MULTI-DIMENSIONAL VOLUME RENDERING FOR PC-BASED MEDICAL SIMULATION

ZHENLAN WANG

NATIONAL UNIVERSITY OF SINGAPORE

2005

MULTI-DIMENSIONAL VOLUME RENDERING FOR PC-BASED MEDICAL SIMULATION

ZHENLAN WANG
(B.Eng, Xian Jiaotong University)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2005

Acknowledgements

I am grateful to many people for their help and support in the course of this research. First of all, I would like to express my sincerest gratitude to my supervisor Dr. Ang Chuan Heng for his patient guidance and constructive advice throughout the duration of my research. I would also like to express my deepest appreciation to my co-supervisors, Prof. Teoh Swee Hin from Dept. Mechanical Engineering, NUS and Prof. Wieslaw L. Nowinski from Biomedical Imaging Lab, for their guidance and support.

I would like to take this opportunity to give special thanks to Dr. Chui Chee Kong for his countless encouragement and valuable advice at key times, without which this research cannot be completed.

In addition, I would like to thank my colleagues and friends, Hua Wei, Chen Xuesong, Li Zirui, Yang Yanjiang and Jeremy Teo, in the I^{2}R, BIL and VSW group for their friendship and help in both my work and life.

Special thank also goes to Dr. Goh P.S. and Mr. Christopher Au of National University Hospital (NUH), Singapore for the dynamic MRI data and Prof. J.H. Anderson of Johns Hopkins University School of Medicine, USA for the phantom head data, and their medical advice.

I would like to express my gratitude to the National University of Singapore for providing me with the scholarship in the early years of this research.

Finally, I would like to thank my parents and my wife for their love and encouragement. I dedicate this dissertation to them.

Contents

Summary

Four-dimensional volume rendering is a method of displaying a time-series of volumetric data as an animated two-dimensional image. With the development of diagnostic imaging technology, the contemporary medical modalities not only can image the internal organs or structures of a human body in more and more details, but are also able to capture the dynamic activity of a human body over a period time. Visualization of the four-dimensional/time-varying volume data is meaningful for clinicians for better diagnosis and treatment but it also poses a new challenge to the computer graphics technology due to the tremendous increase in the size of data and computational expense. Therefore, there is an urge to seek for a cost effective solution for this task.

This thesis describes two new four-dimensional volume rendering algorithms. Both of them are characterized by using a data decomposition technique to take advantage of the four-dimensional features of time-varying volume data, while they also have their distinct advantages. For the first method, a new data structure called dynamic linear level octree is proposed for efficient rendering. It is effective in exploiting both the spatial and temporal coherence of time-varying data. The second method explores more extensively on ways to reduce the space requirement and uses global coherence to achieve higher performance. The variants of the two algorithms in thread-level parallelism also increase their potential in performance improvement and the scope of applications. In comparison with conventional rendering methods, both algorithms are superior in terms of both speed optimization and space reduction. The two algorithms have also been successfully used in our medical simulation systems to provide interactive and real-time four-dimensional volume rendering on personal computers.

List of Tables

List of Figures

Publication

- **Journal Articles**

WANG, Z.L., CHUI, C.K., CAI, Y.Y., ANG, C.H. AND TEOH, S.H. 2005, Dynamic Linear Level Octree-Based Volume Rendering Methods for Interactive Microsurgical Simulation, to appear *International Journal of Image and Graphics*.

WANG, Z.L., TEO, J.C.M., CHUI, C.K., ONG, S.H., YAN, C.H., WANG, S.C., WONG, H.K. AND TEOH, S.H. 2005, Computational Biomechanical Modeling of the Lumbar Spine Using Marching-Cubes Surface Smoothened Finite Element Voxel Meshing, *Computer Methods and Programs in Biomedicine*, 80, 1, 25 – 35.

WANG, Z.L., ANG, C.H., CHUI, C.K. AND TEOH, S.H. 2005, A Clustering-Based Algorithm for Fast Time-Varying Volume Rendering, Submitting for publication.

MA, X., **WANG, Z.L.**, CHUI, C.K., ANG, JR. M.H., ANG, C.H. AND NOWINSKI, W.L. 2002, A Computer Aided Surgical System, *Computer Aided Surgery (CAS)*, 7, 2, 119.

CHUI, C.K., LI, Z., ANDERSON, J.H., MURRPHY, K., VENBRUX, A., MA, X., **WANG, Z.L.**, GAILLOUD, P., CAI, Y., WANG, Y. AND NOWINSKI, W.L. 2002, Training and Planning of Interventional Neuroradiology Procedures - Initial Clinical Validation, *Studies in Health Technology and Informatics*, 85, 96 – 102.

- **Conference Articles**

WANG, Z.L., CHUI, C.K., CAI, Y.Y. AND ANG, C.H. 2004, Multidimensional Volume Visualization for PC-Based Microsurgical Simulation System, *Proceedings of ACM SIGGRAPH International Conference on Virtual Reality Continuum and its Applications in Industry (VRCAI)*, 309 – 316.

