5-10-2003

Wavelet-Based Volume Rendering

Pujita Pinnamaneni

WAVELET-BASED VOLUME RENDERING

By

Pujita Pinnamaneni

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

Mississippi State, Mississippi

May 2003

WAVELET-BASED VOLUME RENDERING

By

Pujita Pinnamaneni

Approved:

Joerg Meyer Assistant Professor of Computer Science (Major Professor)	Thomas Philip Professor of Computer Science (Committee Member)
James C. Harden Professor of Computer Engineering (Committee Member, Graduate Coordinator)	A. Wayne Bennett Dean of the College of Engineering

Name: Pujita Pinnamaneni

Date of Degree: May 10, 2003

Institution: Mississippi State University

Major Field: Computer Engineering

Major Professor: Dr. Joerg Meyer

Title of Study: WAVELET-BASED VOLUME RENDERING

Pages in Study: 52

Candidate for Degree of Master of Science

Various biomedical technologies like CT, MRI and PET scanners provide detailed cross-sectional views of the human anatomy. The image information obtained from these scanning devices is typically represented as large data sets whose sizes vary from several hundred megabytes to about one hundred gigabytes. As these data sets cannot be stored on one's local hard drive, SDSC provides a large data repository to store such data sets. These data sets need to be accessed by researchers around the world to collaborate in their research. But the size of these data sets make them diffcult to be transmitted over the current network. This thesis presents a 3-D Haar wavelet algorithm which enables these data sets to be transformed into smaller hierarchical representations. These transformed data sets are transmitted over the network and reconstructed to a 3-D volume on the client's side through progressive refnement of the images and 3-D texture mapping techniques.

DEDICATION

I dedicate this thesis to my parents, grandparents, sister and to my niece.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

FIGURE Page

CHAPTER I

INTRODUCTION

1.1 Overview

Large-scale biomedical data sets enable physicians and biologists to obtain detailed insights in complex biomedical structures. Enhanced imaging techniques allow them to distinguish pathological from healthy tissue or to study microscopic cell structures in greater detail. But as the datasets obtained from medical imaging devices such as CT, MRI, and PET scanners range from a few megabytes to several gigabytes in size, the storage of these data sets on one's local hard drive can prove expensive. San Diego Supercomputer Center (SDSC) maintains a high peformance storage facility that enables researchers to store their data sets remotely, and enables authorized access to data for collaborators around the world. But another hurdle that needs to be crossed is the current transmission rate of the network. Transmitting large datasets over low/medium networks can prove time-consuming. If the user needs to wait for the entire data set to be downloaded onto a local system, then the system lacks interactivity and responsiveness. This thesis proposes a wavelet-based rendering system in which a lower-resolution representation of the data set is initially transmitted across a network to give the end user an instant preview of the data set, followed by progressive transmission of detail coeffcients that ensure that the data set

is eventually reconstructed to the fullest resolution. Texture mapping is used to render a series of 2-D slices into a 3-D volume on the client's machine.

Most graphics applications are written in C/C++ and OpenGL. Both standards have been developed to enable hardware-effcient implementations. However, programs using these standards typically lack hardware-independence. A new standard called Java was developed in the early 1990s at Sun Microsystems [1]. Java was designed to be platform independent by using a so-called Java virtual machine (JVM). Java3D is a graphics extension to Java and was introduced to enable programmers to develop graphical applications. The JVM that is used for the Java implementation is based on a higher-level language implementation which introduces an additional layer between the application and the application programming interface (API) and therefore causes additional overhead. Another feature that adds to the extra overhead in Java3D is the scenegraph concept [20]. OpenGL, on the other hand, is an API to the graphics hardware, and it typically does not make use of the scenegraph concept. These reasons suggest that Java3D rendering applications are typically much slower than OpenGL rendering applications. The idea of this thesis is to show that Java3D rendering applications can be made more effcient by using improved hierarchical data models and storage schemes, and that Java3D is capable of providing comparable results to OpenGL applications. The goal of this framework is to demonstrate that comparable performances can be obtained by using a set of new methods. These methods include Haar wavelet transformation, multi-resolution representations, and 3D-texture-based rendering techniques.

1.2 Motivation

Large-scale data sets can be grouped into two categories: mesh-based and non-mesh-based (scattered data). For our purposes, we use mesh-based biomedical data sets defned over a structured grid, also known as volume data sets. One of the project partners (San Diego Supercomputer Center, SDSC) developed a set of new fle formats to store different types of structured and unstructured meshes. The fle format that is used to store volume data sets is called the VOL fle format. The VOL format supports three different variants namely $vols, volb,$ and $volc$. If a data set contains 8-bit scalar values, it is encoded as a $.vols$ fle. If the data set contains 32-bit RGB or RGBA values, then it is encoded in the $.volb$ fle format. Each byte represents a color or transparency channel. If no alpha channel is present, then the alpha value is zero. 64-bit image data is written in the .volc format. The $.volc$ format stores image data as 64-bit RGB-alpha-beta values. The color component values are tructated to 10 bits of red, 12 bits for green and 10 bits for blue. The alpha and beta components are truncated to 16 bits each [19].

The $National\ Partnership\ for\ Advanced\ Computational\ Infrastructure$ (NPACI) has developed a set of tools that has been used to access datasets in VOL format, among other mesh formats. These so-called $Vistools$, i.e., the methods defned in the Scalable Visualization Toolkit, enable access to large-scale data sets and have been developed as a part of the NPACI Alpha project "$Scala\ \ leVisualizaton\ Toolkit\ for\ Bays\ to\ Brains$". The $Vistools$ project is supposed to develop an integrated set of tools for analysis, fl - tering, compositing, rendering, and interacting with very large multi-dimensional, mutli-

modal, time-varying data sets that are too large to ft into main memory and swap space of desktops to terafops computer systems [19]. The toolkit can be used to extract 2-D cross-sections or subvolumes from a large volumetric data set. The toolkit is available for download on the Internet [19] and has been implemented both in a Java and a C++ version.

After reading a data set from a storage device using the toolkit, Haar wavelets are used to transform the 3-D volumes or sub-volumes into lower-resolution representations. The transformation is performed on the server before the data set or a part of it is transmitted across the network. This transformation enables progressive transmission of various detail levels over the network. First, a low-resolution version is transmitted to provide the user with a quick preview version of the data set. Then, a set of detail coeffcients is progressively transmitted to eventually enable a complete reconstruction of the data set on the client side. The 3-D volume is rendered at progressive levels of detail using 3D-texture-mapping in Java3D on the client's machine.

Texture mapping maps an image defned in texture space onto an object defned in object space to change its visual appearence. This method can be used to map a texture image onto a polygon. To obtain a 3-D volume, a series of texture images is mapped onto an array of polygons in a back-to-front order. In order to enable correct blending and compositing, transparency and correct depth ordering are applied to display the textured objects.

This thesis presents a wavelet-based rendering system that has been implemented in both Java/Java3D and C++/OpenGL. As Java is platform-independent and widely sup-

ported on many systems, it is well suitable for Internet-based applications. For this reason, this rendering system focuses on using Java3D which is an application interface to a sophisticated three-dimensional graphics rendering and sound rendering system. The system has also been implemented in C++ to conduct a comparative study of the rendering performance on both platforms. The limitation of the C++ version of the application is that it is platform-dependent and hence lacks portability. The purpose of all the techniques mentioned before is to make the system more effcient and to facilitate interactive rendering.

This thesis focuses on a Haar wavelet implementation in 3-D and discusses in detail the methodologies to render a 3-D volume using texture mapping. Chapter 2 presents a brief backgound study and related work in this feld of research. Chapter 3 discusses the proposed system architecture of the wavelet-based rendering system. This section also discusses the principles of wavelets and reasons why Haar wavelets have been chosen for this particular application. A brief discussion of the structure of the rendering pipeline will follow. Chapter 4 deals with the implementation details which includes a description of the data types, the data structures, and the classes and methods are used in implementing the wavelet-based rendering system. Chapter 5 describes the performance statistics of the system and the results obtained on both the Java and the C++ platforms. This chapter also evaluates the statistics obtained and discusses the reasons that cause a difference in the performance on the two platforms. Chapter 6 presents the conclusion of this thesis showing that it is possible to implement an interactive volume rendering system in Java3D with reasonable performance, and it discusses future prospects of this research.

CHAPTER II

RELATED WORK

2.1 Why Haar wavelets?

Uncompressed data sets require massive storage capacity and transmission bandwidth. Though there have been advances in mass-storage density, processor speeds, and digital communication system performance, demand on data storage capacity and data-transmission bandwidth continue to pose a challenge to the existing applications. JPEG compression has been established as a standard for still image compression [21]. JPEG compression uses the discrete cosine transform (DCT) to calculate the transformation coeffcients for a pixel block. These DCT coeffcients are then rounded off according to the specifed quantization matrix. JPEG is a lossy technique since quantization introduces artifacts due to reduction of detail information. Despite several advantages of JPEG like simplicity, satisfactory compression and decompression performance and availability of special purpose hardware implementations, there are several drawbacks, for instance, loss or shift of color information due to the chosen color model (YIQ), and block artifacts at low bit rates [23]. Wavelets have been proven to eliminate these artifacts as they typically don't use color model transformations, and because their basis functions have local support of variable length, they usually don't expose block artifacts. Moreover, wavelet coding is

more robust under transmission and decoding errors because on the next level, errors are smoothed out, and it also facilitates progressive transmission of images (recursive image decomposition). Wavelets also have an inherent multiresolution nature which makes them suitable for applications where scalability and tolerable degradation are desirable.

Wavelets have their roots in approximation theory and signal processing. Recently they have been applied to many felds in computer graphics such as image processing, image compression and image querying, automatic level-of-detail control for editing and rendering curves and surfaces, surface reconstruction from contours, and fast methods for solving simulation problems in animation and global illumination [26].

A discrete wavelet transform (DWT) can be interpreted as an improved variant or extension of a discrete fourier transform (DFT). A basic DFT is a linear, invertible function, which transforms a periodic signal from a spatial domain into the frequency domain. The basic goal of a Fourier series is to take a signal, which is considered as a function of a time variable t, and decompose it into its various frequency components [5].

One disadvantage of a Fourier series is that its building blocks, weighted sine and cosine functions, are periodic waves with infnite support. While a DFT is appropriate for fltering or compressing signals that have time-independent wave-like features, it fails for non-periodic signals that have localized features which cannot be represented very well using sine and cosine functions. Representing those features accurately would require an extremely large amount of detail coeffcients. In cases of signals having localized features, wavelets typically prove to be more advantageous. A wavelet looks like a wave segment

that ranges over a short period and is non-zero only over a fnite interval instead of propagating forever the way sines and cosines do [5]. This means that wavelets feature local support. While a DFT can provide only frequency information, a DWT can keep track of both spatial and frequency information, and therefore preserves location. This property of wavelets is useful for instance for biomedical imaging to store large images, for computational fuid dynamics to store large simulation results, and for applications in geophysics to analyze data from seismic surveys [5].

