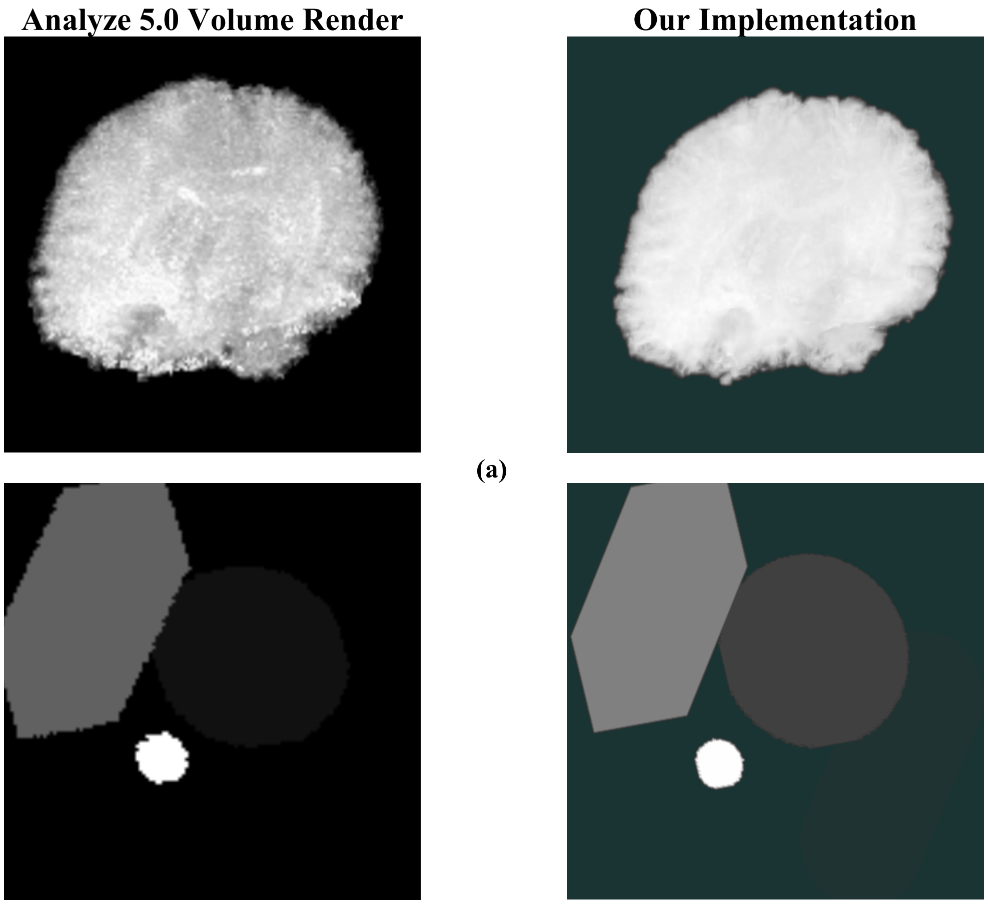
Figure 5.3 Volume Compositing Comparisons



(e)

Figure 5.3 Volume Compositing Comparisons

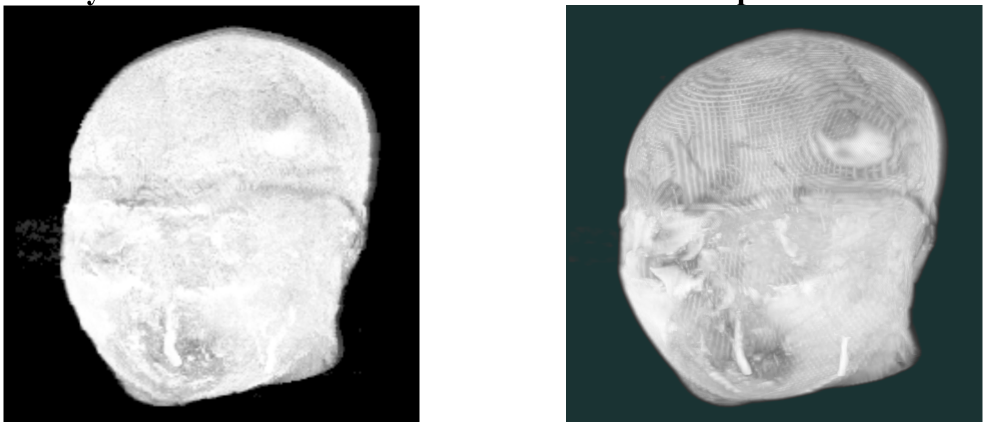
(a) Brain dataset; (b) Artificially built test geometries; (c) Head dataset; (d) Human body dataset; (e) Same human body dataset of (d) with a different threshold filter function. *The images on the left side are rendered with Analyze 5.0 Volume Render module, the images on the right side are rendered with our implementation.*

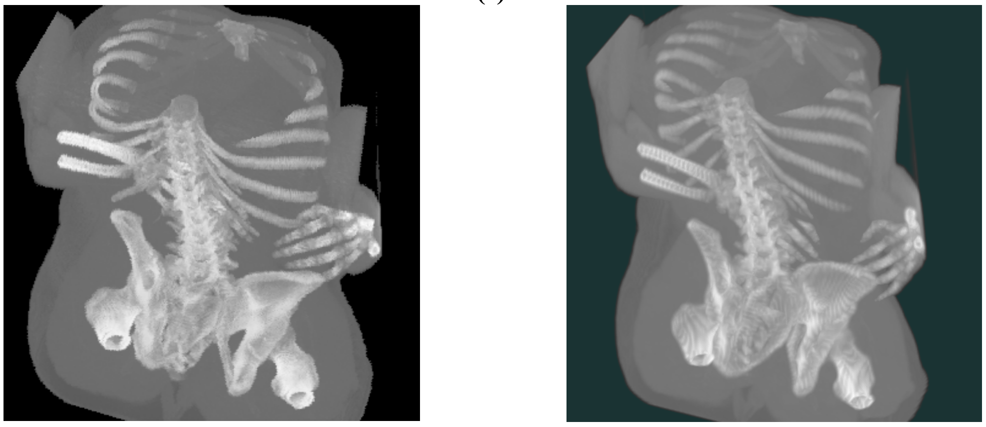


(b)

Figure 5.4 Maximum Intensity Projection Comparisons

Analyze 5.0 Volume Render **Our Implementation**


(c)


(d)

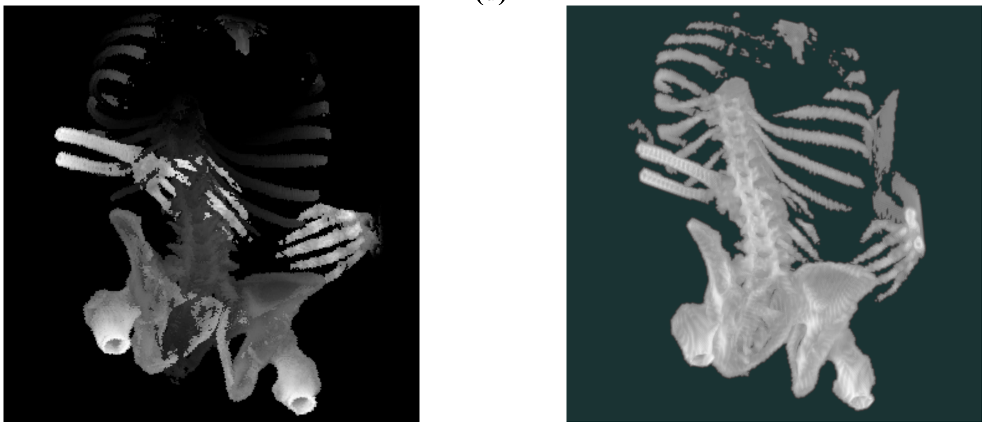

(e)

Figure 5.4 Maximum Intensity Projection Comparisons

(a) Brain dataset; (b) Artificially built test geometries; (c) Head dataset; (d) Human body dataset; (e) Same human body dataset of (d) with a different threshold filter function. *The images on the left side are rendered with Analyze 5.0 Volume Render module, the images on the right side are rendered with our implementation.*

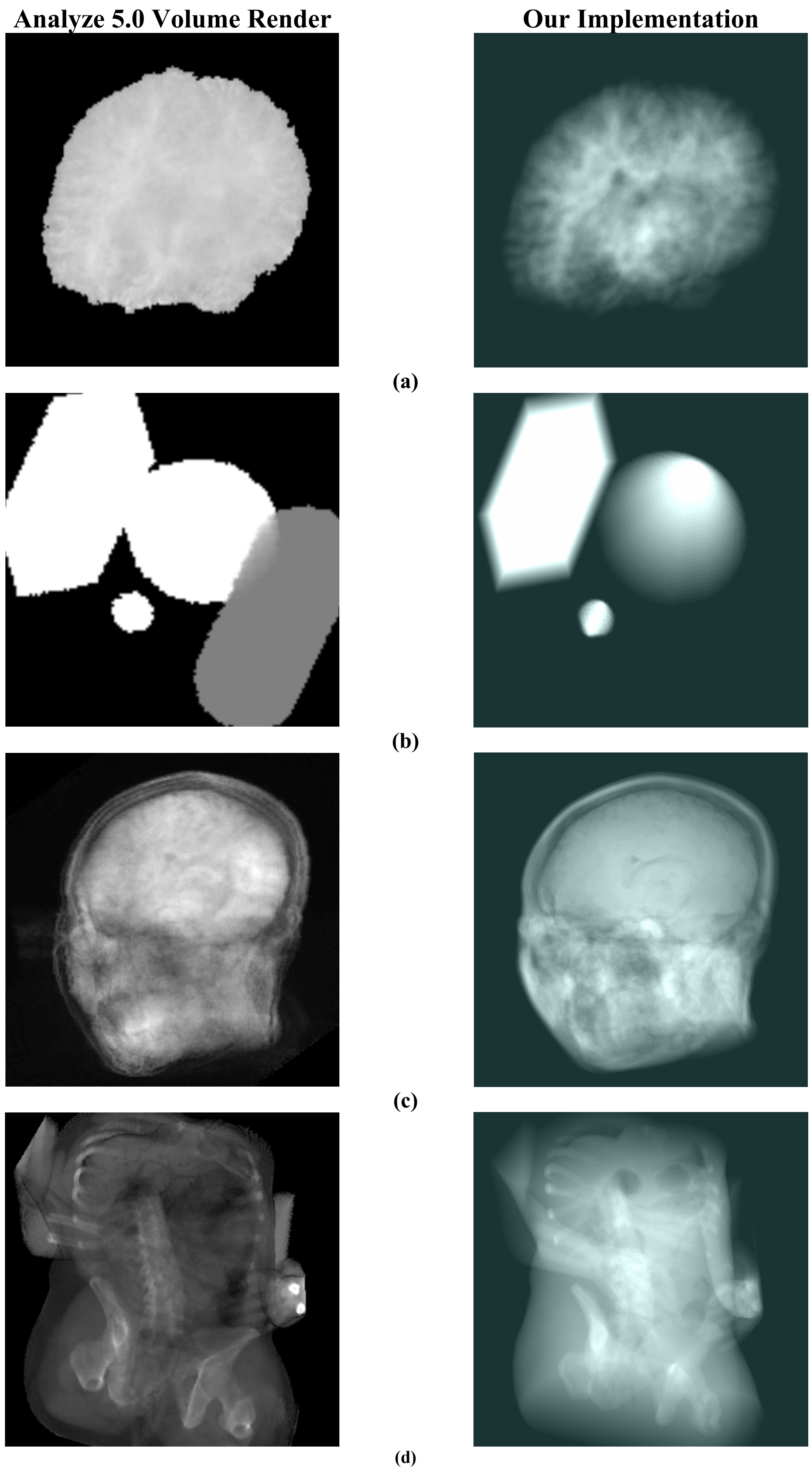
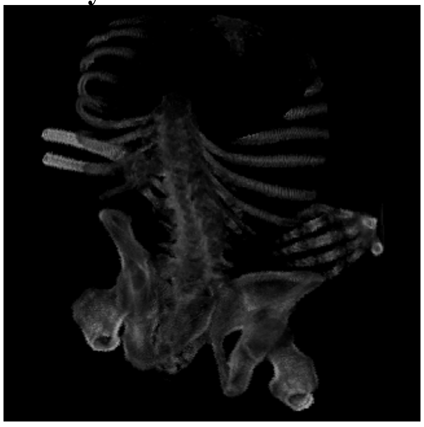
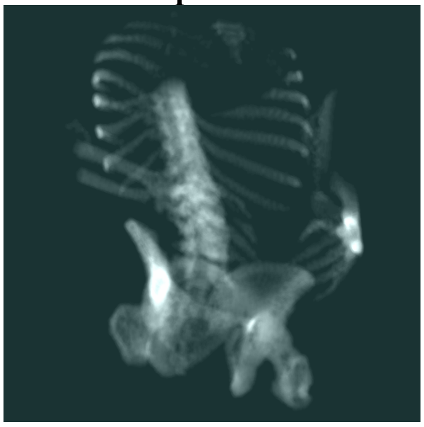


Figure 5.5 Summed Voxel Projection Comparisons

Analyze 5.0 Volume Render **Our Implementation**





(e)

Figure 5.5 Summed Voxel Projection Comparisons

(a) Brain dataset; (b) Artificially built test geometries; (c) Head dataset; (d) Human body dataset; (e) Same human body dataset of (d) with a different threshold filter function. *The images on the left side are rendered with Analyze 5.0 Volume Render module, the images on the right side are rendered with our implementation.*

In the comparisons of rendering without lighting (depth shading), it is seen that our software renders smoother on the surfaces, and Analyze software is more successful on showing the edges. In the lighting (gradient shading, volume compositing) comparisons, soft tissues that have complicated surfaces, like brain seems to be rendered more smoothly with our implementation, however Analyze software outputs gives more details for the model with a specular lighting effect. In both of the software, there are different kinds of options which enable the user to change the rendering output, operate with different methods. The images that we have used in the comparisons are the ones that were rendered with most similar rendering options.