YANG, Y., **WANG, Z.L.**, BAO, F. AND DENG, R.H. 2003, Secure the Image-based Simulated Telesurgery System, *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS)*, 596 – 599.

WANG, Z.L., CHUI, C.K., ANG, C.H., LI, Z. AND NOWINSKI, W.L. 2002, Shear-Warp Volume Rendering Algorithm using Linear Level Octree for PC-based Medical Simulation, *Proceedings of International Conference on Medical Imaging Computing and Computer Assisted Intervention (MICCAI)*, LNCS, 2489, 2, 606 – 614.

WANG, Z.L., CHUI, C.K., CAI, Y., ANG, C.H. AND NOWINSKI, W.L. 2002, Fast PC-based Visualization Algorithms for Virtual Reality Simulation of Microsurgical Procedures, *Proceedings of International Conference on Biomedical Engineering (ICBME)*.

CHUI, C.K., TEO, J., TEOH, S.H., ONG, S.H., WANG, Y., LI, J., **WANG, Z.L.**, ANDERSON, J.H. AND NOWINSKI, W.L. 2002, A Finite Element Spine Model from VHD Male Data, *Proceedings of VHD Conference*.

CAI, Y., CHUI, C.K., WANG, Y., **WANG, Z.L.** AND ANDERSON, J.H. 2001, Parametric Eyeball Model for Interactive Simulation of Ophthalmologic Surgery, *Proceedings of Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, LNCS, 465 – 472.

WANG, Z.L., MA, X., ANG, M.H. JR., CHUI, C.K., ANG, C.H. AND NOWINSKI, W.L. 2001, A Virtual Environment-Based Practical Surgery System, *Proceedings of Asian Conference on Robotics and its Applications*, 69 – 73.

HUA, W., CHUI, C.K., WANG, Y., **WANG, Z.L.**, CHEN, X., PENG, Q. AND NOWINSKI, W.L. 2000, A Semiautomatic Framework for Vasculature Extraction from Volume Image, *Proceedings of International Conference on Biomedical Engineering*, 515 – 516.

Chapter 1

Introduction

Visualization is the use of computer-generated media based on data in the service of human insight/learning.

— Carol Hunter

1.1 Background

Medical training is a lengthy and sophisticated process. For example, a clinical fellow will need at least seven years of practical training in a reputable hospital to become an interventional radiologist [Anderson et al. 2002]. The long duration may be attributed to the limited opportunities there must be available for trainees to learn and exercise their skills before practice in clinical routines. The risk of failed operations on living patients exists. With rapid advancement of virtual reality technology, a simulation system for the purpose of training and pre-treatment planning based on patient-specific medical images is becoming possible by using state-of-the-art computing technologies. In medicine, visual information plays an essential role for accurate diagnosis and effective therapy planning. Approximately 80% of all information perceived by human is through the eyes, while the visual system of humans is the most complex of all sensory modalities [Demiris et al. 1997]. Visualization thereof is critical in the medical simulation systems as surgeons perform operations and make decisions mostly based on visual cues.

We want to design a low cost medical simulator for image-guided procedures that can be comfortably placed on the desktop of medical personnel. Therefore, it is expected that the visualization solution can work effectively and efficiently on standard personal computers. It should be based on medical images of a patient, and a visual environment that resembles patient-specific surgical scenario provides realistically.

In this thesis, I propose multi-dimensional visualization solutions, including three-dimensional (3D) and four-dimensional (4D) rendering, for the PC-based medical simulation systems. Parallel processing and hardware-accelerated methods of visualization for full view rendering are also discussed.

1.2 Medical Image Modalities

Medical images are the source for medical visualization. Medical imaging makes it possible for us to investigate an area of patient body that is usually not visible. There have been many attempts to visualize the interior of the human body [Lichtenbelt et al. 1998]. Advancement in medical imaging creates various medical modalities such as computed tomography, magnetic resonance imaging and ultrasonography that are widely used for different diagnostic and therapeutic purposes.

Computed Tomography (CT) is used to obtain a series of 2D grayscale images depicting a cross section of the body parts under examination. Figure 1.1, as an example, shows a set of 2D CT scan images of VHD[1] head dataset. As the CT tube revolves around the patient, multiple X-ray images are taken. The system calculates the amount of X-ray penetration through the specific plane of the body parts examined, and gives each a numeric value. This information is then used in the reconstruction of images. Therefore, CT images have advantages over conventional X-ray images in that they contain information from individual plane. A conventional X-ray image, on the other hand, contains aggregated information from all the planes, and the result is the accumulation of shadows that is a function of the density of the tissues, bones, organs and anything that absorbs the X-rays [Pawasauskas 1997]. CT scanning has been commonly used to obtain a detailed view of internal organs.