Wavelets are used to represent different levels of detail. The Haar wavelet is one of the simplest wavelet transforms. There is a wide variety of popular wavelet algorithms, including Daubechies wavelets, Mexican Hat wavelets and Morlet wavelets [12]. These wavelet algorithms have the advantage of providing better resolution for smoothly changing time series. But they have the disadvantage of being more expensive to calculate than Haar wavelets [12]. Also, since we are dealing with pixel data, which resemble more a rectangular signal than a continuous analog signal, Haar wavelets that use box functions as base functions are best suited to represent the original signal.

In most cases of image transformation wavelets have been discussed in two dimensions. As volume rendering of images has become more and more important, the necessity to store volume data in a more compact representation and the need for hierarchical, multi-resolution representations and progressive data transmission has inspired us to implement a Haar wavelet transfomation in 3-D. Wavelets have proved to be faster in terms of their computational complexity as compared to Fast Fourier Transforms. What makes

wavelets superior to established compression methods such as JPEG is their ability to adapt to the size and location of features in the image. We use this technique to prove that the performance of a 3-D volume rendering algorithm in Java3D can be signifcantly improved without signifcant loss of image quality, and that this technique is suitable to obtain performance rates that are comparable to a C/C++ implementation.

2.2 Basic rendering techniques

There are various methods to visualize a large amount of scientifc data. Two major categories of rendering techniques are surface rendering and volume rendering [8]. Surface rendering is a technique in which volumetric data is frst converted into geometric primitives (such as polygonal meshes or contours) which are then rendered for display using conventional rendering techniques [3]. The major advantage of surface rendering is that the 3-D data set is reduced to a set of geometric primitives which can result in a signifcant reduction in the amount of data to be stored and in a much faster display due to the hardware support. But the conversion to geometric primitives can become diffcult if the surfaces are not well defned (e.g. in noisy 2-D images), and the number of surface polygons produced can sometimes be extremely high and exceed the capabilities of the rendering hardware engine [3]. Marching cubes is a 3-D surface reconstruction algorithm that produces models with unprecedented detail. This algorithm creates a polygonal representation of iso-surfaces from a 3-D array of data [14].

Volume rendering is a technique in which an object of interest is represented by a large number of cubic blocks called voxels [3]. A voxel is analogous to a pixel and it represents a measure of a unit volume. A 3-D voxel grid can be assembled from multiple 2-D images and then displayed by projecting the volume into a 2-D pixel space where the image is stored in a frame buffer.

Volume rendering techniques has been developed to overcome problems of accurate representation of surfaces in isosurface techniques [2]. A major advantage of volume rendering techniques is that the 3-D volume can be displayed without any knowledge of the geometry of the data set and hence without intermediate conversion to a surface representation. Since the entire data set is preserved in volume rendering, any part, including the internal structures and details, may be viewed.

Ray tracing is a rendering technique that projects a ray from the observer through each pixel on the screen into the scene. If the ray intersects an object in the environment, the object's color contributes to the color of the corresponding screen pixel [6]. Ray-tracing is a very intutive method, because it is physically plausible and, if the data set is properly sampled, accurate.

Classical ray-tracing is computationally expensive as the ray intersection and shadow tests consume much of the available computing resources [6]. This can be deviated by simplifying the algorithm to neglect refected rays. This technique is called ray-casting.

Splatting is another volume rendering technique that differs from ray tracing in the projection method [2]. Splatting is an object-order traversal algorithm where the vertices

of a grid voxel are splatted onto an image plane [11]. Splatting is computationally expensive due to the complex projection of the volume data although it is more effcient than ray tracing and volume shearing. Splatting does not correctly render cases where the volume sampling rate is higher than the image sample rate (i.e., more than one voxel maps into a pixel) [27].

In this thesis, texture mapping is used to render a series of 2-D slices as a 3-D volume. Texture mapping is a technique that has been traditionally used to add realism to a scene in computer graphics. This technique can be used to enhance the appearance of an object in a raster scan image with only a small increase in computation. Textures are usually rectangle arrays of data. In basic texture mapping, a texture image is applied to a polygon by assigning texture coordinates to the polygon's vertices [10]. Texture mapping can be implemented both in hardware and software. Software implementations beneft from the fact that they do not depend on the specifc hardware platform for which they are written, which enables a certain level of platform portability. But software implementations have limited rendering speeds [16]. Hardware implementations can take adavantage of interacting with specialized graphics processors that can speed up the rendering process. The high-performance hardware texture mapping solutions makes hardware-accelerated texture-based visualization a more desirable tecnique for volume rendering [22]. OpenGL and the latest versions of Java3D have their own built-in hardware texture mapping routines.

2.3 What is Java3D?

The Java3D Application Programming Interface (API) provides a set of object-oriented methods to support a simple, high-level programming model for 3D graphics. This technology enables developers to incorporate high-quality, scalable, platform-independent 3D graphics into Java applications and applets. The Java3D API uses the scene-graph model that helps the users to focus on objects and scene composition [25]. Scene graphs are hierarchical structures for composing scenes containing geometric primitives, volume data sets and space-flling functions [20]. Scene graphs tend to speed up applications since programmers do not need to design specifc geometric shapes or write rendering code for the scene display. Rendered scenes can also be easily modifed using the scene graph concept. Java3D takes advantage of existing hardware accelerators through the underlying low-level APIs such as OpenGL or Direct3D. This allows the Java3D API to run on any platform with a Java virtual machine, Java3D, and an OpenGL or Direct3D implemetation [25]. Java3D enables developers to create realistic and complex graphics using advanced texture mapping technologies such as texture cube mapping, extended texture environment combiners, anisotropic texture fltering, texture LOD, and texture data update. All these features have encouraged us to use the Java3D API to design our graphics application.

CHAPTER III

SYSTEM ARCHITECTURE

The main goal of this thesis is to develop and analyze a platform-independent rendering system that uses the wavelet concept to transform large data sets into more compact representations that can be easily transmitted over a low-/medium-bandwidth network. The purpose of the study is to fnd out whether wavelet compression techniques are suffcient to compensate for the loss of performance caused by the use of Java/Java3D and therefore introducing an additional layer between the application and the underlying graphics hardware API, compared to a more direct implementation in C/C++, which avoids a language interpretation step and an additional API layer. The rendering system proposed here has a server and a client.

To make use of enhanced storage capabilities, the data sets are stored on a High-performance Storage System(HPSS), which allows the user to defne public and private user groups, and multiple copies or versions of a fle at different locations. NPACI's Scalable Visualization Toolkit(Vistools) accesses the HPSS through a Storage Resource Broker(SRB) developed at SDSC. The actual storage pattern is transparent to the user. The data is retrived as a subset of 2-D cross-sections that are assembled into 3-D subvolumes

[24]. These subvolumes are transformed using a Haar wavelet transformation on the server before being transmitted to the client's machine.

On the client's machine the transformed wavelet packets are reconstructed using a Haar wavelet reconstruction algorithm, and the 3-D volume is rendered using the texture mapping methods in Java3D. A set of 2-D planes is created and placed in a three-dimensional texture space in order to obtain a three-dimensional image. By changing the texture coordinates, the volume can be rotated, translated or zoomed. The number of textured planes that are used is determined by the client application.

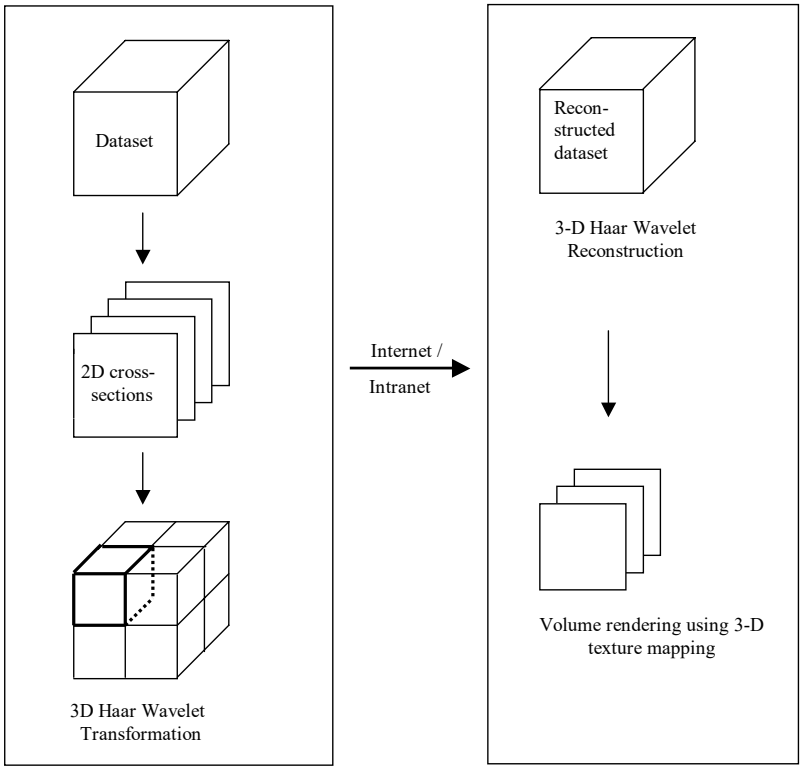


Figure 3.1 System architecture

3.1 Extraction of 2-D cross-sections and subvolumes

The $Vistools$ are available in both a Java and a C++ version. To implement a platform-independent application, the Java version of the $Vistools$ is used. $Vistools$ provide a method that reads a range of elements from a data set, and returns their values in the host's byte order and word size. This method implements a decoder, which reads the data from the data set and decodes its format. $Vistools$ transparently supports paging of data in and out of main memory to enable larger-than-core visualization. Sometimes there are situations where there is no need to render the entire data set. For example, a physican or biologist might be interested only in a particular part of the brain for conducting his or her analysis or experiment. For such cases, $Vistools$ support a special fle format termed as the chunked fle format which enables the extraction of subvolumes from a data set. $Vistools$ provide a method which computes a one-dimensional memory index corresponding to the given $n-$dimensional grid coordinates in the data set. This index is used to read the data from the data set using the method that decodes the format of the data set [19]. The details about these methods will be discussed in later chapters.

3.2 3-D Haar Wavelet Algorithm

After the cross-sections are extracted from the data set and combined into a 3-D volume, the data needs to be transformed into a multi-resolution representation to enable faster transmission over present networks. The Haar wavelet transform is one of the simplest and most effcient transformations. This method has been implemented in 3-D space

to enable transformation of a 3-D volume. The Haar wavelet transfomation decomposes an image into a set of low-pass flter coeffcients and a set of detail coeffcients. To give a better idea of the actual implementation of the wavelet transformation, we illustrate the procedure with a simple example.