Render image sets have been examined by a nuclear medicine expert who is part of the project *"Three Dimensional Brain Image Processing"*. She gave the following comments for the outputs:

a) The patient seems to have a disorder on his right hand in our outputs. The volumetric datasets do not always contain whole of the acquired objects. For example, the dataset used in Figure 5.1-5.5 (d) and (e) contain information of a human body, seems to have a disorder on his right hand in our implementation outputs. However as clearly seen in the Analyze 5.0 results, the information of the whole right hand of the body is not present in the dataset. Our implementation does not provide a warning indicator for the lacking parts that are cut because of the edges of the datasets. (As a future work, this problem shall be solved by changing contrast or color on the parts that are at the edge of the datasets.)

b) Using a larger kernel while calculating the gradient provides smoother passes between voxels and the surfaces are seen soft. However, this leads to a difficulty in perceiving the information by the user. (This comment will be in our future studies for achieving smooth surfaces without decreasing the details of the information in the datasets.)

5.2 Hardware Tests

This section provides information about the performance of the implementation.

Table 5.1 Testing results

Hardware and Operating System:	*OS:* Microsoft Windows XP *GPU:* NVIDIA GeForce 4 Ti 4200 *Texture Memory:* 128MB *RAM*: 448MB *CPU*: Intel Celeron, 416Mhz	*OS:* Microsoft Windows 2000 *GPU:* NVIDIA GeForce 4 Ti 4200 *Texture Memory:* 128MB *RAM*: 448MB *CPU*: Intel Celeron, 416Mhz	*OS:* Microsoft Windows XP *GPU:* NVIDIA GeForce 4 Ti 4200 *Texture Memory:* 128MB *RAM:* 1.00GB *CPU:* Intel Pentium 4, 2.01Ghz	*OS:* Microsoft Windows XP *GPU:* NVIDIA GeForce 5600 XT *Texture Memory:* 256MB *RAM* 512MB: *CPU*: Intel Pentium 4, 2.40Mhz
Dataset name and size:	MR dataset of a patient's body, 128 x 128 x 128 voxels			
Dataset Loading Time:	24 seconds	24 seconds	7 seconds	6 seconds
Rendered image size:	1024x712 pixels			
Frame Rates (Frames per second):	*No Bump Mapping:* 10fps *With Bump Mapping:* 5fps	*No Bump Mapping:* 10fps *With Bump Mapping:* 5fps	*No Bump Mapping:* 10fps *With Bump Mapping:* 5fps	*No Bump Mapping:* 10fps *With Bump Mapping:* 5fps

Table 5.1 Testing results

Dataset name and size:	MR dataset of a patient's foot, 256 x 128 x 128 voxels			
Dataset Loading Time:	47 seconds	47 seconds	13 seconds	13 seconds
Rendered image size:	1024x712 pixels			
Frame Rates (Frames per second):	*No Bump Mapping:* 5fps *With Bump Mapping:* 2-3fps	*No Bump Mapping:* 5fps *With Bump Mapping:* 2-3fps	*No Bump Mapping:* 5fps *With Bump Mapping:* 2-3fps	*No Bump Mapping:* 5fps *With Bump Mapping:* 2-3fps
Dataset name and size:	MR dataset of a patient's body, 256 x 256 x 256			
Dataset Loading Time:	14 minutes 25 seconds	14 minutes 30 seconds	48 seconds	3 minutes 25 seconds
Rendered image size:	1024x712 pixels			
Frame Rates (Frames per second):	*No Bump Mapping:* 4fps *With Bump Mapping:* FAILED	*No Bump Mapping:* 4fps *With Bump Mapping:* FAILED	*No Bump Mapping:* 4fps *With Bump Mapping:* FAILED	*No Bump Mapping:* 4fps *With Bump Mapping:* 2fps

As it is seen in the Table 5.1, large datasets (256 x 256 x 256 voxels or more) fail when rendered with lighting by graphics devices having texture memory size 128MB. However the same dataset was successfully rendered with a larger texture memory sized (256MB) graphics device. This shows that our method is dependent on the texture memory size of the graphics device. The loading time of a dataset is mainly determined with the size of the system RAM rather than the performance of the CPU.

CHAPTER 6

CONCLUSION AND FUTURE WORKS

6.1 Conclusion

In this study, we developed an implementation for visualizing *Cartesian grid* volumetric datasets, using PC graphics hardware. The entertainment market caused the graphics hardware producers to develop more sophisticated devices in the recent years. This fast evolution enables new technologies to be used in the scientific visualization field. We tried to develop an application that can be a basis for our future works for implementing end user software for visualizing medical images. By this study we gained some experience on the abilities of graphics hardware and experience on implementing real-time applications to render volumetric datasets for different clinical expectations in medical image visualization.

Testing results of the implementation shows that the performance of the rendering process is directly related with the capabilities of the graphics hardware. This gives an opportunity to achieve a high performance rendering with a low CPU power PC. The rapid and extensive consumption and production of the graphics hardware devices makes the prices become less everyday. This gives an opportunity to the users to obtain real-time rendering solutions without upgrading all of the system or spending money on high priced workstations.

Comments given by the nuclear medicine expert showed us that our implementation requires some improvements, especially for indicating the edges of the volumetric datasets and avoiding the decrease in the detail of the visualized information while increasing the smoothness of the surfaces.

In order to improve the visualization results and develop better tuning techniques, medical stuff should further use and test this software. This will also help in emergence of new ideas for future implementations. The environment we have chosen for the implementation provides a lot of flexibility in the rendering process to achieve such improvements.

Dot3 bump mapping technique has been used for creating a per-voxel lighting effect in OpenGL. Segments section of the implementation simulates the result of visualizing a segmented dataset. Trilinear interpolation capability of OpenGL 3D Texture Mapping enables the GPU to calculate the all the texels to be mapped on a polygon. Using this ability 3-axis plane command has been implemented. Multi modality visualization section of the software displays two different volumetric datasets in a single scene. Our implementation uses two different display modes for visualizing multi modality datasets.

The outputs of the implementation have been compared with worldwide used medical image analysis software. The visualization methods that are compared are: depth shading, gradient shading, volume compositing, maximum intensity projection, summed voxel projection.

The development time spent for the implementation is about 8 months. 18 classes consist of about 14000 lines of code has been written for the implementation. (More than 5000 lines of this code have been automatically generated by some development tools for the user interface part of the software.)

6.2 Future Works

The aim of our implementation was to gain experience and knowledge on volume rendering using graphics hardware. The focus of our implementation was to achieve high performance rendering by using GPU. Therefore, the user interface, PC memory and CPU use efficiency was not the main issue in the development process. The testing results show that the loading time of a volumetric dataset is very much dependent on the size of the system memory size. We think that system memory size shall always be an issue in loading large sized datasets. However, our implementation requires a modification for the management of the memory.

Programming Language: Our implementation has been built using Java programming language. The reason for this selection was mainly because of past experience. This platform has many advantages for our application; for example, it does not require an installation and can be operated from a CD, it is not dependent on the operating system. However, we observed some low performance of creating objects in Microsoft Windows platform. For example, creation of an object of a user interface element requires more time than an application built with C++. Our implementation focus was to use the graphics hardware as efficient as possible, so the part of the code that processes the rendering was kept out of any kind of computation that requires the application to use the CPU time or create any objects. The testing results show that rendering time does not depend on CPU power. It might not be wrong to conclude that changing the programming platform shall not change the rendering performance. However, there is a need to test the same methods of rendering with different programming languages to prove this thought.

Data Format: Implementation supports only the *Analyze* volumetric data format. Libraries used for loading data supports DICOM formatted datasets to be read. This format and other formats can be supported if other libraries were made available.

As specified in the other sections, volume rendering can be used in many different fields of scientific visualization. Areas which are interested in visualizing data which are formed of grid type that is not a Cartesian grid may use this application by implementing a data import module which makes interpolation and converts the input into Cartesian grid format.

Large Datasets: Testing results shows that the main obstacle in visualizing a large volumetric dataset using graphics hardware is the size of the texture memory of the graphic device. For example, *bump mapping* technique that we have used for diffuse lighting here, requires the volume and its gradient array to be loaded to the texture memory. While visualizing a large dataset with a graphics hardware that has a 256MB of texture memory, this method works; however the same data fails with a graphics device having 128MB of texture memory. There are some ways for solving this issue: using Pixel Shader 2.0 API in computing the ray casting methods with hardware acceleration [33], "Trex, a scalable system that takes advantage of parallel graphics hardware, software based compositing, and high performance I/O" [34] are some of the researches about this problem.

Volume Rendering combined with Surface Rendering: There are some methods for extracting surfaces from volumetric dataset [3], [2], [7]. OpenGL and DirectX presents an environment in which texture based volume rendering can be combined with surface based volume rendering technique results. It may be a future work to implement one of the surface rendering techniques and render both results together.

Stereo Viewing: OpenGL supports simulation of stereoscopic vision with suitable devices [71]. Stereo rendering might be useful for visualizing the datasets with a three dimensional view in some cases.

Material Assignment: Segmentation phase of volume rendering enables the different kind of materials in the volume to be distinguished. Implementing a method which enables the user to assign different material properties to specify how the materials in the volume reflects light might create more photorealistic rendering results.

In order to render segmented volumetric datasets, there is a requirement for defining a data format which specifies how the labeling information shall be stored in the volumetric dataset. The implemented segmentation section was done for simulating purposes as there is no labeling information exists in the data format yet.

Web based volume rendering: The capabilities of volume rendering might be useful for medical staff. However, the method that we have proposed is dependent on graphics processing unit power and texture memory. In an institution like a hospital employing large numbers of medical staff, this method requires a huge expense for upgrading the graphics processing devices of the personal computers of the staff. A web based solution might decrease this expense and enables more medical stuff to have the advantage of the three dimensional visualization. A web based solution requires an implementation that renders the specified volumetric datasets in a server and broadcasts the rendering results as compressed two dimensional images, so that dummy terminals or personal computers having no graphics accelerator devices can be able to visualize the patient's acquired three dimensional medical datasets from a browser. This method shall decrease the interactivity of rendering but might present a useful solution for a wide area of use with minimum expense. This method also requires the server side to be more powerful in rendering as the number of clients that use this process increase. Using parallel graphics hardware [34] might be a pathway to solve this issue.

REFERENCES

[1] Robb, R. A., (2000). Three-Dimensional Visualization in Medicine and Biology. Handbook of Medical Imaging (pp. 685-712). San Diego, Academic Press.

[2] Schilling, A., Klein, R., (1998, April). Fast generation of multiresolution surfaces from contours. Proc. Eurographics Workshop, Blaubeuren. 35-46.

[3] Lorensen, W.E., Cline, H. E., (1987, July). Marching cubes: A high resolution 3D surface construction algorithm. ACM Computer Graphics, (Proc. SIGGRAPH) 21(4). 163-169.