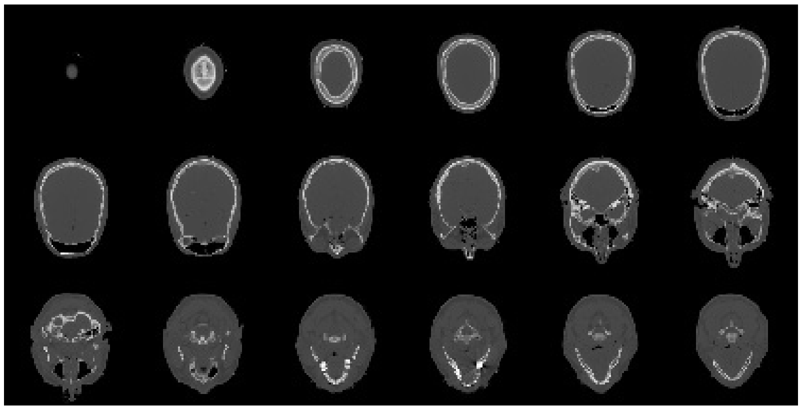


Figure 1.1 CT scan images of VHD head

[1] The Visible Human Dataset (VHD) provides complete visual insight of the entire human body. The Visible Human Project, *http://www.nlm.nih.gov/research/visible/visible_human.html*, National Library of Medicine.

Magnetic Resonance Imaging (MRI) is another common modality for non-invasive imaging of the body, particularly the soft tissues. It uses strong magnetic field and radio waves to alter the natural alignment of hydrogen atoms within the body. Computers monitor and record the summation of the spinning energies of the hydrogen atoms within living cells and translate that into images. MRI offers increased-contrast resolution, enabling better visualization of soft tissues, brain, spinal cord, joints and abdomen. It can selectively image different tissue characteristics [Riederer 2000]. MRI also allows for multi-planar imaging, as opposed to conventional CT, which is usually only axial. MRI provides highly detailed information without exposing the body to radiation.

The other common modalities are ultrasound and nuclear imaging. Ultrasound imaging uses high frequency sound waves that are reflected by tissue at varying rates to produce images. It images muscle and soft tissue very well and is particularly useful for delineating the interfaces between solid and fluid-filled spaces. An example application of this imaging is the examination of pregnancy. Nuclear medicine imaging systems, such as Single Photon Emission Computed Tomography (SPECT) and Positron Emission Tomography (PET), image the distribution of radioisotopes and provide a direct representation of metabolism, function in the organ or structure in the body [Robb 1995; Dev 1999].

In recent years, a wide scope of advanced medical imaging techniques such as dynamic magnetic resonance imaging (dMRI), functional magnetic resonance imaging (fMRI) and dynamic computer tomography (dCT) has been introduced into the biomedical practice. They are characterized to capture motions or changes of investigated organs or structures over a period of time. dMRI and dCT allow the radiologist to acquire sequential image scan data with a contrast agent in short intervals repeatedly. fMRI is used to register the blood flow to functioning areas of the brain so that functions of the brain such as speech or recognition can be monitored as they occur. These dynamic modalities are important resources for multi-dimensional imaging research.

Other medical imaging modalities include Diffuse Optical Tomography (DOT), elastography, Electrical Impedance Tomography (EIT) and so on. These techniques are mainly in research and yet to be deployed in clinical practice.

1.3 Visualization of Medical Images

The medical imaging techniques are characterized to produce static two-dimensional (2D) slice images of body parts (with the support of image reconstruction technique). Experienced medical personnel normally are required to interpret these slices. However, it is very difficult for people to reconstruct the highly complicated 3D anatomical structures mentally based on the 2D slices. Mental reconstruction is difficult and highly subjective as different people mentally reconstruct different shapes. The visual interpretation of dynamic/time-series datasets is an even harder process. Therefore, visualization of those medical modalities in 3D or higher to reveal the real appearance of the anatomical objects is necessary. With the ability to visualize important structures in great detail, visualization methods are valuable resources for the basic biomedical research, the diagnosis and surgical treatment of many pathologies.

Since visualization of medical images in higher dimensions is important, many methods and approaches have been attempted by researchers and scientists over the last two decades. The

2D medical images (e.g., Figure 1.1), organized as a stack of slices in a regular pattern (e.g., one slice every millimeter) that occupies a 3D region of space, is referred to as volume images/dataset (e.g., Figure 1.2). A collection of volume images scanned at a sequence of time steps builds up a 4D volume dataset. The additional dimension referred in multi-dimensional volume datasets is typically associated with the time, and the 4D dataset is called a *time-varying volume dataset* as well. The visualization of volume dataset is then termed *volume visualization*.

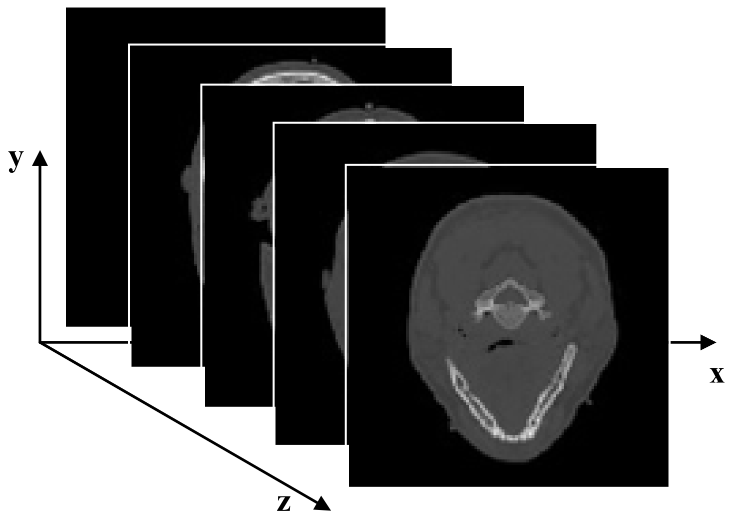


Figure 1.2 Organization of images as a volume dataset (CT scan of VHD head)