Assume we have a one-dimensional image with an 8-pixel resolution where the pixels have the following values [26]:

7	5	3	9	3	7	5	3

Figure 3.2 Original data

By applying the Haar wavelet transformation we can represent this image in terms of a low-resolution image (low-pass flter coeffcients) and a set of detail coeffcients (high-pass flter coeffcients). The low-pass flter coeffcients are obtained by averages two consecutive pixels, while the detail coeffcients represent the difference between the average and one of the two consecutive pixels. So the above image will be represented as follows after the frst cycle:

6	6	5	4	1	-3	-2	1

Low-pass filter coefficients Detail coefficients

Figure 3.3 Low-pass and high-pass flter coeffcients

Now the original image can be represented as a low-resolution image ((a+b)/2), which consists of four pixels, and another four-pixel image which contains the detail coeffcients ((a-b)/2). The low-resolution image constitutes the low-pass flter coeffcients after the wavelet tranformation. The other half contains the detail coeffcients. Recursively iterating this algorithm leads to an image that is reduced by a factor of two for each cycle.

This simple 1-D scheme can be lifted to higher-dimensional cases. For a 2-D wavelet transformation, this algorithm is applied in $x-$direction frst, and then in $y-$direction. Similarly, in a 3-D wavelet transformation the structures are defned in 3-D and the transformation algorithm is applied in $x-$, $y-$, and $z-$directions, respectively. One cycle for an $n-$dimensional data set is defned as the completion of the algorithm for all n directions. The details about the data structures and implementation will be discussed in the next chapters.

3.3 3-D Haar Wavelet Reconstruction

During transmission, the low-pass flter coeffcients are sent frst while the detail coeffcients are transmitted at a later time. When the detail coeffcients are received on the client side, detail information is added to the coarser volume, which has already been rendered, to refne the image. The reconstruction of the image data uses simple arithmetic operations (integer arithmetic). As the image array received on the client side consists of the low-pass and high-pass flter coeffcients, the respective pixel values at each cycle are obtained by adding and subtracting the corresponding detail coeffcients to and from

the low-pass flter coeffcients. To illustrate this technique, let us consider the 1-D image, which was introduced in the previous section for explaining the transfomation process.

```
Low-resolution image: 6 6 5 4

Detail coefficients: 1 -3 -2 1
```

To obtain the pixel values of the next higher level of detail, we add the low-pass and detail coeffcients to obtain one pixel of the frst pixel pair, and subtract them to get the second pixel of each pair. The reconstructued pixel values are

6+1	6-1	6-3	6+3	5-2	5+2	4+1	4-1
7	5	3	9	3	7	5	3

Figure 3.4 Reconstructed Data

These values are identical to the original image. After the data set has been reconstructed using the Haar wavelet reconstruction algorithm, the volume is rendered using 3-D texture mapping in Java3D.

3.4 3-D Texture Mapping

Texture mapping is a technique that maps an image from a texture space into object space. When mapping an image onto a object, the color of the object at each pixel is modifed by a color value read from a corresponding location in the texture image. Normally, the texture image is stored as a contiguous array so that it is easier to reconstruct a

continuous image from the samples. The individual values of the texture image are called texels. The texture image must be warped to match any distortion in the projected object being displayed. Since the texture is made up of discrete texels, fltering operations must be performed to map the texels onto the fragments. Texture flters are used to interpolate between texels. Due to the fltering calculations, texturing can be expensive, but the presence of hardware support for texture mapping makes it a fast and potentially interactive method of rendering.

We have implemented two different methods for volume rendering based on textures. The frst one uses 2-D texture mapping and three sets of perpendicular planes. The second one uses 3-D texture mapping and only one set of planes that always faces the camera (parallel to the viewing plane).

For the frst method, we use 2-D texture mapping to map a series of 2-D images onto a series of quadrilaterals to make up a 3-D volume. The 2-D images are mapped in back-to-front order onto a series of quadrilaterals arranged parallel in the $x-$, $y-$, and $z-$direction (Figure 3.5) [7]. All the quadrilaterals are drawn as parallel planes, and the 3-D texture coordinates are choosen accordingly. Since the cross-sections are mapped one behind the other, perpendicular to the viewer, only the frst cross-section is visible while the other sections are occluded. To make the other cross-sections visible, transparency values are assigned to all quadrilaterals that are used for texture mapping. The resulting semi-transparent images of the data set reveal the interior structure of the object. The two-dimensional texture mapping techniques take advantage of the bilinear texture mapping

hardware available both in Java3D and in OpenGL. However, this technique suffers from sampling artifacts and also requires three copies of the data sliced along the three major axes [13].

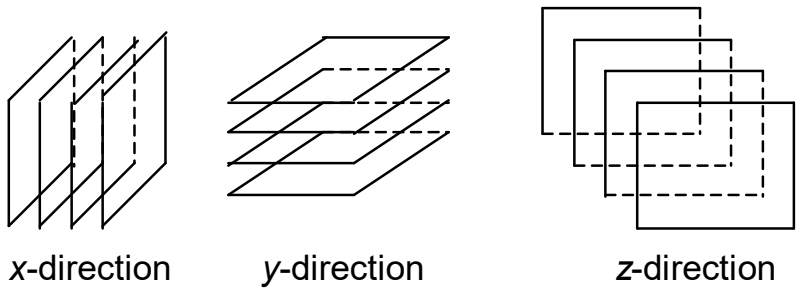


Figure 3.5 Simple 2-D texture mapping

The basic idea of the second method, 3-D texture mapping, is to interpret a voxel array as a 3-D texture defned in a 3-D space. First, the three-dimensional data set is loaded as a 3-D texture block into the texture buffer of the graphics system [4]. The scene comprises of a parallel stack of quadrilaterals that are orthogonal to the screen. Each quadrilateral vertex is associated with a point in texture space, and the graphics system maps values from the texture value onto the surface of the quadrilateral by trilinearly interpolating [13] the texture co-ordinates [9]. To enable to see through the planes, transparency values are assigned to each surface point and the data is correctly blended into the frame buffer [16] [28]. The quadrilaterals remain parallel to the screen even as the client changes the position or orientation of the texture map. As the number of quadrilaterals drawn increases and the spacing between the quadrilaterals decreases, the image quality gets better. With

fewer quadrilaterals, the rendering speed is higher. Therefore, there is a trade-off between rendering performance and the level-of-detail used during visualization.

This technique requires 3-D texture mapping support of the rendering engine, but it has the advantage that the texture memory needs to be loaded only once no matter what the viewpoint is. Therefore, this method is very effcient for interactive rendering with a dynamic viewpoint.

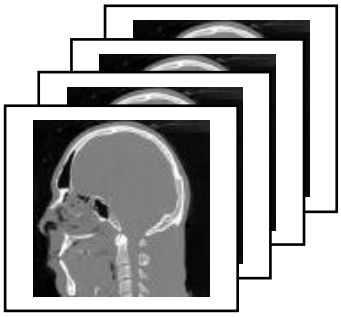


Figure 3.6 3-D Texture mapping

3.5 Texturing Pipeline

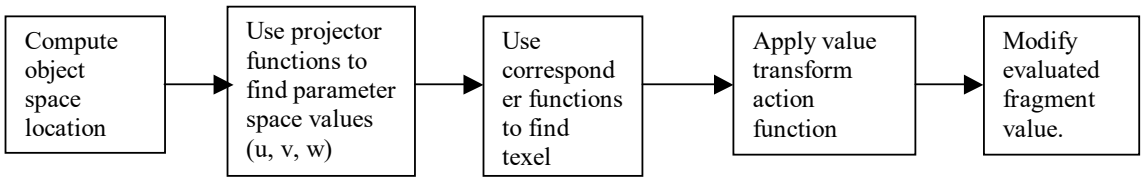


Figure 3.7 Texturing pipeline

Texturing starts with defning a location (x, y, z) of a model in world co-ordinates Figure 3.7 [18]. This location is in reference to the model, i.e., as the model moves the texture also moves. The projection function is applied to the world co-ordinates to obtain a set of parameter-space values (u, v, w) that are used to access the texture. This process is called texture mapping. Projection functions are typically used to convert a three-dimensional point in space into 2-D or 3-D texture co-ordinates. Spherical, cylindrical, and planar projections are the most commonly used projection functions. Before the new parameter-space values are used to access the texture, corresponder functions can be applied to transform the parameter-space values to texture space. Corresponder functions can be used to provide fe xibilty in applying textures to surfaces. Typical corresponder functions in OpenGL are wrap, repeat, and tile. The texture values read from the texture buffer are further transformed by a value transformation function. These values are fnally used to modify some property of the surface such as material or an associated shading normal [18].

CHAPTER IV

IMPLEMENTATION

This chapter gives a formal description of the data structures and the classes used in implementing the "Wavelet Based Volume Rendering System". The data sets that are required for this application are stored in the VOL fle format. The $Vistools$ package provides classes and methods to extract data from these data sets.

4.1 Accessing and storing the data sets

The $Vistools$ architecture is composed of multiple layers Figure 4.1 [19]. Each layer builds upon the layer underneath it and extends the underlying layer. Raw data are read from the local or remote storage at the bottom-most layer. The $PagedArray$ class creates a 1-D array abstraction that enables the application and the higher level toolkits to manipulate very large 1-D arrays of fx ed size elements. The get and the set methods are used to access the element bytes in a 1-D array. The $Mesh$ class provides a n-dimensional structured or unstructured mesh of elements.

The $FormatCatalog$ class implements a $createDecoder$ method that creates a fle format decoder to decode the named input fle. This method returns an object of type $FormatDecoder$ that is cast to an object of class $MeshFormatDecoder$. A $Structured$-

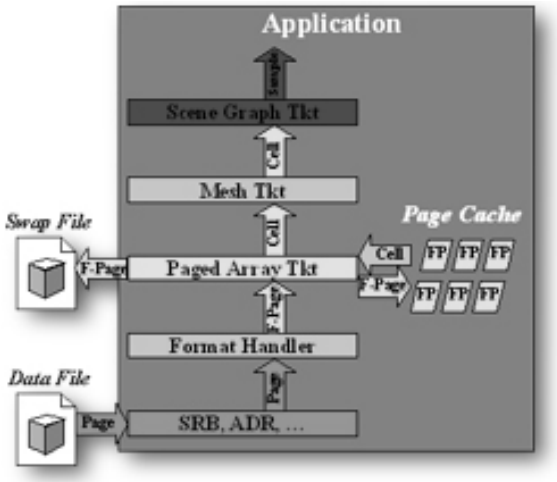
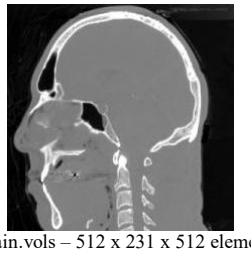


Figure 4.1 Vistools architecture

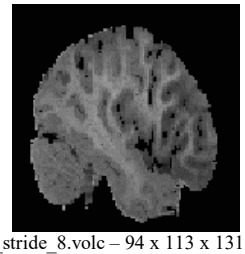
$Mesh$ is created using the $MeshFormatDecoder$. A structured mesh describes a structured arrangement of elements within a grid occupying a multi-dimensional foating-point coordinate space. The $getMeshAttri\ utes$ method in the $StructuredMesh$ class returns a $StructuredMeshAttri\ utes$ object. Using this object, the grid dimensions can be obtained. The $getGridDimensionAttri\ utes$ method in $StructuredMeshAttri\ utes$ class returns an object of type $GridDimensionAttri\ utes$. The $getSize$ method in the $GridDimensionAttri\ utes$ class retrieves the size of the grid dimension specifed. Grid dimensions 0, 1, 2 represent the width, height and depth of the $StructuredMesh$ created. The $Element$ class is used to hold a copy of values found within an element in the mesh. The get method in the $StructuredMesh$ class retrieves a copy of the values within an element, returning them in the given $Element$ object. The element whose value needs to

be retrived is specifed by an index within the 3-D mesh. $Element$ implements a range of methods to access the value of the selected element feld.