[4] Berk, H., Aykanat, C., Güdükbay U., (2003) Direct volume rendering of unstructured grids. Computers and Graphics, Volume 27, Issue 3 (June 2003) (pp. 387-406). Elsevier Science

[5] Udupa, J.K., (1991). Computer aspects of 3D imaging in machine: a tutorial. 3D Imaging in Medicine. (pp. 1-69). Baton, Florida, CRC Press, Inc.

[6] Lichtenbelt B., Crane R., Naqvi S., (1998) Introduction To Volume Rendering, Prentice Hall.

[7] Chuang, J. H., Lee, W. C., (1995) Efficient Generation of Isosurfaces in Volume Rendering. Computer & Graphics, Volume 19(6). 805-813.

[8] Phong, B.,T., (1975). Illumination for computer generated images. Comm. ACM 18, 6 (June). 311-317.

[9] Levoy, M., (1988). Display of surfaces from volume data. IEEE Computer Graphics and Applications, 8(3). 29-37.

[10] Swan, J. E., (1998). Object-Order Rendering of Discrete Objects, PH.D. Thesis, The Ohio State University, Department of Computer and Information Science

[11] Walsum, T., Hin, A. J. S., Versloot, J., Post F. H., (1992). Efficient hybrid rendering of volume data and polygons. Advances in Scientific Visualization. (pp. 83-96). Springer-Verlag Berlin-Heidelberg.

[12] Yagel, R., Kaufman, A., (1992). Template-based volume viewing. Computer Graphics Forum, 11(3). (pp. 153-167)

[13] Yagel, R., Shi, Z., (1993). Accelerating volume animation by space-leaping. In Proceedings of IEEE Visualization 1993. (pp. 62-69)

[14] Levoy M., (1990), Efficient ray tracing of volume data. ACM Transactions on Graphics, 9(3). (pp. 245-261)

[15] Cohen D., Sheffer Z., (1994). Proximity clouds: An acceleration technique for 3D grid traversal. The Visual Computer, 11(1). (pp. 27-38)

[16] Freund, J. L., Sloan K., (1997) Accelerated volume rendering using homogenous region encoding. In Proceedings of Visualization 1997. (pp. 191-196)

[17] Stander, B. T., Hart, J. C., (1994). A Lipschitz method for accelerated volume rendering. In Proceedings of the Symposium on Volume Visualization 1994. (pp. 107-114).

[18] Westover, L., (1990). Footprint evaluation for volume rendering. Computer Graphics, 24(4). (pp. 367-376)

[19] Upson, C., and Keeler, M., V-BUFFER, (1988) Visible Volume Rendering. Computer Graphics (proceedings of SIGGRAPH), 22(4), (pp. 154–159).

[20] Lacroute, P., Levoy, M., (1994). Fast volume rendering using a shear-warp factorization of the viewing transformation. Computer Graphics, 28(Annual Conference Series). (pp. 451-458)

[21] Knittel, G., (1995). A scalable architecture for volume rendering. Computers & Graphics, 19(5). (pp. 653-665).

[22] Günther, T., Poliwoda, C., Reinhart, C., Hesser, J., Männer, R., Meinzer, H. P., Baur. H. J., (1995) VIRIM: A massively parallel processor for real-time volume visualization in medicine. Computers & Graphics, 19(5). (pp. 705-710)

[23] Meißner, M., Kanus, U., Wetekam, G., Hirche, J., Ehlert, A., Straßer, W., Doggett, M., Proksa. R., (2002). VIZARD II: A reconfigurable interactive volume rendering system. In Proceedings of the Workshop on Graphics Hardware 2002. (pp. 137-146)

[24] Osborne, R., Pfister, H., Lauer, H., McKenzie, N., Gibson, S., Hiatt, W., Ohkami, T., (1997). EM-Cube: An architecture for low-cost real-time volume rendering. In Proceedings of the Workshop on Graphics Hardware 1997. (pp. 131-138).

[25] Westermann, R., Ertl, T., (1998). Efficiently using graphics hardware in volume rendering applications. In Proc. of SIGGRAPH 1998. (pp. 169-177)

[26] Weiskopf, D., Weiler, M., Ertl, T., (2004) Maintaining Constant Frame Rates in 3D Texture-Based Volume Rendering, Computer Graphics International, 2004. Proceedings. University of Stuttgart
(pp. 604- 607)

[27] Sadleir, R. J., Whelan, P. F., Bruzzi, J. F., Moss, A. C., MacMathuna, P., Fenlon H.M., (2002). A portable toolkit for providing straightforward access to medical image data. Radiology 225(Suppl):762

[28] Porter, T., Duff, T., (1984). Compositing digital images. ACM Computer Graphics (SIGGRAPH '84 Proceedings), volume 18. (pp. 253-259)

[29] Woo M., Neider J., Davis T., Shreiner D., OpenGL Architecture Review Board, (1997). OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL, Version 1.1 (2nd Edition). Silicon Graphics, Inc.

[30] Blinn J. F., (1978). Simulation of wrinkled surfaces. ACM Computer Graphics (SIGGRAPH '78 Proceedings), volume 12. (pp. 286-292)

[31] Vannier, M. W., Marsh, J. L., Warren, J. O., (1983). Three Dimensional Computer Graphics for Craniofacial Surgical Planning and Evaluation. ACM Computer Graphics, (Proc. SIGGRAPH'83) 17(3). (pp. 263-273)

[32] Höhne, K. H., Riemer, M., Tiede, U., Bomans. M., (1988) Volume rendering of 3D-tomographic imagery. In C. N. de Graaf, M. A. Viergever (eds.): Information Processing in Medical Imaging, Proc. IPMI-10. (pp. 403-412) Plenum Press, New York.

[33] Krüger, J., Westermann, R. (2003). Acceleration Techniques for GPU-based Volume Rendering. Proceedings of IEEE Visualization, 2003. 287-292.

[34] Kniss, J., McCormick. P., McPherson, A., Ahrens J., Painter J., Keahey A. (2001). Interactive Texture-Based Volume Rendering for Large Data Sets. Computer Graphics and Applications, IEEE. 52-61.

[35] Hearn D, Baker M. P., (1997). Computer Graphics, C Version, Prentice Hall, Inc.

[36] Steenberg, E., (2004, February). Real-time global illumination. Game Developers Conference, San Jose USA

Web References:

[37] Mayo Clinic, Software: Analyze Program, URL: http://www.mayo.edu/bir/Software/Analyze/Analyze.html, Last visited date: 30-08-2004

[38] Imaging Technology Group. URL: http://www.itg.uiuc.edu/publications/forums/1998-03-17/multiplanar.htm, Last visited date: 30-08-2004

[39] Barré, S., Volume Visualization Web Page, URL: http://www.barre.nom.fr/medical/these/pictures-3.html, Last visited date: 30-08-2004

[40] What is haptics?, URL: http://iroi.seu.edu.cn/books/ee_dic/whatis/haptics.htm, Last visited date: 30-08-2004

[41] The Visible Human Project, URL: http://www.nlm.nih.gov/research/visible/visible_human.html, Last visited date: 30-08-2004

[42] Stanford Encyclopedia of Philosophy, http://plato.stanford.edu/entries/qt-uncertainty/, Last visited date: 30-08-2004

[43] Wong K., Volume Rendering Techniques, CSIS7501 Advance Computer Graphics & Virtual Reality Lecture Notes, URL: http://www.csis.hku.hk/~c7501/, Last visited date: 30-08-2004

[44] 12.4AU1 Computer Graphics: A Course in three-dimensional Computer Graphics, URL: http://www.ece.eps.hw.ac.uk/~dml/cgonline/hyper00/polypipe/render/illum.html, Last visited date: 30-08-2004

[45] Power K., Illumination and Shading Models, URL: http://glasnost.itcarlow.ie/~powerk/Graphics/Notes/node10.html, Last visited date: 30-08-2004

[46] Sullivan J. A., Illumination and Shading, URL: http://www.nps.navy.mil/cs/sullivan/MV4470/resources/notes/IlluminationAndShading.ppt, Last visited date: 30-08-2004

[47] Illumination Models, URL: http://graphics.cs.msu.su/courses/cg99/notes/lect11/illum_local.htm, Last visited date: 30-08-2004

[48] Szymczak A., Shading Models, URL: http://www.cc.gatech.edu/classes/AY2004/cs4451a_fall/smodels/linint.html, Last visited date: 30-08-2004

[49] Shen, H. W., Digital Image Compositing, URL: http://www.cse.ohio-state.edu/~hwshen/788/sp01/composite.ppt, Last visited date: 30-08-2004

[50] Heiler, M., Volume Rendering Project – Results, URL: http://www.cvgpr.uni-mannheim.de/heiler/volume.html, Last visited date: 30-08-2004

[51] Brown, J., Volume Rendering Project, URL: http://wwwcsif.cs.ucdavis.edu/~brownjb/arch/graphics_classes/177/3/, Last visited date: 30-08-2004

[52] ATI Technologies Inc., URL: http://www.ati.com/, Last visited date: 30-08-2004

[53] NVIDIA Corporation, URL: http://www.nvidia.com/page/home, Last visited date: 30-08-2004

[54] Blythe, D., Advanced Graphics Programming Techniques Using OpenGL, Silicon Graphics, URL: http://www.opengl.org/resources/tutorials/sig99/advanced99/notes/node298.html, Last visited date: 30-08-2004

[55] Volume Rendering, Visualization Process, URL: http://www.cse.ohio-state.edu/~hwshen/788/VR.ppt, Last visited date: 30-08-2004

[56] Kniss, J., Hadwiger, M., Rezk-Salama, C., Westermann R., Kindlmann, G., High-Quality Volume Graphics on Consumer PC Hardware, VIS2002, URL: http://www.sci.utah.edu/~jmk/vis02/Part2_TextureBased.ppt, Last visited date: 30-08-2004

[57] Blythe, D., Advanced Graphics Programming Techniques Using OpenGL, Silicon Graphics, URL: http://www.opengl.org/resources/tutorials/sig99/advanced99/notes/node306.html, Last visited date: 30-08-2004

[58] Java3D, URL: http://www.j3d.org/, Last visited date: 30-08-2004

[59] Java Technology, URL: http://java.sun.com/, Last visited date: 30-08-2004

[60] Java & Programming Language Bindings to OpenGL, URL: http://www.opengl.org/resources/java/, Last visited date: 30-08-2004

[61] Java bindings for OpenGL, URL: https://jogl.dev.java.net/, Last visited date: 30-08-2004

[62] JOGL User Guide, URL:

https://jogl.dev.java.net/nonav/source/browse/*checkout*/jogl/doc/userguide/index.html?rev=HEAD&content-type=text/html, Last visited date: 30-08-2004

[63] NeatMed, Medical Imaging Application Developer Interface, URL: http://www.eeng.dcu.ie/%7Evsl/DICOM/, Last visited date: 30-08-2004

[64] DICOM (Digital Imaging and Communications in Medicine), URL: http://medical.nema.org/, Last visited date: 30-08-2004

[65] Blythe, D., Advanced Graphics Programming Techniques Using OpenGL, Silicon Graphics, URL: http://www.opengl.org/resources/tutorials/sig99/advanced99/notes/node303.html, Last visited date: 30-08-2004

[66] Blythe, D., Advanced Graphics Programming Techniques Using OpenGL, Silicon Graphics, URL: http://www.opengl.org/resources/tutorials/sig99/advanced99/notes/node304.html, Last visited date: 30-08-2004

[67] Dot3 bump mapping in Java3D, Java3D Org. URL: http://www.j3d.org/tutorials/quick_fix/dot3_bumps.html, Last visited date: 30-08-2004

[68] Multitexturing with OpenGL, URL: http://tfpsly.planet-d.net/english/3d/multitexturing.html, Last visited date: 30-08-2004

[69] Bump Mapping, URL: http://www.paulsprojects.net/tutorials/simplebump/simplebump.html, Last visited date: 30-08-2004

[70] Mayo Clinic, College of Medicine, URL: http://www.mayo.edu/, Last visited date: 30-08-2004

[71] OpenGL on Silicon Graphics Systems, URL: http://www.parallab.uib.no/SGI_bookshelves/SGI_Developer/books/OpenGLonSGI/sgi_html/ch04.html#id76349, Last visited date: 30-08-2004

[72] OpenGL, URL: http://www.opengl.org, Last visited date: 30-08-2004

[73] Silicon Graphics, URL: http://www.sgi.com, Last visited date: 30-08-2004

[74] Blythe, D., Advanced Graphics Programming Techniques Using OpenGL, Silicon Graphics, URL: http://www.opengl.org/resources/tutorials/sig99/advanced99/notes/notes.html, Last visited date: 30-08-2004

[75] The VRML Consortium. The virtual reality modeling language specification. web site, URL: http://vag.vrml.org, Last visited date: 30-08-2004

APPENDICES

APPENDIX A User Interface of the Implementation

1. File Menu

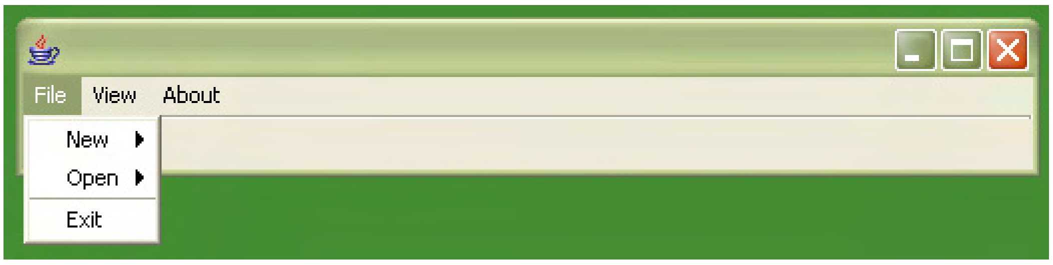


Figure A.1 Main Frame, File Menu

Application first generates the *main frame*.