One of the most attractive and fast-growing areas in volume visualization is *volume rendering*. Volume rendering is often called direct volume rendering as well. It is the process to create high-quality images by directly projecting data elements (called *voxel*s) defined on multi-dimensional grids onto the 2D image plane for the purpose of gaining an understanding of the structure contained within the volumetric data [Elvins 1992]. The above VHD head dataset is visualized by a volume renderer in two different effects (Figure

1.3). Although 2D CT images are useful in diagnosis, the volume rendered images appear to be more natural and easier to comprehend the whole anatomy by human being.

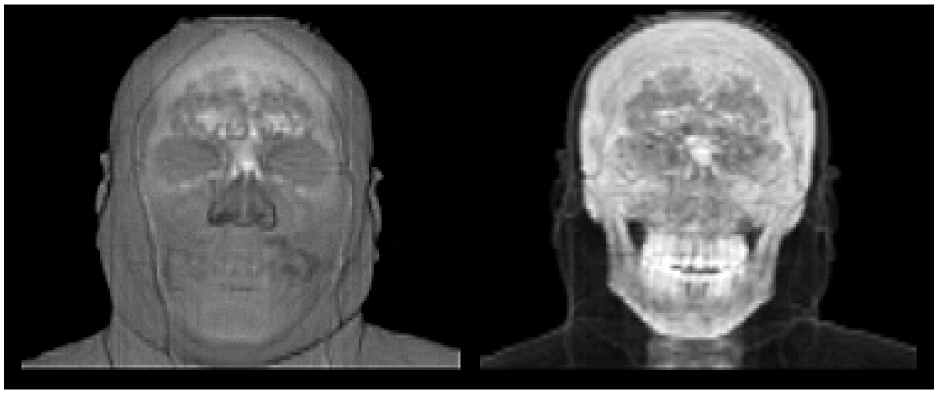
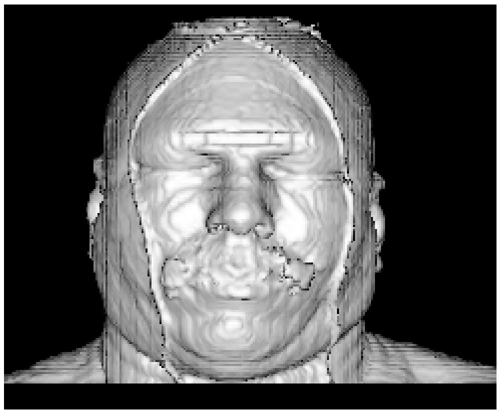


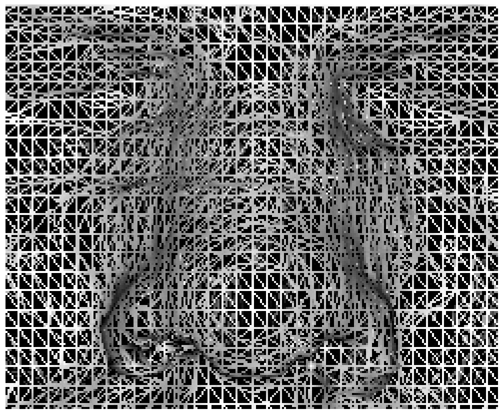
Figure 1.3 Volume rendering images produced from a CT scan of a VHD head

To reveal or hide different structures in a volume, we can assign different transparencies to voxels during volume rendering (called *classification*). This assignment is a function of the properties of a voxel such as its intensity or gradient magnitude. The function is called opacity transfer function, which can have any number of parameters as its input. As we know, the gradient magnitude tends to be high at object boundaries. By using this character, for example, the right image in Figure 1.3 demonstrates the result of an opacity transfer function with the involvement of gradient magnitude while the left image is produced by the opacity transfer function considering only voxel intensities. To enhance visual understanding of volume data, we can also map voxel intensities to colors (called *coloring*). Normally, three color transfer functions are used, one transfer function each for red, green and blue. If they were the same, a gray scale image would be produced. We can assign different colors to different features for meaningful interpretation of volume data. Similar to the opacity transfer function, color transfer functions can also be a function of any voxel properties and not restricted to voxel intensities. With these four transfer functions and together with other functionalities, volume rendering appears to be powerful in visualization of volumetric data.

Besides volume rendering, extracting and generating geometric models from the volume images is another technique, named *surface rendering*, which is frequently used for volume visualization. Geometric primitives are generated at object boundaries in the volume dataset and they are stitched together to obtain a surface representation. The volume dataset is then indirectly visualized as polygonal meshes with traditional polygon rendering techniques. The marching-cubes algorithm [Lorensen and Cline 1987] is a common technique for extracting a surface, typically called *iso-surface*, from volume data. Figure 1.4(a) shows such an iso-surface extracted from the VHD head dataset by the marching-cubes algorithm. A magnified view of the surface mesh in the region of the nose is shown in Figure 1.4(b) that the triangular meshes can be clearly identified.