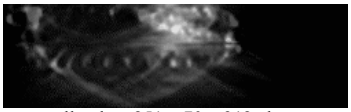
$Vistools$ support a number of fle formats. The $vols$ fle format supports 8-bit scalar values. Hence, the $getByte$ method is used to access the value of the selected element feld. The $volc$ fle format supports 64-bit values where the frst 32-bit word contains a 10-bit red, a 12-bit green, and a 10-bit blue value. The second 32-bit word contains 16-bit alpha and 16-bit beta values. As each color component occupies more than 8 bits, the $getShort$ method is used to extract the red, green, and blue components individually. If the full color resolution is not needed, the color values can be mapped into the range between 0 to 255 by shifting the red, green, and blue color components 2, 4, and 2 bits respectively to the right to obtain 8-bit red, green, and blue values. The color components are stored in an array which is later used for the Haar wavelet transformation.



tbrain.vols – 512 x 231 x 512 elements



brain_stride_8.volc – 94 x 113 x 131 elements



cell.volc – 251 x 70 x 312 elements

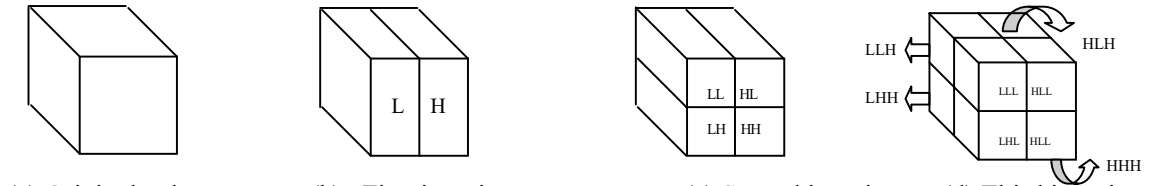
Figure 4.2 Extracted 2-D cross-sections

4.2 Haar Wavelet Transform

The original data is stored in a 4-D array. The Haar wavelet algorithm described in section 3.2 of chapter 3 is implemented in a separate class. This class implements the $WaveletTransform$ method that takes the number of wavelet cycles that the user specifes to transform the data, the number of cycles for which the data has already been transformed, and the 4-D data array that needs to be transformed. The user is not required to know how many cycles the data has been previoulsy transformed. The application keeps track of the number of cycles the data set has been transformed previously. The frst three dimensions of the 4-D array represent the depth, height and width of the original data set rounded to the nearest powers of two. The fourth dimension represents the number of color components used to represent the pixel data. To implement the Haar wavelet algorithm in 3-D the data array needs to be transformed in $x-$, $y-$, and $z-$ directions. The implementation of the algorithm is as follows:

1. Calculate the array indices depending on the number of wavelet cycles requested by the user for the storage of the low-pass flter and detail coeffcients. Care is taken such that at the end of each cycle the transform coeffcients occupy the upper left corner of the array.
2. Perform the Haar wavelet transform along the $x-$direction, for all slices. A temporary array is used to store the calculated coeffcients of each row. After the transformation is complete for a single row, the contents of the temporary array are copied into the specifed row of the main array. This procedure is implemented to facilitate the usage of a single array to represent the transformed and detail coeffcients for any number of transformation cycles. It also makes the algorithm scalable and reduces the amount of main memory that is required to hold the current data set signifcantly .
3. Similarly perform the Haar wavelet transform in the $y-$ and $z-$direction.
4. Repeat steps 1, 2, and 3 for the number of wavelet transformation cycles requested by the user.

5. The transformed coeffcients are transmitted to the client's machine for rendering.

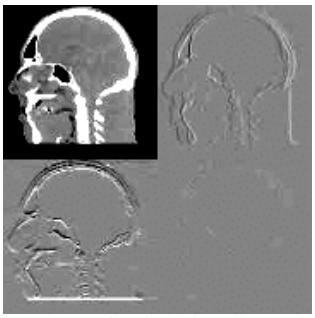


(a) Original volume (n x n x n) (b) First iteration: (x-direction) (c) Second iteration (y-direction) (d) Third iteration (z-direction)

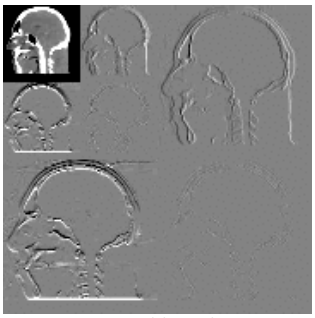
Figure 4.3 One cycle of a 3-D Haar wavelet transformation

4.3 Haar Wavelet Reconstruction

Once the scene is rendered with the low-pass coeffcients, the image can be refned by using the detail coeffcients. In order to initiate a progressive reconstruction of the rendered scene, the user or the underlying application needs to specify the number of reconstruction cycles. Once the request is processed the scene is rendered with the reconstructed data. The reconstruction algorithm is implemented in a separate class. This class implementes a $InverseWaveletTransform$ method that takes the number of reconstruction cycles requested by the user, the number of cycles the data has been transformed, and the 4-D transformed array, as input parameters. This method returns a reconstructed array of data. The $getTransformedData$ method selects the octant that comprises the reconstructed data. This selected data is used in rendering the reconstructed scene. The reconstruction algorithm explained in section 3.3 of chapter 3 is implemented as follows:

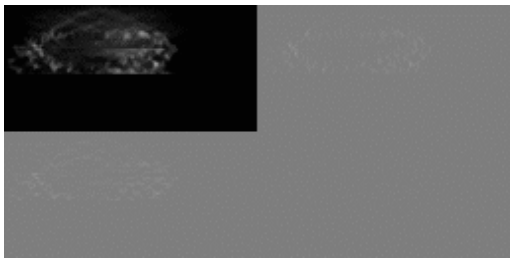


First iteration



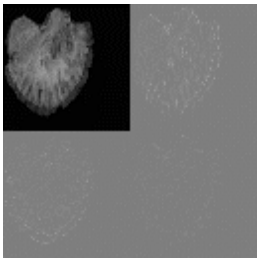
Second iteration

Figure 4.4 Low-pass and high-pass flter coeffcients of a CT skull

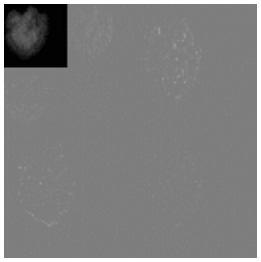


First iteration

Second iteration



First iteration



Second iteration

Figure 4.5 Haar wavelet coeffcients of a brain cell and a MRI brain

1. Calculate the array indices depending on the number of reconstruction cycles to retrieve a low-pass flter coeffcient along with its corresponding detail coeffcient to calculate two adjoining pixel pair values. These pixel values are stored in the upper left front octant of the array. The detail coeffcients of the remaining cycles are left intact in the other octants of the array.
2. The pixel pair values are calculated by summing and subtracting the corresponding low-pass and high-pass flter coeffcient pairs. A temporary array is used to store the calculated pixel pair values for a single row. Once the calculations are complete for a single row, the contents of the temporary array are copied into the main array.
3. The reconstruction process is performed in the $x-, y-$ and $z-$directions.
4. Steps 1, 2, and 3 are performed for the number of reconstruction cycles requested by the user.
5. The reconstructed octant is transmitted and the scene is rendered with the reconstructed data.

4.4 Texture Mapping in Java3D

Java3D uses a scene graph to load all the Java3D objects created to render a scene. The $virtualuniverse$ is the root node of the scene graph. The $virtualuniverse$ is defned by a collection of objects that need to be rendered. Each node in a scene graph represents an instance of a Java3D class and the arcs represent the relationship between the data elements. Two kinds of relationships are represented by the arcs, namely, parent-child relationship and reference relationship [24].

The $Locale$ acts as a root to a subgraph of a scene graph. Its also a provides a reference point in the $virtualuniverse$. The $BranchGroup$ object (BG) is the root of the sub-graph or the branch graph. The right child of the $Locale$ represents the view branch graph and the left child represents the content branch graph. The view branch graph represents the