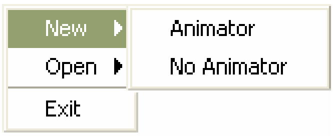


Figure A.2 New Menu

New menu, is used for generating a *GL window* that is shown on Figure A.3. GL window is the frame that the rendering results will be shown.

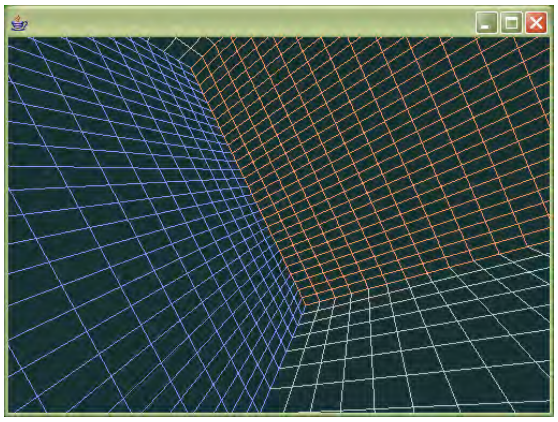


Figure A.3 GL Window (no dataset loaded)

GL window has two modes: *Animator* and *No Animator*. These states are chosen while creating a GL window on file menu as shown in the Figure A.2.

"Animator" state is implemented with a class called *Animator* which invokes the *display* event, whenever the system is ready. In other words, the scene is rendered again immediately after a rendering computation is over. This makes the scene to be animated and enables interactivity. See Appendix A-1 for more details of Animator class.

"No Animator" mode does not use *Animator* class for invoking the *display* event. Instead, in this state, *display* event is invoked by a timer object with a rate of 1/24 second that provides 24fps (frames per second) maximum refresh rate. This option has been added to the software because, in some systems, performance of the other running applications decreased while running this application. Maximum frame rate can be changed; details are given in Appendix A-2.

After creating a GL window, a dataset is chosen with the file open menu shown in the Figure A.4.

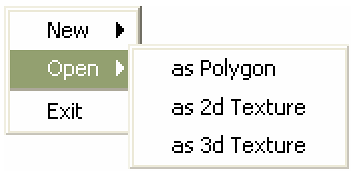


Figure A.4 File Open Menu

File open menu loads volumetric datasets that are encoded by Analyze dataset format. Loading the volume with *Polygon* and *2d Texture* modes have been developed for testing and learning purposes.

Loading a volumetric dataset as *Polygon* was an unsuccessful attempt to render a volumetric dataset with OpenGL. The method used in this mode is:

- Read each voxel in the dataset
- Assign a square shaped polygon in the scene for each voxel
- Place the squares at the coordinates with respect to the volumetric model

- Assign color values for the polygons same with the voxel intensity value
- Assign transparency values according to the intensity values of voxels

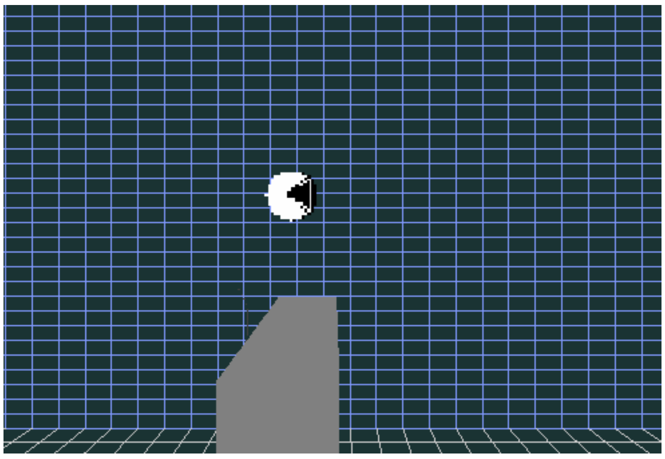


Figure A.5 Polygon Mode Output

The result was vast amount of polygons created and the graphics hardware was unable to render a simple 64x64x64 volumetric dataset with a performance of about 5 - 10 seconds for a frame. In order to increase the performance, number of voxels that shall be rendered is decreased with following method:

- Compute gradient vector for each voxel
- Eliminate the voxels that have gradient magnitude value smaller than a specific value

This attempt increased the performance about 2 frames per second for some datasets; however, the performance was still far away from interactive rates. Besides, filtering the gradient vector magnitude value more decreased the image quality and some part of the dataset was unable to be seen. If the graphics processing hardware was fast enough to render all the voxels in the dataset at interactive rates, the next attempts would have been:

- To assign vertex normal values to the vertices of the polygons and assign material properties according the segmentation information for the voxels, so that diffuse and specular lighting can be obtained on the rendering result.

- Make the polygons faces turned to the viewer all the time, so that where ever the user flies through in the scene or rotate the model, there would be no lost in quality because of the effect described in the topic "Volume Rendering Using 2D Textures" and shown in the Figure 3.7.

2D Texture mode was implemented only for the purpose of learning the environment and implementing the method described in "Volume Rendering Using 2D Textures" topic.

3D Texture mode loads a volumetric dataset to the GL Window and renders the volume by using 3D texture mapping capabilities of the hardware.

2. GL Window

GL window is the frame that the output of the rendering process is print on it. Each GL window is created as independent threads, so as to enable the user visualize more than one volumetric dataset, or visualize one dataset with more than one GL window at the same time, as shown on the Figure A.6.

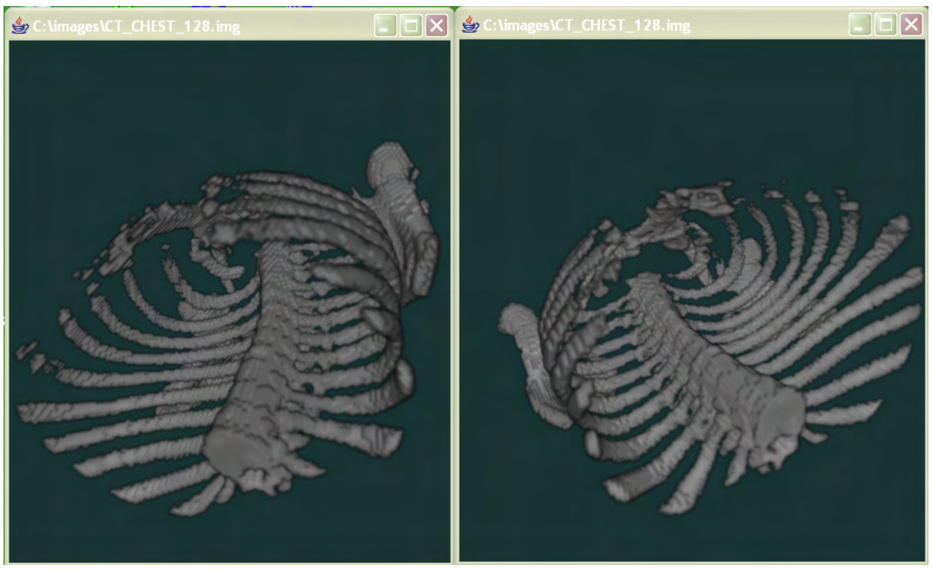
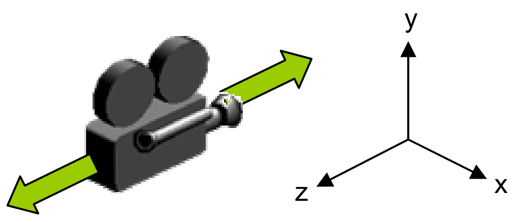


Figure A.6 Multiple GL windows.
Same dataset visualized with two different angles.

GL window present the user a three dimensional interactive environment that the user is able to move around. The interactivity controls implemented for the GL window is as follows:

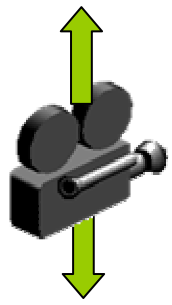
up arrow: fly ahead

down arrow: fly back



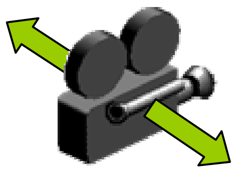
ctrl + *up arrow:* slide upwards

ctrl + *down arrow*: slide downwards



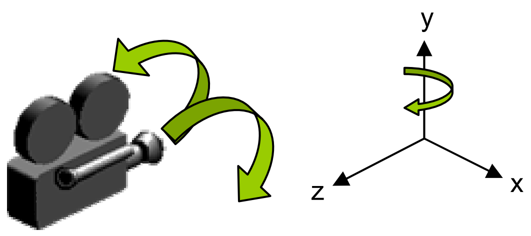
ctrl + *left arrow:* slide to left

ctrl + *right arrow*: slide to right



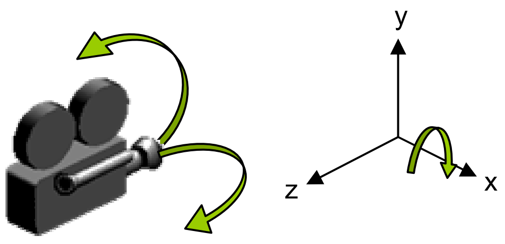
left arrow: rotate to left (about y axis)

right arrow: rotate to right (about y axis)



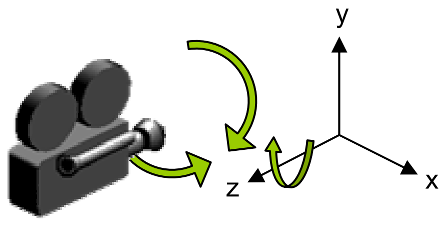
shift + *up arrow:* rotate to upwards (about x axis)

shift + *down arrow:* rotate to downwards (about x axis)



shift + *left arrow:* rotate to left along z axis

shift + *right arrow:* rotate to right along z axis



Mouse drag action while left mouse button is pressed, rotates the camera about the y and x axis, and rotates the camera about z axis when the right button of the mouse is pressed.

These transformations are not done to the model but the position of the camera is changed. When the key "R" is pressed, every action done by the mouse drag is applied as transformations to the loaded volumetric dataset. 3D Texturing gives an advantage here for the transformation of the model. The first attempt to rotate the volumetric dataset was to recalculate the new texture mapping coordinates values after rotation. This process had been done by matrix transformation operations. However, OpenGL gives an advantage to make transformations on the texture coordinates by changing the matrix multiplication mode into texturing by the following code:

```
glMatrixMode(GL_TEXTURE);
```

So, instead of calculating the new texture mapping coordinates with CPU, this process is done by GPU.

3. View

This section shall provide information about viewing properties implemented for different visualizing options. View menu is shown in Figure A.7.