(a) Marching-cubes iso-surface



(b) Polygonal mesh

Figure 1.4 Surface rendering images produced from a CT scan of a VHD head

Multimodality visualization is an important branch of volume visualization providing additional valuable insights of medical images. With the development of medical image acquisition techniques, rich modalities of medical imaging data are available, and they are adept at presenting different tissues or structures in human body. It is desirable to visualize multiple volume images with different modalities of the same object into a single image to get more comprehensive information about desired structures. For instance, because bone is best captured in CT, while MRI is adept in soft tissue structures, CT and MRI are often used in conjunction with one another to produce images with more complete information of examining structures. This technique is called *multimodality rendering*. Both volume rendering and surface rendering techniques can be used for multimodality rendering.

1.4 Volume Rendering versus Surface Rendering

The volume-based visualization approach has many advantages over the surface-based method in several aspects, especially in the area of medical applications.

Volume rendering algorithms are characterized by mapping elements of volumetric data directly into image space without using geometric primitives as an intermediate representation [Elvins 1992]. Since the whole volume of data is represented, the methods potentially provide visual access down to the smallest detail of the internal composition, not just the outer shell of the object being investigated. In medical applications, volume-based models have advantages over surface-based models, in that many important features of the data are lost during surface modeling. In addition, as compared to surface rendering, volume rendering algorithms never need to explicitly determine the surfaces of fuzzy objects contained in the volume, which, however, occurs frequently in medical imaging. On the other hand, since possibly all data in the volume can contribute to the final representation, the immense size of data increase the computation time significantly [Kaufman et al. 1993]. The input data for volume rendered images in Figure 1.3, as an example, contains 5.5 million samples, and fast rendering such quantity of data makes a high demand of computation power and memory bandwidth.

A surface rendering algorithm typically fits surface primitives such as polygons or patches to constant-value contour surfaces found in volumetric datasets [Elvins 1992]. Therefore, before visualization, it is required to extract constant-value surfaces from the volume data. These surfaces can be rendered using traditional geometric rendering techniques. Because the surface extraction procedure is performed only once in data preprocessing stage and subsequently the surface primitives can be used repeatedly for rendering, surface rendering algorithm is typically much faster than volume rendering. However, if there are any changes to the surface criteria, then all the volume data have to be re-traversed and a set of new surface primitives has to be extracted. Such extraction procedure is time consuming. For example, the surface model in Figure 1.4 contains more than 150 thousand triangular patches in total.

In addition to all the advantages of volume rendering, its capability to produce high-quality and detailed images attracts us to use it as the fundamental visualization solution in our medical simulation systems. To implement multi-dimensional volume rendering on a standard personal computer, I improved the approaches to make the computation in volume rendering less intensive. I also explored its potential benefits in medical field to provide a real-time, interactive, flexible, and fully controlled volume rendering for medical simulation.

1.5 Organization

Chapter 2 begins a literature review of existing diversity of volume rendering algorithms and their improved techniques in both 3D and 4D. The survey is presented by highlighting the advantages and disadvantages of each class of the methods.

In Chapter 3, I first describe the spatial data structure used for accelerated 3D volume rendering. Based on it, a new data structure, dynamic linear level octree, and its corresponding algorithms are presented, which forms the basis of one of my solutions for 4D volume rendering [Wang et al. 2005a].

Chapter 4 presents the other solution of mine for 4D volume rendering. I describe a clustering technique to explore the 4D volume data. A new encoded dataset is produced for fast 4D rendering. This method exhibits some advantages over other methods proposed previously.

Chapter 5 discusses the parallelization problems of the two proposed 4D volume rendering methods. Although these methods are initially designed to be implemented on normal personal computers, the parallelization can further improve their performance and are possible to be used for rendering of even larger datasets.

Chapter 6 reviews the use of volume rendering in medicine and demonstrates the application of the proposed algorithms in several medical simulation systems to provide interactive and real-time 4D volume rendering on personal computers. The medical simulation systems are meant for image-guided surgeries.

Chapter 7 discusses the contributions of the proposed methods and compares them with other existing techniques.

Finally, Chapter 8 concludes this work and discusses the research work that can be done in future.

Chapter 2

Volume Rendering - Literature Review

2.1 Introduction

Researchers in volume rendering have made significant progress during the last two decades. About ten years ago, it would take couples of minutes to render a medium size volume into an image on a Silicon Graphics (SGI) workstation. However, at present, a high quality volume rendered image can be produced in just a few tens of milliseconds on a personal computer. Besides the contribution of the rapid development of computing hardware, many successful researches in volume rendering algorithms have been proposed to improve both the quality and the speed of rendering. Some of them are even commercialized and implemented into the specialized hardware, and some become hot research topics and are propelling a revolution in the design of graphics processing unit (GPU).