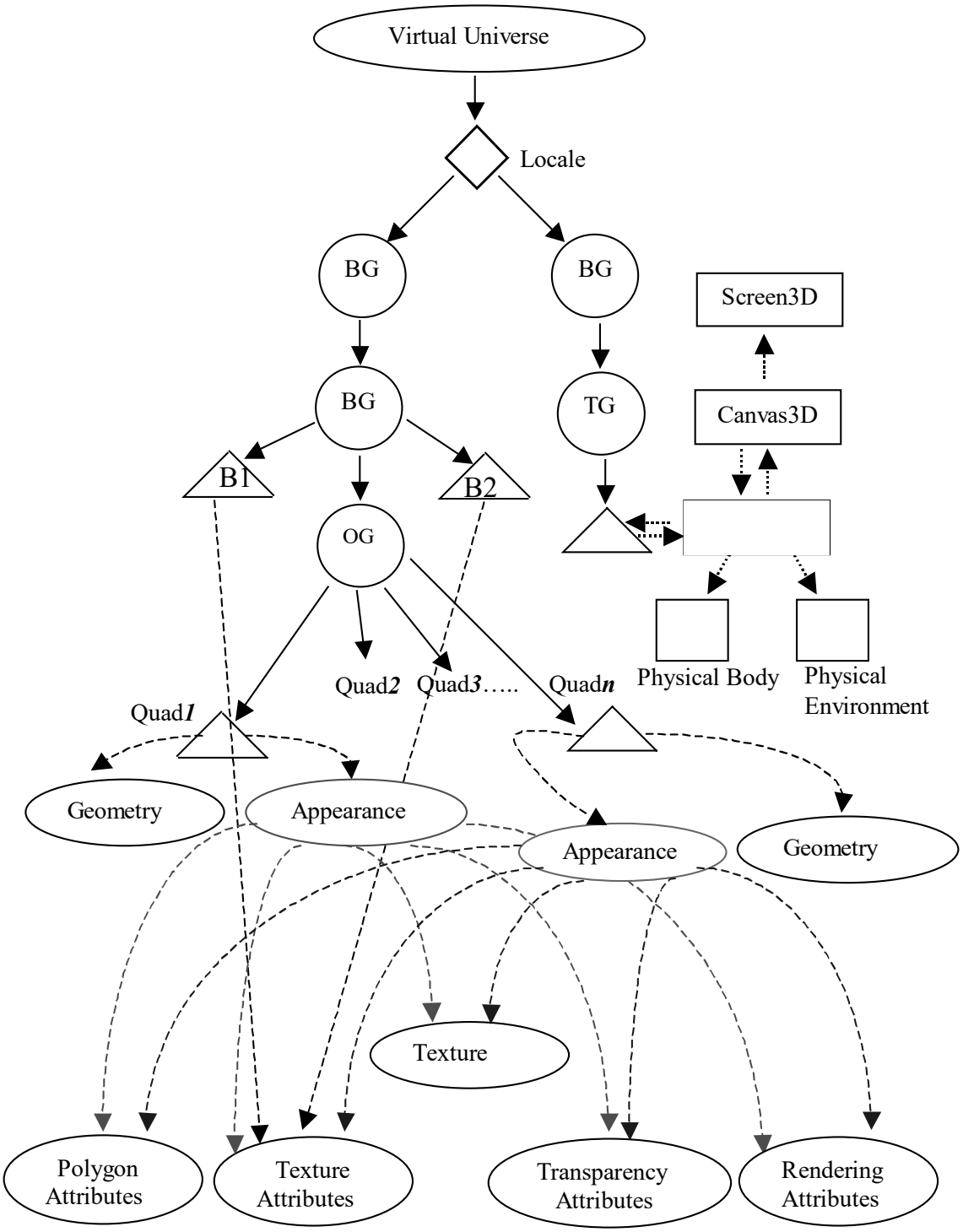


Figure 4.6 Scene graph

viewing location and direction. The content branch graph represents the geometry, appearance, behaviours, and location. The content branch graph of the scene graph in Figure 4.6 consists of two $BranchGroup$ objects. One BG has another BG as its child. The second BG has three children namely B1, B2 and OG. B1 and B2 are $Behavior$ objects that are created to invoke the Java3D renderer on keyboard and mouse inputs. The action of these $Behavior$ objects will affect the $TextureAttri\ utes$ object when a keyboard or mouse event occurs while rotating, translating or scaling the texture. The OG is an $OrderedGroup$ object that is used to place the planes in a particular order for rendering. The OG object consists of several $Shape3D$ nodes, which represent texture quads in our scene graph. Each $Shape3D$ object references a $Geometry$ object and an $Appearance$ object. $Geometry$ represents a plane that is going to be rendered while the $Appearance$ object references to various attributes of the texture such as $Texture$, $PolygonAttri\ utes$, $TextureAtti\ utes$, $TransparencyAttri\ utes$, and $RenderingAttri\ utes$. The $Texture$ object holds the 3-D texture buffer that is used for 3-D texturing. The $PolygonAttri\ utes$ object is used for defning attributes for rendering polygon primitives. The $Texture-Attri\ utes$ object is used to specify the texture mapping attributes. The $Rendering-Attri\ utes$ object is used to specify whether the alpha channel is used or not. The $Trans-parencyAttri\ utes$ object is used for alpha blending [24].

After the data passes the transformation and the reconstruction stages and is ready to be rendered, an $ImageComponent3D$ object is used to create a 3-D buffer which holds a series of 2-D buffered images. 2-D cross-sections are extracted from the data array and

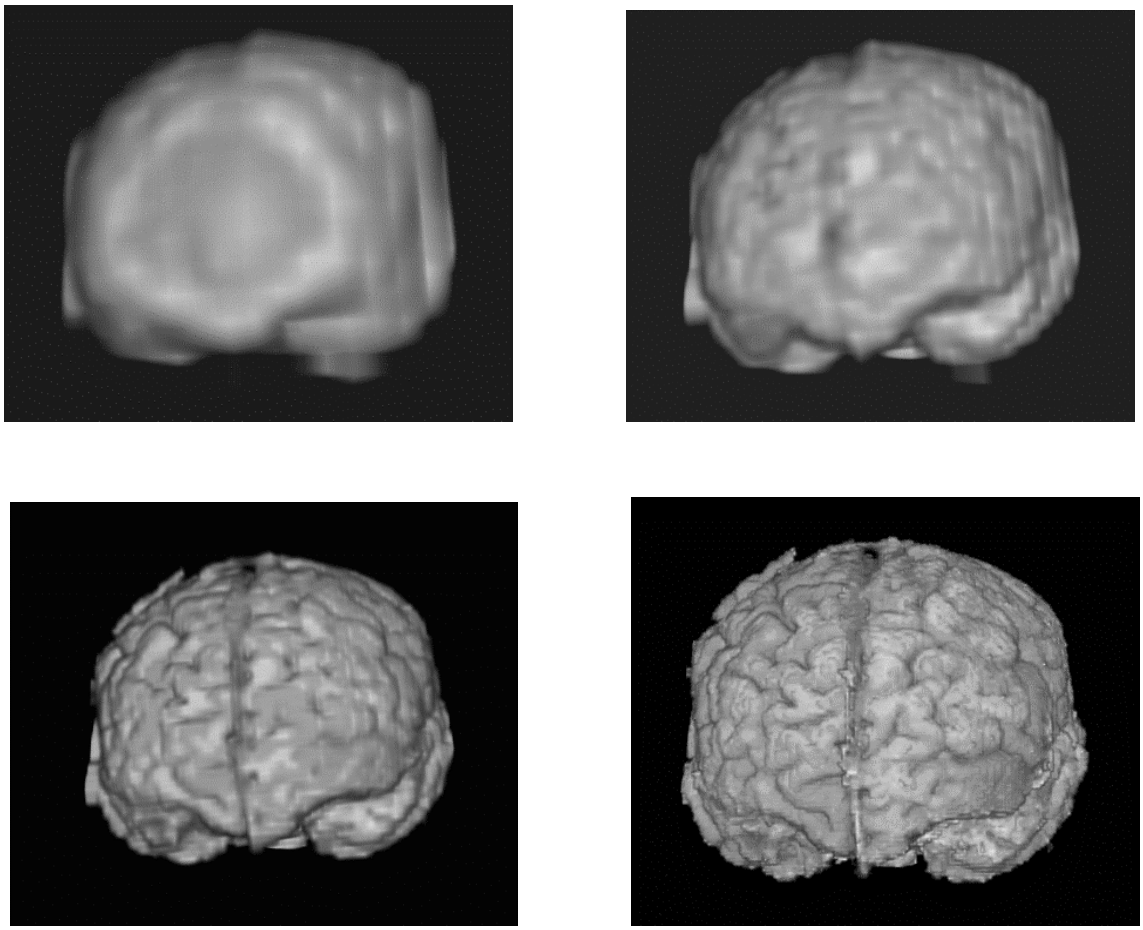


Figure 4.7 Progressive reconstruction of a MRI brain

a 2-D image buffer is created using the $BufferedImage$ object. A $BufferedImage$ object describes an image with an accessible buffer of image data and comprises of a $ColorModel$ and a $Raster$ of image data. A $ColorModel$ is a class that handles pixel values represented by color and alpha information and that stores each value in a separate data element. It is used in a $ColorSpace$. A $Raster$ class is used to represent a rectangular array of pixels. It encapsulates a $DataBuffer$ that stores values of the 2-D slices, and a $SampleModel$ that is used to locate a sample value in the $DataBuffer$. A $SampleModel$ is created by creating an instance of a $ComponentSampleModel$ and by calling the $createCompatibleSampleModel$ method. A $ComponentSampleModel$ represents image data that is stored such that each value of a pixel occupies one data element of the $DataBuffer$. The created 2-D $BufferedImage$ is then added to the $ImageComponent3D$ object by specifying an index into the 3-D buffer. This process is repeated for all the 2-D cross-sections that are used to create a 3-D buffer. A 3-D texture is created using the $Texture3D$ object, which requires an $ImageComponent3D$ object. Once the 3-D texture is created, a set of parallel planes is created using the $Shape3D$ object. These planes are defned with suitable 3-D texture coordinates that are rendered in back-to-front order. $Shape3D$ is a leaf node, which consists of a $Geometry$ and an $Appearance$ object. The attributes defned by the $Appearance$ object are added to the scene graph so that the planes that are rendered use these attributes. Once all the objects listed above are inserted into the scene graph, the $BranchGraph$ is instantiated by inserting the $BranchGraph$ into the $Locale$, which enables Java3D to render the scene

graph. All the planes are rendered in back-to-front order using the 3-D texture mapping hardware, displaying the 3-D recontructed volume. Alpha blending is performed during the rendering process by assigning an appropriate blending function to the textured planes [24].

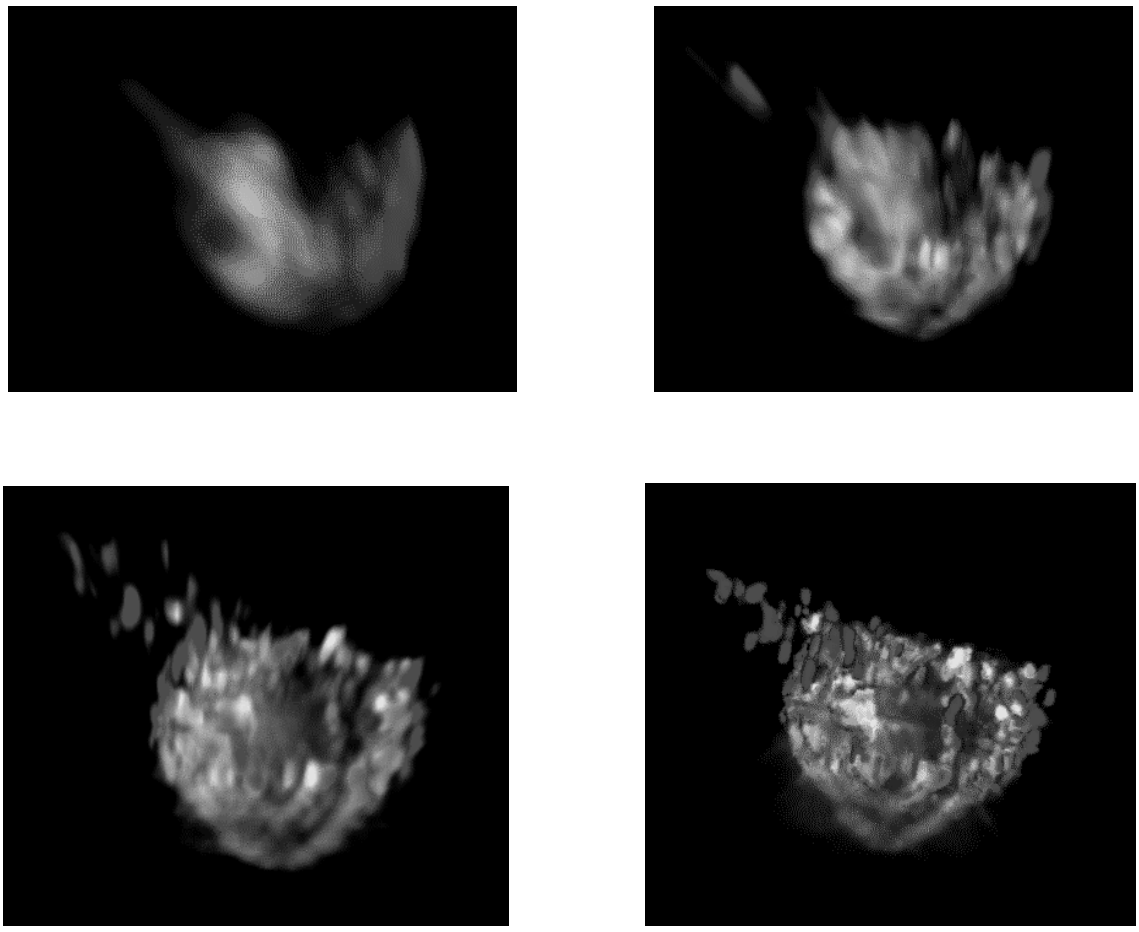


Figure 4.8 Progressive reconstruction of a brain cell

CHAPTER V

STATISTICS

The main goal of this thesis is to show that Java3D can be used to create interactive graphical applications. To obtain substantial proof for this thesis, a wavelet-based rendering system is implemented on both the Java and C++ platforms, and the loading, pre-processing and rendering performance of this system is observed. This thesis does not prove that Java3D applications are faster than OpenGL/C++ application but tries to establish that Java3D exhibits reasonable performance if compared to its C++ counterpart.