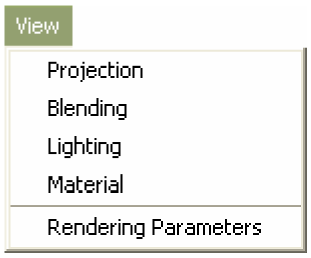


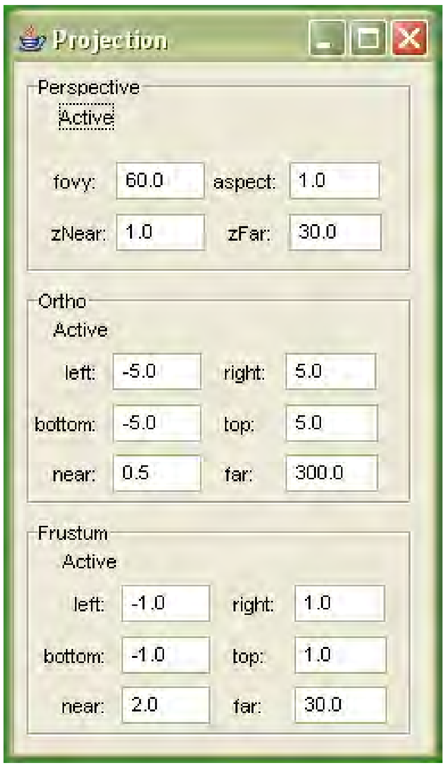
Figure A.7 View Menu

Projection, Blending, Lighting and *Material* options have been implemented for understanding the OpenGL platform in detail. These parts are not directly related with the volume rendering method. However, this experience showed that; in a volume rendering application which is implemented using hardware acceleration, there is a capability of rendering both surfaces and textures. This gives an ability to visualize both indirect volume rendering [3], [2], [7] and direct volume rendering methods together in this platform. Viewing options built, present an environment for implementing new methods of rendering for the future works.

Rendering Parameters appears in the view menu is related with the volume rendering part.

3.1 Projection

This section provides OpenGL projection rendering options. Details about OpenGL projection capabilities and methods are given in Appendix B-3.


Projection

Perspective
Active

fovy: 60.0 aspect: 1.0
zNear: 1.0 zFar: 30.0

Ortho
Active

left: -5.0 right: 5.0
bottom: -5.0 top: 5.0
near: 0.5 far: 300.0

Frustum
Active

left: -1.0 right: 1.0
bottom: -1.0 top: 1.0
near: 2.0 far: 30.0

Figure A.8 Projection GUI

Projection GUI seen in the Figure A.8 provides an interface for *gluPerspective, glOrtho* and *glFrustum* commands of OpenGL. Example rendering results for perspective and orthographic projections are seen on Figure A.9.

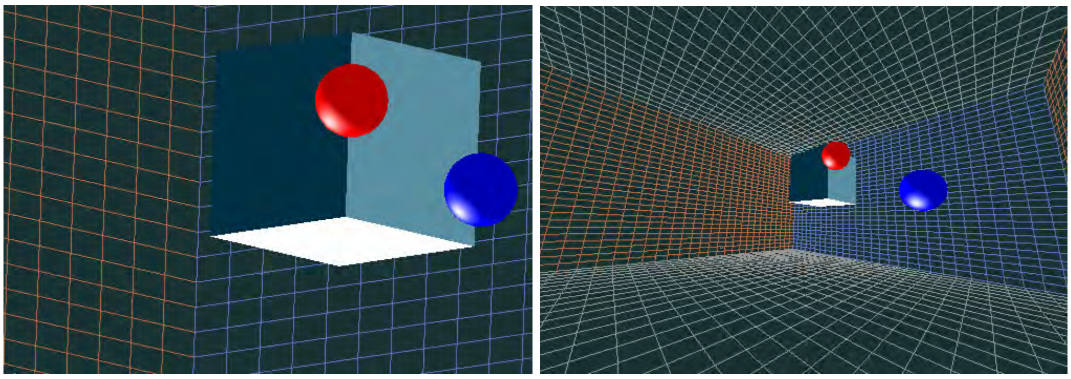
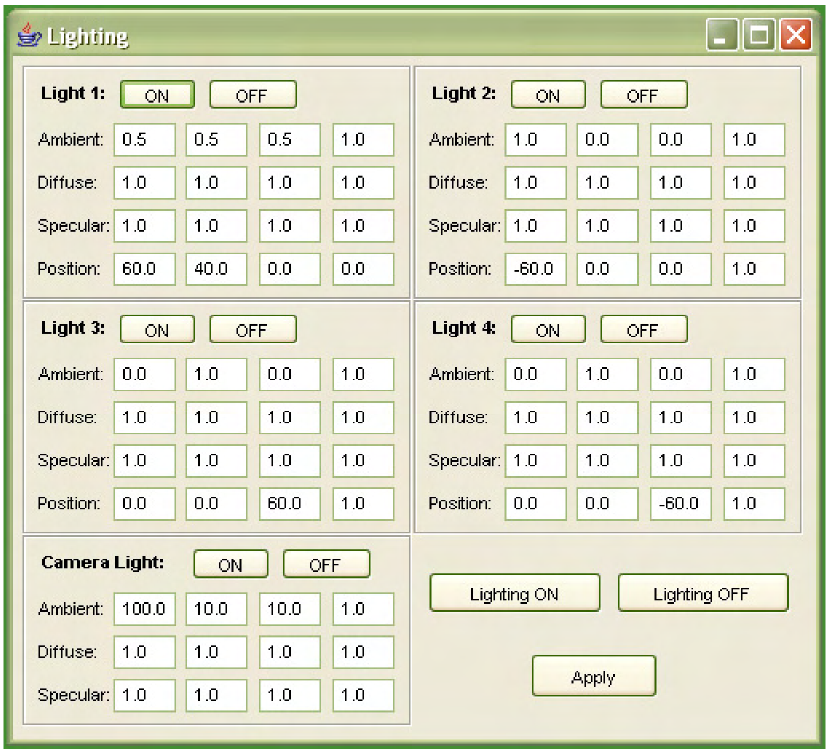


Figure A.9 Orthographic Projection (left), Perspective Projection (right)

3.2 Lighting

This section has been implemented with the purpose of learning OpenGL lighting capabilities. Information about lighting methods in OpenGL is given in Appendix B-4.



Light 1:	ON	OFF		
Ambient:	0.5	0.5	0.5	1.0
Diffuse:	1.0	1.0	1.0	1.0
Specular:	1.0	1.0	1.0	1.0
Position:	60.0	40.0	0.0	0.0

Light 2:	ON	OFF		
Ambient:	1.0	0.0	0.0	1.0
Diffuse:	1.0	1.0	1.0	1.0
Specular:	1.0	1.0	1.0	1.0
Position:	-60.0	0.0	0.0	1.0

Light 3:	ON	OFF		
Ambient:	0.0	1.0	0.0	1.0
Diffuse:	1.0	1.0	1.0	1.0
Specular:	1.0	1.0	1.0	1.0
Position:	0.0	0.0	60.0	1.0

Light 4:	ON	OFF		
Ambient:	0.0	1.0	0.0	1.0
Diffuse:	1.0	1.0	1.0	1.0
Specular:	1.0	1.0	1.0	1.0
Position:	0.0	0.0	-60.0	1.0

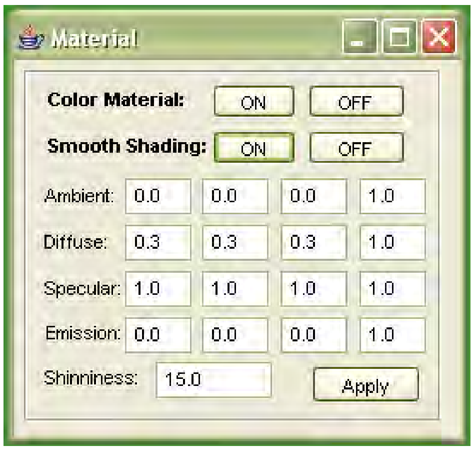
Camera Light:	ON	OFF		
Ambient:	100.0	10.0	10.0	1.0
Diffuse:	1.0	1.0	1.0	1.0
Specular:	1.0	1.0	1.0	1.0

Figure A.10 Lighting GUI

Lighting GUI, seen on the Figure A.10 gives an interface to the user for creating 4 different *positional lights* with user defined parameters, and spotlight that always travels with the camera.

3.3 Material

This section has been implemented with the purpose of learning OpenGL material capabilities. Information about material properties for the geometric primitives in OpenGL is given in Appendix B-5.



Color Material:	ON	OFF		
Smooth Shading:	ON	OFF		
Ambient:	0.0	0.0	0.0	1.0
Diffuse:	0.3	0.3	0.3	1.0
Specular:	1.0	1.0	1.0	1.0
Emission:	0.0	0.0	0.0	1.0
Shinniness:	15.0			

Figure A.11 Material GUI

Material GUI enables the user to assign material properties to the geometric primitives, which were created for testing and learning purposes. User is able to set different parameters for *glMaterial* command. Enabling *Smooth Shading* switches the shading state from *Flat Shading* and *Smooth Shading.* Smooth shading provides a *Gouraud* shading mode defined in the Shading section.

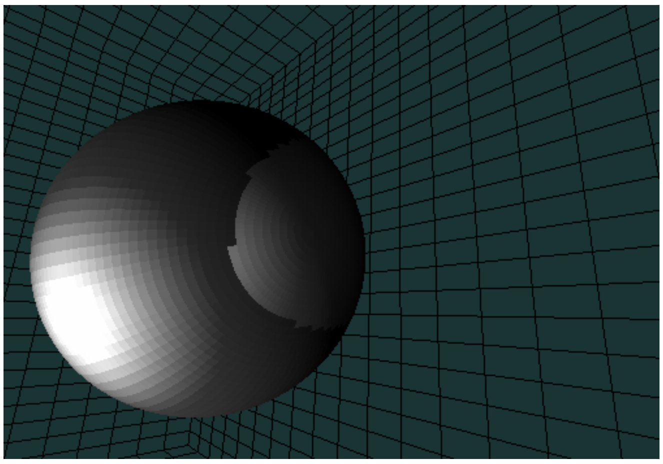


Figure A.12 Flat Shading

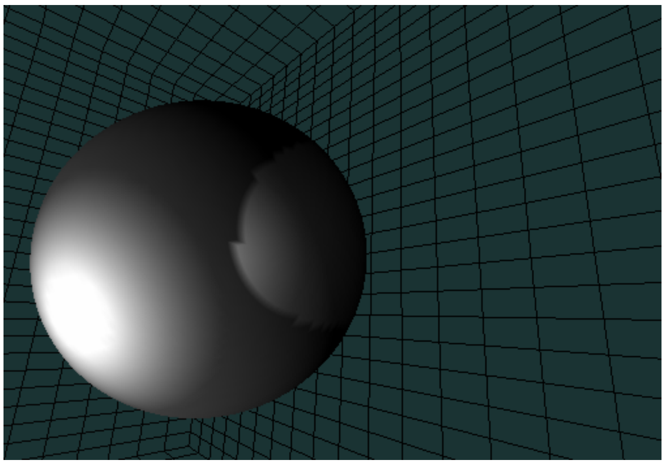


Figure A.13 Smooth Shading

Figures A.12 and A.13 have been rendered with a directional light and a spotlight placed on the camera.

4. How No Animator mode works?

declarations:

```
private class AnimatorTimerTask extends TimerTask {
public void run() {
        if (ANIMATED)
        {
          //myCanvas.requestFocus();
          //myCanvas.getRenderingThread().run();
          myCanvas.display();
        }
      }
  }
private static final int DEFAULT_ANIMATION_DELAY = 1000/24;
```

this line provides a frame rate of 24fps, if animation delay value set to 1000/30 the frame rate shall be 30fps.

```
private java.util.Timer animationTimer;
private TimerTask animationTimerTask = new
AnimatorTimerTask();
```

create:

```
if (animation is requested)
```

```
{
animationTimer = new java.util.Timer();

      animationTimer.scheduleAtFixedRate(animationTimerTask,  0,
      delay);
 }
 else
 {
      if (animationTimer != null)
      animationTimer.cancel();
 }
```

APPENDIX B 3D Computer Graphics and OpenGL

1. Fundamentals of 3D Computer Graphics

1.1 Overview

Computers became more and more powerful tools for producing picture in a fast and economical way. There is almost no area of interest that computer graphics cannot be used as a benefit, which explains why use of computer graphics is so widespread and popular. The early use of computer graphics in science and engineering had to rely on some expensive equipment. Due to fast evolution of computer and computer graphics technologies, it is used in diverse areas like, engineering, science, education, training, advertising, medicine, business, industry, government, art, and entertainment. Especially, the wide market of gaming and entertainment technologies caused a vast amount of production of graphics hardware, and a fast evolution of the performance and implementation techniques for computer visualization. [35].