With the rapid development of modern medical and scientific imaging technology, conventional 3D volume rendering techniques can not satisfy the demands of visualization of large-scale and time-sequence volume datasets newly come forth. Volume rendering of 4D or time-varying volume datasets attracts many researchers from steady-state volume rendering and becomes one of the popular research fields.

It will be too lengthy to summarize all the volume rendering works here, so we will only focus on the most representative 3D volume rendering methods and improved techniques. In addition, 4D volume rendering is still in its infancy and state-of-the-art research attempts will also be reviewed.

2.2 Mathematical Models for Volume Rendering

Volume rendering is based on the physics of light propagation through particles in a volume. Blinn [1982] and Kajiya & Herzen [1984] did the early research work in this field. Since the aim of volume rendering is to visualize the volume data, not to mimic the exact physics, the mathematical models are simplified with assumptions of voxel behavior in interaction with lights. The mathematical models of volume rendering introduced in this section are mostly the basis of the ray-casting algorithm. However, the methods or concepts such as front-to-back/back-to-front composition, *over* operator, illuminations etc. are also the fundamentals of other volume rendering algorithms. They also play an important role in my proposed 4D volume rendering algorithms. The ray-casting algorithm will be introduced in more detail in a later section.

The mathematical model of volume rendering simulates the procedure that, with the interaction of light, samples in the volume along one viewing ray are taken and integrated to form the color of a pixel. Following is the most popular volume rendering equation (also called volume rendering integral) used in the ray-casting algorithm today [Lichtenbelt et al. 1998].

$$I(a,b) = \int_a^b g(s) \cdot e^{-\int_a^s \tau(x)dx} ds \qquad \textbf{(2.1)}$$

I(*a,b*) is the integrated intensity of one pixel. *g*(*s*) describes the illumination model used in ray-casting. *τ*(*x*) defines the rate that light is occluded per unit length due to scattering or extinction of light. *g*(*x*) and *τ*(*x*) are used to map a voxel *x*'s value into its intensity and opacity. *s* is the segment of the ray that intersects with the volume.

To compute *I*(*a,b*), the integral in Equation 2.1 is discretized (with approximation) into two equivalent formats, which lead to two famous compositing methods, namely front-to-back (FTB) compositing and back-to-front (BTF) compositing.

In front-to-back compositing, the volume rendering equation can be written as:

$$I(a,b) = \sum_{i=0}^{n} I_i \prod_{j=0}^{i-1} (1-\alpha_j) \qquad \textbf{(2.2)}$$

or recursively:

$$\begin{aligned} I_{out} &= I_{in} + (1-\alpha_{in})I_i \\ \alpha_{out} &= \alpha_{in} + \alpha_i(1-\alpha_{in}) \end{aligned} \qquad \textbf{(2.3)}$$

where *I* is the intensity, *α* is the opacity, *in* is the composited value up to current sample point *i*, and *out* is the result after the composition of current sample. The intensity *I* of a sample point is different from its color. In this thesis, we adopt the following relationship between the intensity and color, i.e., the intensity of a sample point is the product of the color and opacity of that sample point:

$$I_i = C_i \cdot \alpha_i \qquad \textbf{(2.4)}$$

where C_i could be red, green or blue color component of the sample point. Thus, we can rewrite the FTB compositing formula (Equation 2.3) into the color representation by replacing the intensity of sample points with color:

$$C_{out} = C_{in} + (1 - \alpha_{in})\alpha_i C_i \qquad \textbf{(2.5)}$$

Samples are accumulated along the viewing ray from the entering point to the exiting point in the volume, or from front to back. The opacity increases while samples are composited. When the opacity stored in the pixel approaches unity, the remaining samples will contribute very little to the pixel, and therefore do not need to be processed. This technique is called early ray termination.

Equation 2.2 can be rewritten as follows:

$$\begin{aligned} I(a,b) &= \sum_{i=0}^{n} I_i \prod_{j=0}^{i-1} (1-\alpha_j) \\ &= I_0 + I_1(1-\alpha_0) + I_2(1-\alpha_0)(1-\alpha_1) + \ldots + I_n(1-\alpha_0)\cdots(1-\alpha_{n-1}) \qquad \textbf{(2.6)} \\ &= I_0 over I_1 over I_2 over \cdots over I_n \end{aligned}$$

The *over* operator was first introduced in [Porter and Duff 1984]. With the *over* operator, it is possible that we divide the volume into two or more parts along the ray, visualize each part individually, and finally compose all the intermediate images together with the *over* operator. The result is the same as that achieved in rendering the whole volume. Thus intensive computation of volume rendering can be distributed to multiple computational resources and work in parallel for better performance.

Equation 2.5 is computationally efficient in that it avoids multiplications between the opacities and the colors of the input and output pixels repeatedly. However, it is not compatible with the *over* operator. Pixels from different intermediate images cannot be composited correctly with Equation 2.5. Instead, the following equation is used when we composite multiple intermediate images.

$$C_{out} = C_{in} + \left(1 - \alpha_{in}\right)C_i \tag{2.7}$$

where C_{in} is the composited pixel color up to current intermediate image, C_i is the pixel color of the current intermediate image and the C_{out} is the result color of the composited pixel.