Though Java as an interpreted language executed on a virtual machine is typically slow compared to C++, it has gained popularity in web-based applications. The ”Wavelet-based rendering system” developed for this study is also designed to be accessible via the Internet. Java/Java3D is used to achieve this portability by accessing the data sets stored on the server through the Internet and rendering these data sets using the Java3D API and the hardware capabilities on the client’s machine. To analyze the rendering speeds of the application, the application has been implemented in both Java and C++ for comparison. Three different timings were taken, namely loading, preprocessing and rendering. The loading time is the time the application takes to load the data into a 3-D mesh structure. The preprocessing time comprises the time it takes to access data from the mesh structure

and to store it into arrays for the Haar wavelet transformations, and to create a texture buffer. The rendering time is the time it takes for the rendering API to render the texture planes. To obtain precise results, care has been taken to implement both applications in pretty much the same way on both the platforms, but under certain conditions the implementations tend to vary. This is described in the next section.

5.1 Differences in Java and C++ implementations

The main difference between the Java and the C++ implementation is the rendering method. Java3D uses the scene graph concept. Different components that make up the scene graph are explained in section 4.4 of chapter 4. OpenGL is used to implement 3-D texture mapping on a C++ platform. The image data that needs to be texture-mapped is stored in a 4-D array. To create a 3-D texture, $glTexImage3D()$ is used. The parameters that are passed to this method are the target (GL_TEXTURE_3D or GL_PROXY_TEXTURE_3D), the level of texture resolution, the internal format that is used to represent the texels of the image, the width, height and depth of the texture in powers of two, the value of the border, the format and data type of the texture image data, and the actual texture image data [29]. The mode in which the texture is to be applied can be specifed either as GL_REPLACE or GL_MODULATE. The 3-D texture is enabled using $glEnable$(GL_TEXTURE_3D). The function $glDisa\ le$(GL_TEXTURE_3D) can be used to disable 3-D textures. Finally, the texture coordinates and geometric coordinates are defned to render the texture.

5.2 Timings

Both the C++ and the Java version of the Wavelet-based Volume Rendering System have been executed on two different SGI workstations and their performance statistics are listed below. The timings shown below are an average of three different observations. The rendered images are transformed through one iteration of the wavelet algorithm.

Machine1 is an SGI with four 400 MHZ IP27 processors with 4096 MB of main memory and has an InfnityReality3 graphics engine. Machine1 can support textures with a width, height and depth of 2048, 2048, and 1 respectively. Machine2 is an SGI Octane workstation with two 400 MHZ IP30 processors with 1024 MB of main memory and has a V8 graphics engine. Machine2 can support textures with width, height and depth of 4096, 4096, and 1 respectively.

Table 5.1 Timing of a cancer cell rendering on Machine1

No. of slices/planes	Loading (ms)		Processing (ms)		Rendering(ms)		Total (ms)	
	Java	C++	Java	C++	Java	C++	Java	C++
8/16	10560.6	4.2	5252	6791.1	156	.005	15968.6	6795.3
16/32	10458.3	4.6	8360	13903.5	163.3	.005	18981.6	13908.1
32/64	10268	4.6	17708	28952.9	166.33	.005	28142.33	28957.5
64/128	10179.3	4.6	38370.6	61081.3	188	.005	48737.9	61085.9
128/256	9926.3	4.5	96933.6	120153.7	229	.005	107088.9	120158.2
256/512	10344.6	4.9	276297	241724.8	304.3	.006	286945.9	241729.7

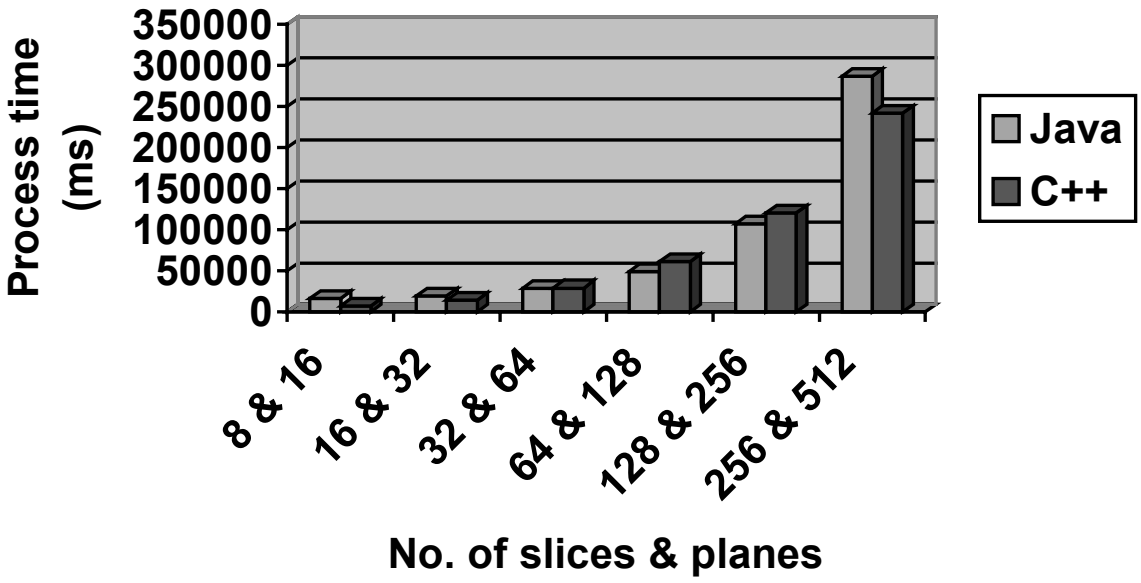


Figure 5.1 Histogram showing timings for rendering a cancer cell on Machine1

Table 5.2 Timings of a cancer cell rendering on Machine2

No. of slices/planes	Loading (ms)		Processing (ms)		Rendering(ms)		Total (ms)	
	Java	C++	Java	C++	Java	C++	Java	C++
8/16	10153	4.8	5849	8256.9	150.3	.005	16152.3	8261.7
16/32	10137.6	4.5	9264.6	14528.3	157	.004	19559.2	14532.8
32/64	10128	4.5	17898.3	27949.7	165	.004	28191.3	27954.2
64/128	10215	4.6	40122	58282.2	183.6	.004	50520.6	58286.8
128/256	10188.3	4.5	104429	122234.3	224.33	.004	114841.6	122243.3
256/512	10101	4.6	298917.6	221770	304	.004	309322.6	221774.6

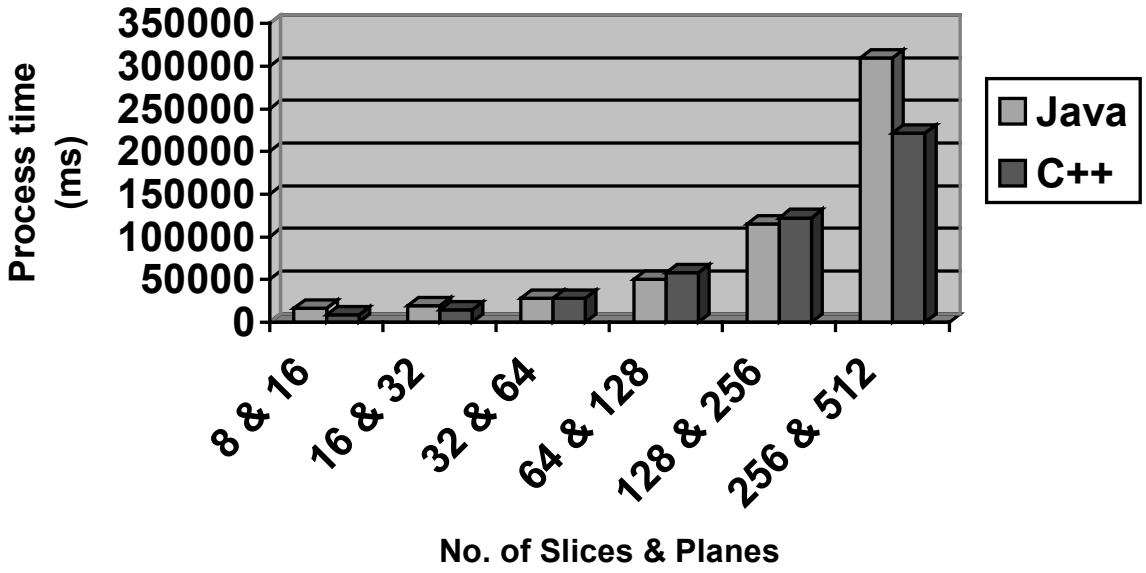


Figure 5.2 Histogram showing timings for rendering a cancer cell on Machine2

Table 5.3 Timings of a MRI brain rendering on Machine1

No. of slices/planes	Loading (ms)		Processing (ms)		Rendering(ms)		Total (ms)	
	Java	C++	Java	C++	Java	C++	Java	C++
8/16	2918	331.7	3172	452.9	151.6	.006	6241.6	784.6
16/32	2929.3	319.0	4817.3	877.9	151.6	.004	7898.2	1196.9
32/64	2800.3	325.6	8367.6	2175.1	167	.005	7769.2	2500.7
64/128	2923.3	336.8	16363	4818.0	187.6	.004	19473.9	5154.8
128/256	2880	344.4	37358	11095.3	224.3	.005	40462.3	11439.7