1.2 APIs for Computer Graphics

Application Programming Interfaces (API) for computer graphics offer interfaces for the software developers to render three dimensional scenes with vast amount of functions using the capabilities of computer graphics hardware. OpenGL [72] developed by Silicon Graphics, and DirectX developed by Microsoft are most popular APIs developed for hardware accelerated computer graphics software developers. There are many libraries or graphics engines which give higher level of interfaces to the software developers. Most of such libraries are based on OpenGL or DirectX APIs. Java3d [58] is an example which presents an object oriented interface that use OpenGL or DirectX as interface for the graphics hardware.

In the implementation of hardware accelerated volume rendering that was done for this work was built on OpenGL using JOGL API [61] which was built to give interface to the OpenGL commands directly in the java development environment.

DirectX and OpenGL are the industry standard APIs, which define two dimensional and three dimensional graphics pipelines in which the graphics datasets or the geometric objects are processed and rendered. The viewing of an object is controlled using the model, the viewer position, and the projection matrices.

1.3 Spatial Transformation and Linear Algebra

The term, rendering, can be defined as, conversion of graphics primitives into an image. In other words, it is like drawing picture of a model onto a canvas. Some mathematical fundamentals are required to be understood for having an idea of how a graphics rendering process is done.

In computer graphics, primitives for a three dimensional object that is modeled, is represented in a Cartesian coordinate space with points having an address of x, y and z locations, lines with multiple points, polygons with multiple lines, surface with multiple polygons, and so fort. In order to view an object in different angles, transforming these graphical primitives is a useful method. In order to transform an object in the Cartesian coordinate space, each coordinate can be transformed by a linear equation.

$$\begin{aligned} x' &= ax + ey + iz + m \\ y' &= bx + fy + jz + n \\ z' &= cx + gy + kz + o \end{aligned}$$

Equation (B.1) [6, pp. 36]

In this equation, *x', y'* and *z'* are the transformed location of a point that is at the point *x, y* and *z,* where the coefficients *a - o* describe the relationship between them. Linear algebra allows representing such system of equations in matrices, makes it simpler to represent the relations and gives ability to define more complex spatial transformations.

A relationship of transformation between geometric primitives given in the Equation (B.1) can be defined in a matrix representation like this:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} a & e & i & m \\ b & f & j & n \\ c & g & k & o \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Equation (B.2) [6, pp. 37]

If a point that is to be transformed is denoted by *P* matrix and the new location of this point's matrix after the transformation is denoted *P'*:

$$P' = T \times P$$

where *T* is called the transformation matrix. Each elements of transformation matrix is for computing a specific transformation operation. Many different kinds of spatial transformations can be encoded into a transformation matrix.

Translation:

Translation is moving a point to another location. This operation can be achieved by adding the amount of translation on each axis to the original values of the point. Translation operation can be computed by a transformation matrix like this:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

where *tx, ty* and *tz* represent the amount of translation at each axis. Here a 4x4 matrix has been used which is different from Equation (B.2). Actually it is not necessary to use the bottom line of this matrix for the translation operation, however, this part is necessary for computing multiple transformation operations. It is seen that when there is no translation in any axis, in other words, when *tx, ty* and *tz* are all 0, an identity matrix is obtained where no change occurs when the matrix multiplication is done.

Scaling:

Scaling operation is used for changing the sizes of the objects. The transformation matrix for scaling is:

$$\begin{bmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where *sx, sy* and *sz* denotes the amount of scaling along each axis. When value 1 is used for the scaling parameters, an identity matrix is obtained and no change occurs in the output. If the value for scaling factor is grater than 1 the output model will become greater in size and if scaling factor is between 0 and 1, the output model will be smaller in size than the original. If the scaling factor for each axis is assigned as -1 a mirror affect is obtained. Different axis may have different values for scaling factor. This operation is useful in volume rendering. Especially in medical imaging, volumetric datasets may have different scaling factors for z axis. This operation is needed for balancing the volume dimensions.

Rotation:

Rotating an object in a three-dimensional space about an arbitrary axis can be achieved by multiplying the object coordinates by rotation transformation matrix. Transformation matrix for rotating about the x, y and z axes are given in the following figures.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

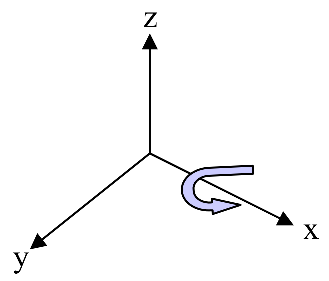


Figure B.1 Transformation matrix for rotation about x axis

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

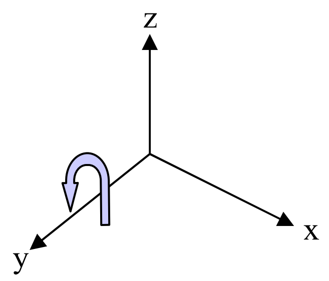


Figure B.2 Transformation matrix for rotation about y axis

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

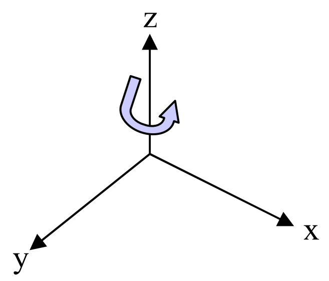


Figure B.3 Transformation matrix for rotation about z axis

The rotation transformation matrixes given in Figures B.1, B.2, B.3 rotate the objects along the axes x, y and z respectively, by counter clock wise direction as shown in the figures. In order to rotate the objects along an arbitrary axis other than the major axes x, y and z; the model can be translated so that, the rotation axis lies along one of the axes x, y and z; and apply the rotation transformation matrix, then place back the rotated object to the original position by my making inverse operations for the transformations made at the beginning.

Matrix operations give a power for computing because several different kinds of transformation operations can be combined into a single multiplication matrix which is called *compound transformation matrix*. So, instead of more than one transformation to the model as described in the preceding paragraph, a compound transformation matrix can be built first, and then applied to the whole dataset at once. For example, a matrix denoted by R, rotates the model along x axis; and another matrix denoted by T, translates the model with 10 units along the y axis. If the matrix multiplication is done between T and R matrixes, like:

$M1 = R \times T$

M1 is a new transformation matrix, which first translates the model by 10 units along the y axis, and rotates it by the x axis. However, if the multiplication is done in another way:

$M2 = T \times R$

The transformation matrix M2 shall first rotate the model about the x axis, then translate it along the y axis. So, M1 is not equal to M2 and the matrix multiplications are applied from right to left. [6, pp. 44]

1.4 Rendering

Having the knowledge of transformation, volumetric data can be scaled, rotated or translated by building a transformation matrix. In order to see what is happening on the model we are transforming, the model has to be painted on a two dimensional image consisting of pixels representing the projection of the voxels or the geometric primitives like point, line or surface.

Rendering is the process of projecting the transformed geometric primitives onto an image plane. Main function of the graphics APIs like OpenGL is rendering. OpenGL gives an interface for rendering geometric primitives with two methods. *Intermediate mode* [29, pp. 195] executes the OpenGL commands immediately and renders the scene. *Display list mode* enables the developer to store some commands for rendering for a later execution, which gives a better performance.

Real time rendering is the process of sequential rendering and produces an animation with the rendered images. In order to establish a high sequence and obtain a smooth animation, the rendering time required for producing one image is very important. APIs like OpenGL uses hardware graphics units to increase the performance of rendering, and establish a real time animation. *Frame per second* is the metric used for measuring the performance of real time rendering. What is the minimum limit for real time rendering is a relative question that changes from one type of user to another [36]. While watching a movie with a rate of 24 frames per second, it may be ok but a person who plays flight simulator or an arcade game would require 60-100 frames per second render results for a satisfactory smoothness in the animation. “Modern people are trained in picking up fast cuts, around 10 frames per second is probably the limit.” [36, pp. 1]

2. OpenGL

OpenGL is an environment for developing interactive two and three dimensional graphics applications. OpenGL, which was created by Silicon Graphics [73], in 1992, is now one of the most widely used and supported application programming interface (API) in the computer graphics industry [72]. GL stands for *graphics library*. OpenGL is a hardware independent API, which gives a software interface to the graphics hardware.

2.1 Overview

OpenGL consists of more than 150 distinct commands which are used for specifying geometric objects and required operation for producing interactive three dimensional applications. It is a hardware independent interface which can be implemented on many different hardware platforms. In order to achieve this property, OpenGL does not include commands for performing windowing tasks or obtaining user input. It does not provide high level commands for describing three dimensional objects models. Instead, the models for complicated objects are created by using geometric primitives which are points, lines and polygons. Some sophisticated libraries, which provide high level commands to define complicated models, can be built on top of OpenGL. The OpenGL Utility Library (GLU) is an example for that, which provides many modeling features. [29]

OpenGL presents a *procedural* interface rather than a *descriptive* interface [74]. In a system having a descriptive interface (VRML is an example for such systems [75]) for example, the user can create a blue sphere at a certain place. However, in OpenGL, in order to render such an object, the developer has to follow a sequence of commands, set up the camera view, transform the model, draw the geometric primitives for the blue ball and so fort. Specifying all the required commands, operation in an appropriate order in an exact detail is a disadvantage for the procedural interfaces. However, this gives a great flexibility while rendering a scene, because every step for rendering is applied and the programmer has the ability to modify each step and control rendering speed, image quality. A descriptive interface can be built on top of a procedural interface but the opposite cannot be the case, and this explains the power of a procedural interface over a descriptive one. Some basic steps required for writing an OpenGL application is as follows:

- OpenGL is not a model builder API but a renderer. This means that the model should be built by the user. There are some libraries like OpenGL Utility Library (GLU) which establish a descriptive interface for practical ways for building models. The quality of rendering is affected by the model detail quality. The primitives of a model for OpenGL are points, lines, polygons, images and bitmaps.

- When the model to be rendered is ready, the objects are placed at a desired location by using transformation commands, and the desired position for viewing the composed scene is selected.

- Color calculations, are done for all objects.

- Rasterization which is the process of projecting the geometric model to the pixels on the screen is done.

2.2 Syntax

Prefix “gl” is used for OpenGL commands, and the command name starts with capital letter like: “glClearColor()”. The predefined constants use the prefix “GL_” and all letters are used as capital like: “GL_POLYGON”.

Suffixes used in the commands define number and type of the variables that will be used. For example, “glColor3f()” command indicates that three arguments are given by the letter “3”, and the letter “f” indicates that the arguments are floating point variables.

Some of the OpenGL commands accept eight different data types as shown in the Table B.1.

Suffix	**Data Type**	**Typical Corresponding C-Language Type**	**OpenGL Type Definition**
b	8-bit integer	signed char	GLbyte
s	16-bit integer	short	GLshort
i	32-bit integer	int or long	GLint, GLsizei
f	32-bit floating-point	float	GLfloat, GLclampf
d	64-bit floating-point	double	GLdouble, GLclampd
ub	8-bit unsigned integer	unsigned char	GLubyte, GLboolean
us	16-bit unsigned	unsigned short	GLushort

Suffix	Data Type	Typical Corresponding C-Language Type	OpenGL Type Definition
	integer		
ui	32-bit unsigned integer	unsigned int or unsigned long	GLuint, GLenum, GLbitfield

Table B 1 OpenGL Data Types [29, pp.17]

OpenGL commands work as a state machine, which means when a mode is set for a property, the remaining part implementation will apply always that mode for that property. For example, current color can be assigned as white by the command glColor3f(1.0, 1.0, 1.0). This means that, after the line of this command all the objects drawn shall be painted in color white until current color state is changed by another "glColor" command. Many of the states need to be enabled or disabled by using the commands "glEnable()" and "glDisable()". Each state variable are assigned to default values and those values that are used currently, can be queried an any place of the system.

2.2.1 State Management

In *display list* styles of rendering, while drawing objects, a list of commands to be done is prepared and the graphics interface library expects a command to execute that list. OpenGL supports this style and enables the programmer to build many lists of commands and execute these groups of commands at any point in the system. However, OpenGL default style works in *immediate mode*, which executes immediately the command at every time and every place issued by the programmer.

For each frame to be rendered, the window should be cleared entirely with a background color that is specified by the user. The command set for clearing an entire window to color black is as follows. [29]

```
glClearColor(0.0, 0.0, 0.0, 0.0);
glClear(GL_CLEAR_BUFFER_BIT);
```

In “glClearColor” command, the color values of red, green, blue and alpha determines the color to be used for clearing the window, and GL_CLEAR_BUFFER_BIT is an enumerated mask name that represents “Color Buffer” to be cleared by the “glClear” command.

The following pseudo code shows how coloring mode is managed with OpenGL:

```
set current color(blue);
draw object(X);
set current color(red);
set current color(green);
draw object(Y);
draw object(Z);
```

Result for this implementation will result like this: X drawn in blue, Y and Z drawn in green [29].