In the back-to-front composition, the volume rendering equation is written as:

$$I(a,b) = \sum_{i=0}^{n} I_i \prod_{j=i+1}^{n} (1 - \alpha_j) \tag{2.8}$$

or a recursive representation:

$$I_{out} = I_i + I_{in}(1-\alpha_i) \qquad \textbf{(2.9)}$$

In this method, samples are accumulated from back to front. Note that in Equation 2.9, we do not need to keep track of the accumulated opacities any more, and hence it reduces the computational task. However, early ray termination is no longer possible either. The color representation of the recursive BTF compositing formula is:

$$C_{out} = \alpha_i C_i + (1-\alpha_i)C_{in} \qquad \textbf{(2.10)}$$

Unlike the FTB compositing formula, Equation 2.10 can be used for the composition of both sample points and pixels from intermediate images.

Since volume rendering simulates the physics of the interaction of lights and volume elements, it is necessary to include the illumination models. The Phong model [Phong 1975] is one of the often used illumination models for volume rendering. The Phong illumination model counts the contribution of ambient, diffuse and specular reflection, and mathematically it is written as:

$$I_\lambda = K_a C_{a\lambda} I_{a\lambda} + \sum_{i=1}^{m} f(d_i) I_{p_i\lambda}[K_d C_{d\lambda}(\vec{L}\cdot\vec{N}) + K_s C_{s\lambda}(\vec{H}\cdot\vec{N})^n] \qquad \textbf{(2.11)}$$

I_λ is the result intensity of the investigated point after the illumination of m point-lights with wavelength of λ (for red, green and blue color components). α, d and s represent for ambient, diffuse and specular reflection respectively. K is a material-property-based reflection coefficient, C is the light color and I is the light intensity.

2.3 Three-Dimensional Volume Rendering

According to the means of mapping volume data from the object space to the image space, 3D volume rendering techniques can be divided into two groups. The first group consists of *object-space methods* or *forward rendering methods*. The methods intuitively transform each volume element (voxel) separately from the object space to the image space and then project it onto the image plane/screen. In contrast to an object-space method, an *image-space method* or *backward rendering method* is characterized in transforming the entire volume data to image space, and performing volume sampling for each image element (pixel). Image-space methods are regarded superior over object-space methods since they can be extended to support global illumination and volume deformation [Yagel 1999].

2.3.1 Fundamental 3D Volume Rendering Algorithms and Optimizations

Ray-casting algorithm, splatting algorithm and shear-warp algorithm are the milestones in the history of volume rendering development. They are distinguished either in performance or in image quality characterized as the category of algorithms with no assistance of parallel computing or hardware acceleration. In the following, I will outline these algorithms and their related optimizations.

- Ray-casting algorithm

Ray-casting is the most often used volume visualization algorithm for the generation of high-quality images, and has been seen the largest body of publications over the years [Tuy 1984; Upson and Keeler 1990; Levoy 1988; Levoy 1990a; Levoy 1990b].

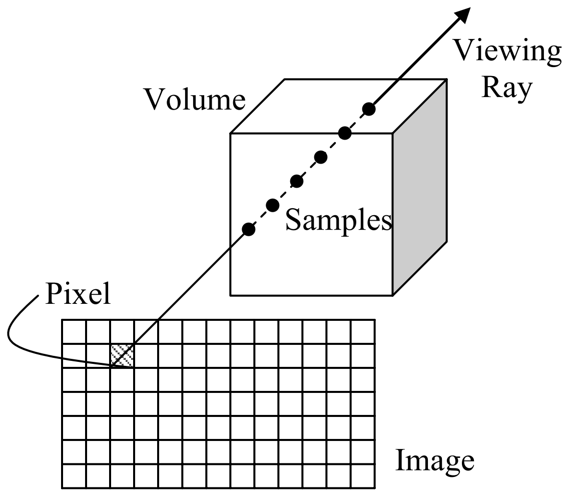


Figure 2.1 Schematic diagram of ray-casting model

Figure 2.1 illustrates a simplified model of ray-casting algorithm. Ray-casting algorithm conducts an image-space rendering. An image is produced by casting rays through each pixel of the image plane into the volume data and accumulating the color and opacity along the ray. In order to be evaluated by the computer, the continuous contribution of the ray is discretized, and samples are taken along each ray within the portion where it intersects with the volume. Each of the samples will gather the contributions of its surrounding voxels. Finally, the sample values are accumulated to the pixel that the ray was fired through.

Figure 2.2 gives a flow chart, which shows the operations applied to each sample. In this figure, the left column gives the process applied to each sample, and the right column indicates actions performed by users in a typical ray-casting session. Firstly, in re-sampling and gradient estimation stages, the intensity and gradient of a sample are interpolated among its neighboring voxels. A user can choose interpolation methods varying from zero order (nearest neighbor) to higher order. Tri-linear interpolation (first order) is generally selected for its small computation but satisfactory results produced in most cases. The gradient can also be calculated based on different estimators ranging from intermediate difference operator, central difference operator to Sobel 3D operator. With the interpolated intensity, in the classification stage, the sample is separated into different feature classes, typically done by assigning different colors and opacities to the sample based on transfer functions. Normally, a predefined color lookup table and an opacity lookup table are employed as the transfer functions for efficient classification. Subsequently in the shading stage, the sample is shaded according to its color and gradient by using an illumination model, typically a Phong illumination model. The Phong model is a local illumination model and is largely an empirical model. However, it is fast to compute and gives reasonably realistic results. Finally, in the composition stage, the sample value, including color and opacity, is accumulated to the pixel that the ray was fired through. The process for this sample is completed.