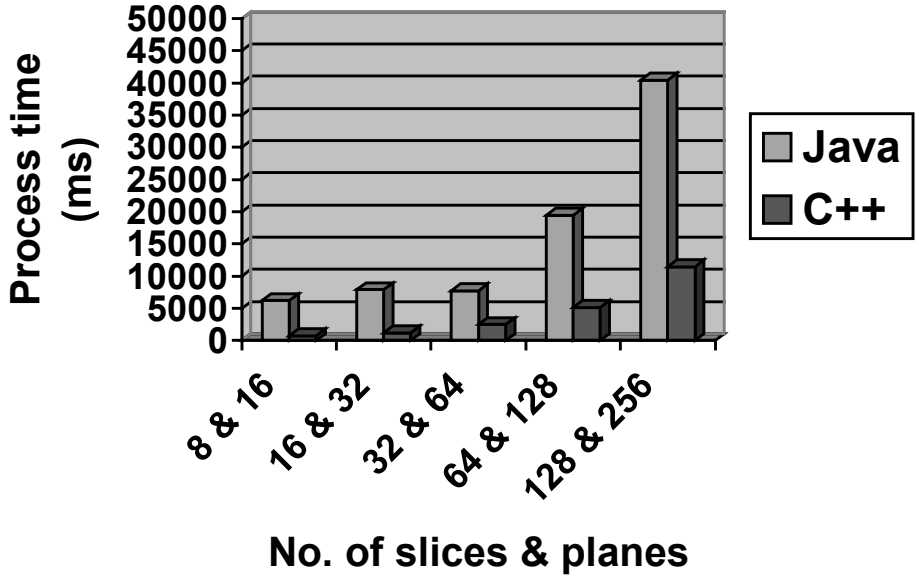


Figure 5.3 Histogram showing timings for rendering a MRI brain on Machine1

Table 5.4 Timings of a MRI brain rendering on Machine2

No. of slices/planes	Loading (ms)		Processing (ms)		Rendering(ms)		Total (ms)	
	Java	C++	Java	C++	Java	C++	Java	C++
8/16	2918	331.7	3172	452.9	151.6	.006	6241.6	784.6
16/32	2929.3	319.0	4817.3	877.9	151.6	.004	7898.2	1196.9
32/64	2800.3	325.6	8367.6	2175.1	167	.005	7769.2	2500.7
64/128	2923.3	336.8	16363	4818.0	187.6	.004	19473.9	5154.8
128/256	2880	344.4	37358	11095.3	224.3	.005	40462.3	11439.7

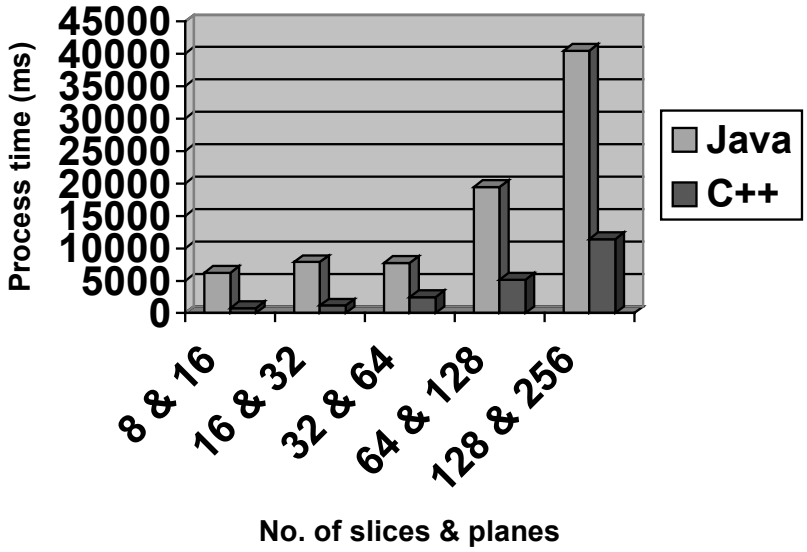


Figure 5.4 Histogram showing timings for rendering a MRI brain on Machine2

Table 5.5 Timings of a CT skull rendering on Machine1

No. of slices/planes	Loading (ms)		Processing (ms)		Rendering(ms)		Total (ms)	
	Java	C++	Java	C++	Java	C++	Java	C++
8/16	1815.6	4.4	14460.3	9958.9	151.6	.005	16427.5	9963.3
16/32	1857.3	4.8	34466.3	21316.0	157.3	.006	336480.9	21321.8
32/64	1857.6	4.8	94608.3	46291.5	168.6	.005	96634.5	46296.3
64/128	1861.6	4.8	275829.3	95996.1	185.6	.005	277876.5	96000.9
128/256	2180	4.9	848489	201215.9	215	.005	850884	201220.8

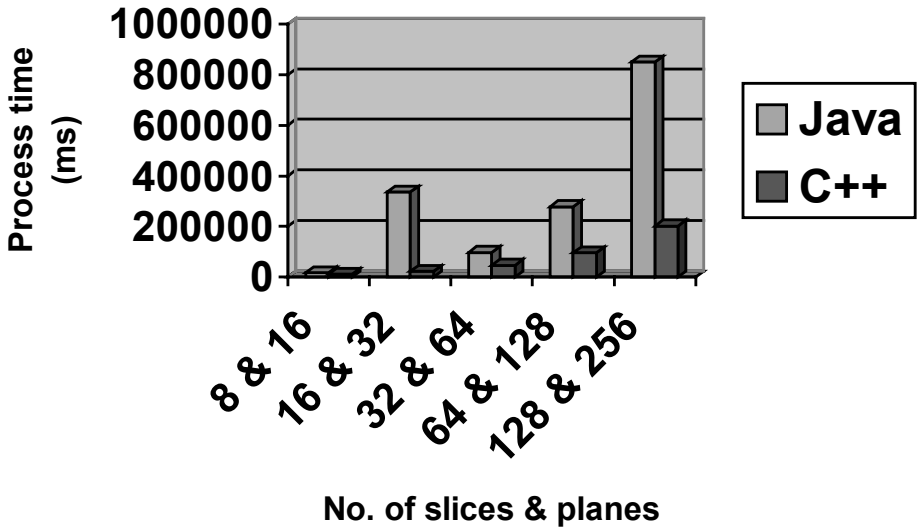


Figure 5.5 Histogram showing timings for rendering a CT skull on Machine1

Table 5.6 Timings of a CT skull rendering on Machine2

No. of slices/planes	Loading (ms)		Processing (ms)		Rendering(ms)		Total (ms)	
	Java	C++	Java	C++	Java	C++	Java	C++
8/16	1609.6	4.8	16093	9687.7	153.3	.004	17855.9	9692.5
16/32	1565.33	4.8	37753	19241.9	157	.004	39475.3	19246.7
32/64	1532.33	4.8	101831	41092.1	165.6	.006	103528.9	41096.9
64/128	1528	4.9	296632	84611.7	186.6	.005	298346.6	84616.6
128/256	1525	4.9	951055	177003.8	234	.005	952814	177008.7

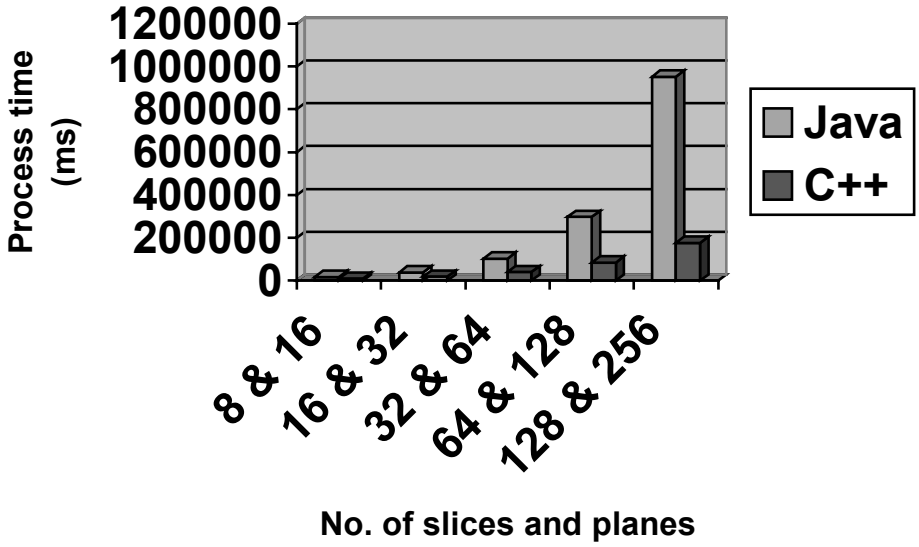


Figure 5.6 Histogram showing timings for rendering a CT skull on Machine2

5.3 Analysis of the timings

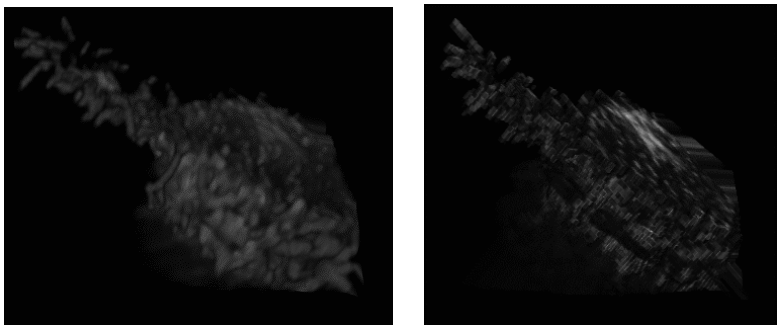
From the above timings it is obvious that OpenGL/C++ is much faster than Java3D/Java. There are several reasons why Java applications tend to be slower than C/C++ applications. The most obvious reason is that Java is an interpreted language and C++ is a natively compiled language. As an interpreted language, the Java source code is initially compiled into a byte code that is stored on the hard disk. At the time of execution, a class loader loads the byte code into memory [1]. The Java Virtual Machine (JVM) interprets the byte code and begins executing the code. This additional layer adds an overhead but introduces a layer of protection and abstraction between the computer hardware and the software [17]. During preprocessing, the Haar wavelet transformation has three loops for transforming the data along each axis. Before the loop is executed, the JVM converts the

loop instructions to byte code and at the time of execution the byte code is once again interpreted for the same number of instruction cycles as were taken while converting the loop instruction into a byte code. This happens whenever loop instructions are encountered [15]. From, the timing shown above it can be noticed that the Java implementation is slower during I/O and rendering. But during pre-processing, the Java implementation sometimes does reasonably better than the C++ implementation. This gain is mainly due to the garbage collector mechanism in Java. In the C++ implementation, each time a temporary array is no longer needed, the memory is de-referenced by using the delete() statement. This adds an extra instruction to the program which needs to be executed every time a memory location is de-referenced. This extra statement is not necessary in case of the Java implementation. However, the garbage collector defers memory deallocation to an undefned point later in time, which could cause some unexpected delays. This effect was not observed during the experiments.