2.2.2 Geometric Primitives

In OpenGL all geometric primitives are described in terms of their *vertices* which are the points that store the coordinates of end points of a line segment or corner of a polygon.

OpenGL definition for point line, and polygon have the similar meaning of the same names used in mathematics, but not quite the same. The difference comes from the limitations of computer computations. For example, floating point calculations have finite precision so the results have some round off errors. The coordinates used for the modeling in OpenGL suffer from this situation. Another difference is occurred because of the result of rendering is a raster graphic display. A pixel has a specific size and in some views as it does not have an infinitively small size, more than one object might happen to be rendered in a single pixel.

A point is a single vertex consisting of three dimensional coordinates stored as floating point numbers. Every internal calculation done by OpenGL assumes the vertices are three dimensional. If a point has been assigned as two dimensional, the z coordinate is assumed as zero by OpenGL.

The term *line* is not understood like the mathematician's version that it goes to infinity in both directions, but refers to a line segment.

Polygon is an area that is enclosed single closed loops of line segments which are specified by the vertices of their end points. OpenGL has some restriction rules for constructing a polygon.

2.3. Blending In OpenGL

This section is taken from the following references, and some part of it is modified for the summarization of the subject.

References:
1. http://www.opengl.org/resources/tutorials/sig99/advanced99/notes/node1.html
2. Woo M., Neider J., Davis T., Shreiner D., OpenGL Architecture Review Board, (1997). <u>OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL, Version 1.1 (2nd Edition)</u>. Silicon Graphics, Inc
3. http://tfpsly.planet-d.net/OpenGL/Faq/

OpenGL doesn't support a direct interface for rendering translucent primitives. However, a transparency effect with the blend feature can be created. An OpenGL application enables blending function as follows:

```
glEnable (GL_BLEND);
glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

After blending is enabled, the incoming primitive color is blended with the color already stored in the frame buffer. *glBlendFunc* controls how this blending occurs. The typical use given in the above example, modifies the incoming color by its associated alpha value and modifies the destination color by one minus the incoming alpha value. The sum of these two colors is then written back into the frame buffer.

While using the depth buffering in an application, the order of the primitives to be rendered is important. Fully opaque primitives need to be rendered first, followed by partially opaque primitives in *back to front* order.

OpenGL provides a rich set of blending operations which can be used to implement transparency, compositing, painting, and other effects. The glBlendFunc() command selects the source and destination blend factors. The most frequently used factors are GL_ZERO, GL_ONE, GL_SRC_ALPHA and GL_ONE_MINUS_SRC_ALPHA.

Alpha values are specified with *glColor*, when using *glClearColor* to specify a clearing color and when specifying certain lighting parameters such as a material property or light-source intensity. The pixels on a monitor screen emit red, green, and blue light, which is controlled by the red, green, and blue color values. So how does an alpha value affect what gets drawn in a window on the screen?

When blending is enabled, the alpha value is used to combine the color value of the fragment being processed with that of the pixel already stored in the framebuffer. Blending occurs after your scene has been rasterized and converted to fragments, but just before the final pixels are drawn in the frame buffer.

Without blending, each new fragment overwrites any existing color values in the framebuffer, as though the fragment were opaque. With blending, how and how much of the existing color value should be combined with the new fragment's value can be controlled. So alpha blending can be used for creating a translucent fragment that lets some of the previously stored color value.

The most natural way to think of blending operations is to think of the RGB components of a fragment as representing its color and the alpha component as representing opacity. Transparent or translucent surfaces have lower opacity than opaque ones and, therefore, lower alpha values. For example, if you're viewing an object through green glass, the color you see is partly green from the glass and partly the color of the object. The percentage varies depending on the transmission properties of the glass: If the glass transmits 80 percent of the light that strikes it (that is, has an opacity of 20 percent), the color you see is a combination of 20 percent glass color and 80 percent of the color of the object behind it. You can easily imagine situations with multiple translucent surfaces.

During blending, color values of the incoming fragment, the *source,* are combined with the color values of the corresponding currently stored pixel, the *destination,* in a two-stage process. First you specify how to compute source and destination factors. These factors are RGBA quadruplets that are multiplied by each component of the R, G, B, and A values in the source and destination, respectively. Then the corresponding components in the two sets of RGBA quadruplets are added. Let the source and destination blending factors be (Sr, Sg, Sb, Sa) and (Dr, Dg, Db, Da), respectively, and the RGBA values of the source and destination be indicated with a subscript of s or d. Then the final, blended RGBA values are given by:

(RsSr+RdDr, GsSg+GdDg, BsSb+BdDb, AsSa+AdDa)

glBlendFunc() is used for supplying two constants: one that specifies how the source factor should be computed and one that indicates how the destination factor should be computed.

glBlendFunc(*sfactor*, *dfactor*)

In the following table the RGBA values of the source and destination are indicated with the subscripts s and d, respectively. Subtraction of quadruplets means subtracting them component wise. The Relevant Factor column indicates whether the corresponding constant can be used to specify the source or destination blend factor.

Constant	**Relevant Factor**	**Computed Blend Factor**
GL_ZERO	source or destination	(0, 0, 0, 0)
GL_ONE	source or destination	(1, 1, 1, 1)
GL_DST_COLOR	source	(R_d, G_d, B_d, A_d)
GL_SRC_COLOR	destination	(R_s, G_s, B_s, A_s)
GL_ONE_MINUS_DST_COLOR	source	(1, 1, 1, 1)-(R_d, G_d, B_d, A_d)
GL_ONE_MINUS_SRC_COLOR	destination	(1, 1, 1, 1)-(R_s, G_s, B_s, A_s)
GL_SRC_ALPHA	source or destination	(A_s, A_s, A_s, A_s)
GL_ONE_MINUS_SRC_ALPHA	source or destination	(1, 1, 1, 1)-(A_s, A_s, A_s, A_s)
GL_DST_ALPHA	source or destination	(A_d, A_d, A_d, A_d)
GL_ONE_MINUS_DST_ALPHA	source or destination	(1, 1, 1, 1)-(A_d, A_d, A_d, A_d)
GL_SRC_ALPHA_SATURATE	source	(f, f, f, 1); f=min(As, 1-Ad)

Source and Destination Blending Factors

Not all combinations of source and destination factors make sense. Most applications use a small number of combinations. The following part describes typical uses for particular combinations of source and destination factors.

- One way to draw a picture composed half of one image and half of another, equally blended, is to set the source factor to GL_ONE and the destination factor to GL_ZERO, and draw the first image. Then set the source factor to GL_SRC_ALPHA and destination factor to GL_ONE_MINUS_SRC_ALPHA, and draw the second image with alpha equal to 0.5. This pair of factors probably represents the most commonly used blending operation. If the picture is supposed to be blended with 0.75 of the first image and 0.25 of the second, the first image is drawn as before, and the second is drawn with an alpha of 0.25.

- If new images are to be added onto another image and what ever the last result, the newly added image will have %90 transparency, this can be done with blending as follows: image is drawn with alpha of 10 percent and GL_SRC_ALPHA (source) and GL_ONE_MINUS_SRC_ALPHA (destination) is used in the blending function..

- The blending functions that use the source or destination colors GL_DST_COLOR or GL_ONE_MINUS_DST_COLOR for the source factor and GL_SRC_COLOR or GL_ONE_MINUS_SRC_COLOR for the destination factor, each color component can be effectively modulated individually. This operation is equivalent to applying a simple filter, for example, multiplying the red component by 80 percent, the green component by 40 percent, and the blue component by 72 percent would simulate viewing the scene through a photographic filter that blocks 20 percent of red light, 60 percent of green, and 28 percent of blue.

2.4 Lighting in OpenGL

This section is taken from the following reference, and some part of it is modified for the summarization of the subject.

Reference:
Woo M., Neider J., Davis T., Shreiner D., OpenGL Architecture Review Board, (1997). OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL, Version 1.1 (2nd Edition). Silicon Graphics, Inc

Creating Light Sources:

Light sources have a number of properties, such as color, position, and direction. The following sections explain how to control these properties in OpenGL and what the resulting light looks like. The command used to specify all properties of lights is `glLight;` it takes three arguments: to identify the light whose property is being specified, the property, and the desired value for that property.

"`void glLight(GLenum` *`light`*`, GLenum` *`pname`*`,` *`TYPE param`*`)`" Creates the light specified by *light*, which can be GL_LIGHT0, GL_LIGHT1, ... , or GL_LIGHT7. The characteristic of the light being set is defined by *pname*, which specifies a named parameter. *param* indicates the values to which the *pname* characteristic is set; it's a pointer to a group of values if the vector version is used, or the value itself if the nonvector version is used. The nonvector version can be used to set only single-valued light characteristics.

Parameter Name	**Default Value**	**Meaning**
GL_AMBIENT	(0.0, 0.0, 0.0, 1.0)	ambient RGBA intensity of light
GL_DIFFUSE	(1.0, 1.0, 1.0, 1.0)	diffuse RGBA intensity of light
GL_SPECULAR	(1.0, 1.0, 1.0, 1.0)	specular RGBA intensity of light
GL_POSITION	(0.0, 0.0, 1.0, 0.0)	(*x, y, z, w*) position of light
GL_SPOT_DIRECTION	(0.0, 0.0, -1.0)	(*x, y, z*) direction of spotlight
GL_SPOT_EXPONENT	0.0	spotlight exponent
GL_SPOT_CUTOFF	180.0	spotlight cutoff angle
GL_CONSTANT_ATTENUATION	1.0	constant attenuation factor
GL_LINEAR_ATTENUATION	0.0	linear attenuation factor
GL_QUADRATIC_ATTENUATION	0.0	quadratic attenuation factor

Default Values for pname Parameter of `glLight()`

Position and Attenuation:

A light source can be chosen as it is located infinitely far away from the scene or as it is nearer to the scene. The first type is referred to as a *directional* light source; the effect of an infinite location is that the rays of light can be considered parallel by the time they reach an object. An example of a real-world directional light source is the sun. The second type is called a *positional* light source, since its exact position within the scene determines the effect it has on a scene and, specifically, the direction from which the light rays come. A desk lamp is an example of a positional light source.

```
GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

As shown, a vector of four values (*x, y, z, w*) for the GL_POSITION parameter is supplied. If the last value, *w*, is zero, the corresponding light source is a directional one, and the (*x, y, z*) values describe its direction. This direction is transformed by the modelview matrix. By default, GL_POSITION is (0, 0, 1, 0), which defines a directional light that points along the negative *z*-axis.

If the *w* value is nonzero, the light is positional, and the (*x, y, z*) values specify the location of the light in homogeneous object coordinates. This location is transformed by the modelview matrix and stored in eye coordinates. Also, by default, a positional light radiates in all directions, but it can be restricted to producing a cone of illumination by defining the light as a spotlight.

For real-world lights, the intensity of light decreases as distance from the light increases. Since a directional light is infinitely far away, it doesn't make sense to attenuate its intensity over distance, so attenuation is disabled for a directional light. OpenGL attenuates a light source by multiplying the contribution of that source by an attenuation factor:

$$\text{attenuation factor} = \frac{1}{k_c + k_l d + k_q d^2}$$

where

d = distance between the light's position and the vertex

k_c = GL_CONSTANT_ATTENUATION

k_l = GL_LINEAR_ATTENUATION

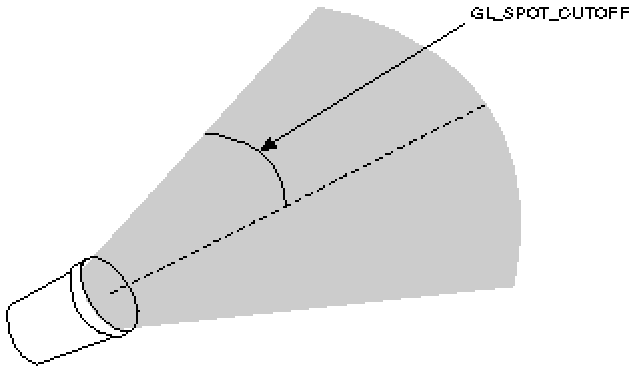
k_q = GL_QUADRATIC_ATTENUATION

By default, k_c is 1.0 and both k_l and k_q are zero, but you can give these parameters different values:

```
glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 2.0);
glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 1.0);
glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0.5);
```

Spotlights:

A positional light source can act as a spotlight—that is, by restricting the shape of the light it emits to a cone. To create a spotlight, the spread of the cone of light needed to be determined. To specify the angle between the axis of the cone and a ray along the edge of the cone, GL_SPOT_CUTOFF parameter is used. The angle of the cone at the apex is then twice this value, as shown in the following figure.