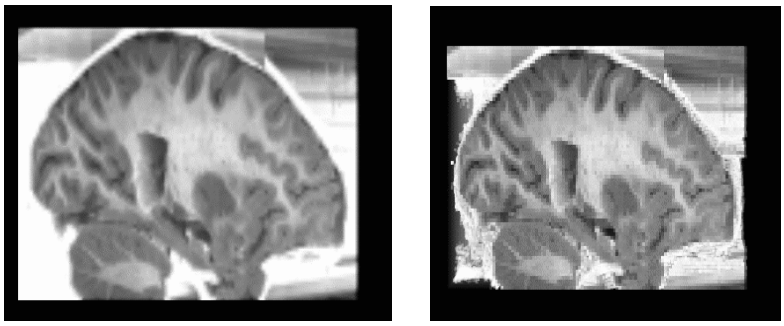
The texture planes are created before the scene graph is rendered. In the case of Java3D, hence, the time the application takes to create the planes is also included in the pre-processing time in the Java implementation. In the C++ implementation, texture planes are created at the time of rendering. During the rendering stage, C++ is much faster because it has the ability to directly interact with the texture hardware using OpenGL libraries. Java3D tends to push its commands to a number of native layers before they reach the texture hardware. Moreover, when Java3D starts rendering, the entire scene graph has to

be traversed. As the number of branches increases, the traversal time tends to increase and so does the rendering time.

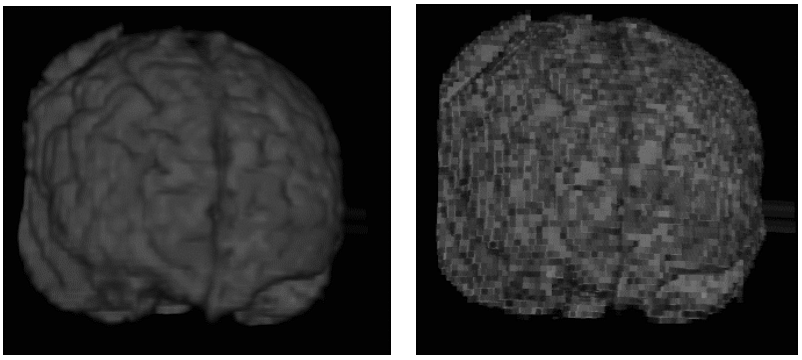
Progressive transmission of the data will tend to decrease the delay in rendering. This is achieved by displaying a low-resolution wavelet image frst, while reconstructing the image later in a progessive way, which leads to a quicker preview image and enables the user to navigate the scene even if the full resolution has not been transmitted yet. Moreover, averaging all the timings, it is observed that the C++ version is faster than Java by only a factor of 3-4 times for the CT brain, while it is only a factor of 1.5 - 2.5 times slower for the cancer cell data set. This factor can be tolerated when platform-independency is a major design factor in the software and it shows that Java and Java3D can be used to implement an interactive, texture-based volume rendering with performance results similar to a C/C++ implementation, which proves the original hypothesis.



Java version C++ version
cell.volc (251 x 70 x 312 elements).



Java version C++ version
brain_stride_8.volc (223 x 144 x 184 elements)



Java version C++ version
brain_stride_8.volc (94 x 113 x 131 elements).

Figure 5.7 Rendered images of Haar-wavelet-transformed data sets

CHAPTER VI

CONCLUSIONS

The wavelet-based rendering system proposed in this thesis aims at enabling web-based access to medical data sets. Through progressive transmission, the user can view a preview of the data set immediately while subsequently the volume is progressively reconstructed to it fullest resolution. Biomedical images need to be represented accurately cannot afford to loose large amounts of detail. The use of wavelets in Java for enabling progressive transmission has proved to provide satisfactory performance results compared to a C++ implementation with no data loss during the transformation and reconstruction (lossless transformation). Quantization can be used to obtain better compression rates, but this is beyond the scope of this work. The purpose of this study was to show that the multiresolution representation of a wavelet-transformed data set is suffcient to compensate for the performance loss due to the use of a virtual machine and an interpreted language rather than a native code implementation in C/C++. The results in chapter V show that the original hypothesis is true and that an interactive, texture-based volume rendering system can be implemented in Java using the wavelet representations to compensate for the loss in the performance and to provide comparable results. The difference in the performance was only a small factor (1.5....4), which proves that the technique proposed in this thesis

works for the given test cases. We expect that the results are scalable and also apply for larger data sets, but further studies would be required to verify the results for larger data sets. Some of the data sets used are larger than core memory, and the effects like memory swapping were observed as described in chapter V.

3-D texture mapping techniques used in this system have produced fat-shaded images. The blending functions used in the system can be varied to produce more realistic images. In the present system, the entire data set is transformed into a low-resolution volume before being transmitted to the client's machine. The $Vistools$ provide a methods to extract sub-volumes. In future versions, this method will be used to extract a particular region-of-interest.

Looking into the pros and cons of Java3D, though speed is a limiting factor for the application, the need to implement web-based technologies will motivate to look into factors that will improve the performance of Java implementations. With the advent of the JIT (just-in-time) compiler for Java, it is possible to translate and store the entire class fle. This eliminates the need for repeated translations of each byte code instruction [15], thus improving the execution time.

In the future, various code optimizations will be considered to make the application even faster. The present implementation of the system supports a single user at a time. In the coming versions of the wavelet based rendering system, web-based user interfaces and servers that support simultaneous access from various sites will be implemented. The system will be integrated with a network module (client/server) in future versions.

REFERENCES

[1] Anonymous, "C++ versus Java," http://www.iusb.edu/ johdougl/ (current September 2002).

[2] Anonymous, "Volume Rendering techniques," http://www.cc.gatech.edu/ scivis/tutorial/linked/volrend.html (current September 2002).

[3] Anonymous, "Volume Rendering vs. Surface Rendering," http://www.cs.ubc.ca/ spider/ladic/volviz.html (current September 2002).

[4] U. Behrens and R. Ratering, "Adding Shadows to a Texture Based Volume Renderer," *IEEE Symposium on Volume Visualization*, Research Triangle Park, NC, October 1998, IEEE Computer Society, pp. 39–46,165.

[5] A. Boggess and F. J. Narcowich, *A First Course in Wavelets with Fourier Analysis*, Prentice-Hall, 2001.

[6] R. A. Cross, "Interactive Realism for Visualization Using Ray Tracing," *Proceeding of IEEE Conference on Visualization*, Atlanta, GA, November 1995, IEEE Computer Society, p. 486.

[7] K. Engel, P. Hastreiter, B. Tomandl, K. Eberhardt, and T. Ertl, "Combining Local and Remote Visualization Techniques for Interactive Volume Rendering in Medical Applications," *Proceeding of Visualization 2000*. IEEE Computer Society, October 2000, pp. 449–452, 587.

[8] H. Fuchs, M. Levoy, and S. M. Pizer, "Interactive Visualization of 3D Medical Data," *IEEE - Computer*, vol. 22, no. 8, August 1989, pp. 46–51.

[9] A. V. Gelder and K. Kim, "Direct Volume Rendering with Shading via Three-Dimensional Textures," *Proceeding of 1996 Symposium on Volume Visualization*. IEEE Computer Society, October 1996, pp. 23–30, 98.

[10] P. Haeberli and M. Segal, "Texture Mapping as a Fundamental Drawing Primitive," Silicon Graphics Incorporated, June 1993.

[11] I. Ihm and R. K. Lee, "On Enhancing the Speed of Splatting with Indexing," *Proceedings of IEEE Conference on Visualization*, Atlanta, GA, November 1995, IEEE Computer Society, pp. 69–76, 441.

[12] I. Kaplan, "Volume Rendering vs. Surface Rendering," http://www.cs.ubc.ca/spider/ladic/volviz.html (current September 2002), August 2001.

[13] J. Kniss, P. McCormick, A. McPherson, J. Ahrens, J. Painter, A. Keahey, and C. Hansen, "Interactive Texture-Based Volume Rendering for Large Data Sets," *IEEE Computer Graphics and Applications*, vol. 21, no. 3, July/August 2001, pp. 52–62.

[14] W. E. Lorensen and H. E. Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," *ACM SIGGRAPH Computer Graphics, Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH:ACM Special Interest Group on Computer Graphics and Interactive Techniques, August 1987, pp. 163–169, ACM Press, NY.

[15] C. Mangione, "Performance tests show Java as fast as C++," http://www.javaworld.com/javaworld/jw-02-1998/jw-02-jperf.html (current September 2002).

[16] M. Meissner, U. Hoffmann, and W. Strasser, "Enabling Classifcation and Shading for 3D Texture Mapping Based Volume Rendering using OpenGL and Extensions," *Proceeding of IEEE Visualization*, D. Ebert, M. Gross, and B. Hamann, eds. IEEE Computer Society, October 1999, pp. 207–526, 24–29.

[17] J. Meyer and T. Downing, *Java Virtual Machine*, chapter 2, O'Reilly and Associates, 1997, p. 9.

[18] T. A. Moeller and E. Haines, *Real-Time Rendering*, second edition, chapter 5, A. K. Perts Ltd., 2002, pp. 119–129.

[19] D. R. Nadeau, "API Architecture," http://vistools.npaci.edu (current September 2002).

[20] D. R. Nadeau, "Volume Scene Graphs," *Proceedings of the 2000 IEEE Symposium on Volume Visualization*, Salt Lake City, Utah, October 2000, SIGGRAPH: ACM Special Interest Group on Computer Graphics and Interactive Techniques, pp. 49–56, ACM Press, NY.

[21] W. B. Pennebaker and J. L. Mitchell, *Jpeg : Still Image Data Compression Standard*, 1 edition, Kluwer Academic Publishers, September 1992.

[22] C. Rezk-Salama, P. Hastreiter, C. Teitzel, and T. Ertl, "Interactive Exploration of Volume Line Integral Convolution Based on 3D-Texture Mapping," *Proceeding of IEEE Visualization*, San Fransisco, CA, October 1999, IEEE Computer Society, pp. 233–528.

[23] S. Saha, "Image Compression-from DCT to Wavelets: A Review," http://www.acm.org/crossroads/xrds6-3/sahaimgcoding.html (current September 2002), January 2000.

[24] S. Saladi, *Texture Based Volume Rendering with Illumination using Java3D*, master's thesis, Mississippi State University, September 2002.

[25] C. W. Sensen, J. Brum, P. Gordon, M. Hood, G. Lindahl, M. Schulman, C. Spindler, M. Stuart, and S. Unger, "Establishment of the frst Java3D enabled CAVE," http://www.java.sun.com/products/java-media/ 3D/calgary.whitepaper.pdf (current September 2002).

[26] E. J. Stollnitz, T. D. Derose, and D. H. Salesin, "Wavelets for Computer Graphics: A Primer Part 1," *IEEE Computer Graphics and Applications*, vol. 15, no. 3, May 1995, pp. 76–84.

[27] J. E. Swan, K. Mueller, T. Moeller, N. Shareef, R. Crawfs, and R. Yagel, "An Anti-Aliasing Technique for Splatting," *Proceeding of IEEE Visualization 97*. IEEE Computer Society, October 1997, pp. 197–204, 544.

[28] R. Westermann and T. Ertl, "Effciently Using Graphics Hardware in Volume Rendering Applications," *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH: ACM Special Interest Group on Computer Graphics and Interactive Techniques, July 1998, pp. 169–177, ACM Press, NY.

[29] M. Woo, J. Neider, T. Davis, and D. Shreiner, *OpenGL Programming Guide*, Addison-Wesley, 1999.