GL_SPOT_CUTOFF Parameter

By default, the spotlight feature is disabled because the GL_SPOT_CUTOFF parameter is 180.0. This value means that light is emitted in all directions (the angle at the cone's apex is 360 degrees, so it isn't a cone at all). The value for GL_SPOT_CUTOFF is restricted to being within the range [0.0,90.0] (unless it has the special value 180.0). The following line sets the cutoff parameter to 45 degrees:

```
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 45.0);
```

In order to specify a spotlight's direction, which determines the axis of the cone of light foolowing code is used :

```
GLfloat spot_direction[] = { -1.0, -1.0, 0.0 };
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, spot_direction);
```

2.5 Material Properties in OpenGL

This section is taken from the following reference, and some part of it is modified for the summarization of the subject.

Reference:
Woo M., Neider J., Davis T., Shreiner D., OpenGL Architecture Review Board, (1997). OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL, Version 1.1 (2nd Edition). Silicon Graphics, Inc

Defining Material Properties:

This section describes how to define the material properties of the objects in the scene: the ambient, diffuse, and specular colors, the shininess, and the color of any emitted light in OpenGL. Most of the material properties are conceptually similar to ones used to create light sources. The mechanism for setting them is similar, except that the command used is called `glMaterial()`.

`"void glMaterial(GLenum face, GLenum pname, TYPE param)"` specifies a current material property for use in lighting calculations. face can be GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK to indicate which face of the object the material should be applied to. The particular material property being set is identified by pname and the desired values for that property are given by param, which is either a pointer to a group of values (if the vector version is used) or the actual value (if the nonvector version is used). The nonvector version works only for setting GL_SHININESS. The possible values for pname are shown in the following table.

Parameter Name	**Default Value**	**Meaning**
GL_AMBIENT	(0.2, 0.2, 0.2, 1.0)	ambient color of material
GL_DIFFUSE	(0.8, 0.8, 0.8, 1.0)	diffuse color of material
GL_AMBIENT_AND_DIFFUSE		ambient and diffuse color of material
GL_SPECULAR	(0.0, 0.0, 0.0, 1.0)	specular color of material
GL_SHININESS	0.0	specular exponent
GL_EMISSION	(0.0, 0.0, 0.0, 1.0)	emissive color of material
GL_COLOR_INDEXES	(0,1,1)	ambient, diffuse, and specular color indices

Default Values for pname Parameter of `glMaterial`

Diffuse and Ambient Reflection:

The GL_DIFFUSE and GL_AMBIENT parameters set with `glMaterial` affect the color of the diffuse and ambient light reflected by an object. Diffuse reflectance plays the most important role in determining what you perceive the color of an object to be. It's affected by the color of the incident diffuse light and the angle of the incident light relative to the normal direction. The position of the viewpoint doesn't affect diffuse reflectance at all.

Ambient reflectance affects the overall color of the object. Because diffuse reflectance is brightest where an object is directly illuminated, ambient reflectance is most noticeable where an object receives no direct illumination. An object's total ambient reflectance is affected by the global ambient light and ambient light from individual light sources. Like diffuse reflectance, ambient reflectance isn't affected by the position of the viewpoint.

For real-world objects, diffuse and ambient reflectance are normally the same color. For this reason, OpenGL provides you with a convenient way of assigning the same value to both simultaneously with `glMaterial`:

```
GLfloat mat_amb_diff[] = { 0.1, 0.5, 0.8, 1.0 };
glMaterialfv(GL_FRONT_AND_BACK,       GL_AMBIENT_AND_DIFFUSE,
mat_amb_diff);
```

In this example, the RGBA color (0.1, 0.5, 0.8, 1.0), a deep blue color, represents the current ambient and diffuse reflectance for both the front- and back-facing polygons.

Specular Reflection:

Specular reflection from an object produces highlights. Unlike ambient and diffuse reflection, the amount of specular reflection seen by a viewer does depend on the location of the viewpoint, it's brightest along the direct angle of reflection. To see this, imagine looking at a metallic ball outdoors in the sunlight.

OpenGL allows to set the effect that the material has on reflected light (with GL_SPECULAR) and control the size and brightness of the highlight (with GL_SHININESS). GL_SPECULAR and GL_SHININESS are assigned values as follows:

```
GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat low_shininess[] = { 5.0 };
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_SHININESS, low_shininess);
```

APPENDIX C Bump Mapping Source Code

1. Dot3 Bump Mapping with Multi Texturing Source Code

```
gl.glActiveTexture(gl.GL_TEXTURE0_ARB);
gl.glBindTexture (gl.GL_TEXTURE_3D, volTexture3d_ID_gradient);
```

first the gradient texture is binded.

```
gl.glEnable(gl.GL_TEXTURE_3D);
gl.glTexEnvf(gl.GL_TEXTURE_ENV, gl.GL_TEXTURE_ENV_MODE,
gl.GL_COMBINE_EXT);
gl.glTexEnvf(gl.GL_TEXTURE_ENV, gl.GL_COMBINE_RGB_EXT,
gl.GL_DOT3_RGB_EXT);

gl.glTexEnvf(gl.GL_TEXTURE_ENV, gl.GL_SOURCE0_RGB_EXT,
gl.GL_TEXTURE);
gl.glTexEnvf(gl.GL_TEXTURE_ENV, gl.GL_OPERAND0_RGB_EXT,
gl.GL_SRC_COLOR);
gl.glTexEnvf(gl.GL_TEXTURE_ENV, gl.GL_SOURCE1_RGB_EXT,
gl.GL_PRIMARY_COLOR_EXT);
gl.glTexEnvf(gl.GL_TEXTURE_ENV, gl.GL_OPERAND1_RGB_EXT,
gl.GL_SRC_COLOR);
```

Until now, the dot product of each pixel with the polygon color has been computed. Below part binds it with the original volumetric texture multiplying with the computed pixel result.

```
gl.glActiveTexture(gl.GL_TEXTURE1_ARB);
gl.glBindTexture (gl.GL_TEXTURE_3D,
volTexture3d_ID_originalTexture);
gl.glEnable(gl.GL_TEXTURE_3D);
gl.glTexEnvf(gl.GL_TEXTURE_ENV, gl.GL_TEXTURE_ENV_MODE,
gl.GL_COMBINE_EXT);
gl.glTexEnvf(gl.GL_TEXTURE_ENV, gl.GL_COMBINE_RGB_EXT,
gl.GL_MODULATE);

gl.glTexEnvf(gl.GL_TEXTURE_ENV, gl.GL_SOURCE0_RGB_EXT,
gl.GL_PREVIOUS_EXT);
gl.glTexEnvf(gl.GL_TEXTURE_ENV, gl.GL_OPERAND0_RGB_EXT,
gl.GL_SRC_COLOR);
gl.glTexEnvf(gl.GL_TEXTURE_ENV, gl.GL_SOURCE1_RGB_EXT,
gl.GL_TEXTURE);
gl.glTexEnvf(gl.GL_TEXTURE_ENV, gl.GL_OPERAND1_RGB_EXT,
gl.GL_SRC_COLOR);
```

2. Added Bump Mapping Mode, Source Code

```
gl.glActiveTexture(gl.GL_TEXTURE0_ARB);
gl.glBindTexture (gl.GL_TEXTURE_3D, volTexture3d_ID_gradient);
gl.glEnable(gl.GL_TEXTURE_3D);
gl.glTexEnvf(gl.GL_TEXTURE_ENV, gl.GL_TEXTURE_ENV_MODE,
gl.GL_COMBINE_EXT);
gl.glTexEnvf(gl.GL_TEXTURE_ENV, gl.GL_COMBINE_RGB_EXT,
gl.GL_DOT3_RGB_EXT);

gl.glTexEnvf(gl.GL_TEXTURE_ENV, gl.GL_SOURCE0_RGB_EXT,
gl.GL_TEXTURE);
gl.glTexEnvf(gl.GL_TEXTURE_ENV, gl.GL_OPERAND0_RGB_EXT,
gl.GL_SRC_COLOR);
gl.glTexEnvf(gl.GL_TEXTURE_ENV, gl.GL_SOURCE1_RGB_EXT,
gl.GL_PRIMARY_COLOR_EXT);
gl.glTexEnvf(gl.GL_TEXTURE_ENV, gl.GL_OPERAND1_RGB_EXT,
gl.GL_SRC_COLOR);

gl.glActiveTexture(gl.GL_TEXTURE1_ARB);
gl.glBindTexture (gl.GL_TEXTURE_3D, volTexture3d_ID);
gl.glEnable(gl.GL_TEXTURE_3D);
gl.glTexEnvf(gl.GL_TEXTURE_ENV, gl.GL_TEXTURE_ENV_MODE,
gl.GL_COMBINE_EXT);
gl.glTexEnvf(gl.GL_TEXTURE_ENV, gl.GL_COMBINE_RGB_EXT, gl.GL_ADD);

gl.glTexEnvf(gl.GL_TEXTURE_ENV, gl.GL_SOURCE0_RGB_EXT,
gl.GL_PREVIOUS_EXT);
gl.glTexEnvf(gl.GL_TEXTURE_ENV, gl.GL_OPERAND0_RGB_EXT,
gl.GL_SRC_COLOR);
gl.glTexEnvf(gl.GL_TEXTURE_ENV, gl.GL_SOURCE1_RGB_EXT,
gl.GL_TEXTURE);
gl.glTexEnvf(gl.GL_TEXTURE_ENV, gl.GL_OPERAND1_RGB_EXT,
gl.GL_SRC_COLOR);
```

APPENDIX D Biomedical Imaging Resource (BIR) and the *Analyze* Software

This section is taken directly from the following reference.

Reference:
http://www.mayo.edu/bir/Software/Analyze/Analyze1.html

The Biomedical Imaging Resource (BIR) at Mayo Clinic is dedicated to the advancement of research in the biomedical imaging and visualization sciences. The BIR provides expertise and advanced technology related to these fields, including image acquisition, processing, display and analysis; volume visualization; computer graphics; virtual reality and virtual environments; image databases; computer workstations, networks and programming.

The Biomedical Imaging Resource at the Mayo Foundation has been involved since the early 1970's in the design and implementation of computer-based techniques for the display and analysis of multidimensional biomedical images.

The algorithms and programs developed through this research have formed the basis for integrated, comprehensive software systems developed by the Biomedical Imaging Resource, useful in a variety of multimodality, multidimensional biomedical imaging and scientific visualization applications. These integrated suites of complementary tools for fully interactive display, manipulation, and measurement of multidimensional biomedical images have been used in applications involving many different imaging modalities, including CT, MRI, SPECT, PET, ultrasound and digital microscopy.

With the advent of advanced biomedical imaging techniques which are most efficiently realized via an integration of algorithms, such as segmentation driven by direct visualization, the availability of standardized interface software through powerful windowing systems, and the need to expediently address an ever-expanding variety of specific biomedical imaging applications, the Analyze software system has continued to mature into the most comprehensive, robust and productive software package available for 3D biomedical image visualization and analysis. It has served as the embodiment of the integrated biomedical imaging algorithms and tools developed in the BIR for over 15 years. Its widespread use and impact on a multiplicity of applications have served to validate the 'toolbox' approach to biomedical imaging software integration, an architecture which provides an effective shell for rapid prototyping of customized imaging applications. The synergistic integration of comprehensive and generic tools for visualization, processing, and quantitative analysis of biomedical images in a highly operator-interactive, intuitive interface has allowed surgeons, physicians, and basic scientists to explore large multidimensional biomedical image volumes efficiently and productively.

The Analyze software system is entirely built upon a toolkit of optimized functions that are organized into a software development library called AVW. The AVW imaging library is a collection of over 600 functions that are accessible to software developers to build advanced image-based application solutions. Analyze is an integration of the full functionality represented in the AVW toolkit with an intuitive windows-based interface which makes it easy to learn and to use. The most important feature of Analyze is the paradigm in which it operates - a powerful software architecture that allows multiple volume images to be simultaneously accessed and processed by multiple programs in a multi-window interface. The user interface for Analyze is based on Tcl/Tk, which offers full compliance with interface standards across multivendor workstations and PCs.

The development of the Analyze and AVW imaging software systems by the BIR software development staff directly benefits from a combined total of over 100 years of programming experience in the BIR, approximately 75 years of which have been with Analyze.