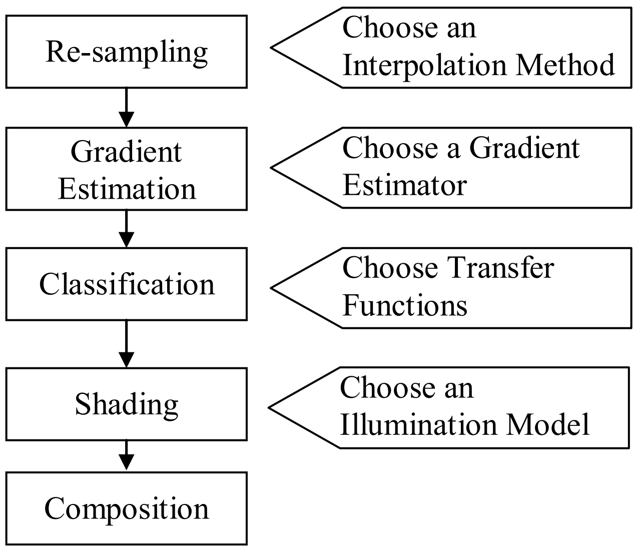

Re-sampling
Choose an Interpolation Method
Gradient Estimation
Choose a Gradient Estimator
Classification
Choose Transfer Functions
Shading
Choose an Illumination Model
Composition

Figure 2.2 Flow chart of sample processing in the ray-casting algorithm

This process will be applied iteratively to the next sample point, which is taken at a step further along the ray. The sample points contributing earlier are weighted heavier than later sample points. The pixel summing continues until the stored opacity value is close to unity for *early ray termination*, or the ray exits the volume. The idea behind early ray termination is that if a ray passes through a dense object, the scene behind that point will contribute little to the final image and thus a ray can be terminated as soon as it has accumulated to be opaque enough for a user defined opacity threshold.

Ray-casting can produce high-quality, colored, and shaded images. However, because of the immense size of volume data, the ray-casting algorithm is very time consuming. It is traditionally only for image generation as it is not suitable for interactive applications. Many approaches have been proposed to speed up the process of ray-casting algorithm.

We note that, in the ray-casting algorithm, even when rays enter into empty space, samples still must be taken and composited, which, however, do not contribute to the final image. Therefore, skipping the empty space, termed *space-leaping*, is the major approach to accelerate the ray-casting algorithm without sacrificing the image quality.

In the Bounding-box algorithm [Avila et al. 1992], one of space-leaping approaches, the objects in the volume are tightly surrounded with boxes or spheres. Only rays intersecting with the bounding objects will be considered during rendering. In proximity cloud algorithm [Cohen and Shefer 1993; Freund and Sloan 1997], all the voxels occupied by objects are surrounded with a layer of one-voxel-deep cloud voxels. So the rays can rapidly skip the empty space until they come into the cloud layer. Based on this idea, distance-coding algorithm [Sramek 1994] skips empty space by assigning each voxel a number that identifies the distance to the nearest opaque voxel. Octree, a hierarchical data structure, is an efficient representation of homogeneous space by decomposing the 3D space recursively into uniform regions. Levoy [1990] proposed to accelerate the ray-casting algorithm through the traversal of a pre-built octree to avoid sampling in empty regions.

- Splatting algorithms

Westover first proposed the *Splatting* algorithm [1989; 1990], a typical object-space method. Its core idea draws from the phenomenon of a drop of water falling into a plate and splashing/splatting its energy around the center. In the splatting algorithm, voxels correspond to the drops of water and image plane to the plate, so voxel intensities are projected and spread cross multiple pixels, which are then composited into the image plane.

Since the volume data points themselves are the input samples of objects there is no need to generate interpolated values for volume re-sampling. The contribution of a volume sample at (i, j, k) to a point (x, y, z) can be evaluated as:

$$Contribution(x, y, z) = h(x-i, y-j, z-k)v(i, j, k) \qquad \textbf{(2.12)}$$

where v is the data value of the sample and h is the weighting function, or reconstruction kernel.

$$h(r, s, t) = \exp(-(r^2 + s^2 + t^2)/\sigma^2) \qquad \textbf{(2.13)}$$

Normally, a circular Gaussian kernel (Equation 2.13) is employed for isotropic influence (rotationally invariant weighting). Therefore, each volume sample can be treated individually and spread its contributions in space. The final image is produced by summing the contributions of all the samples at each pixel.

The splatting algorithm includes three steps. Firstly, the projection order of the voxels is determined according to the view-direction. Voxels that are closest to the image plane will be splatted first. Each voxel will be colored with user-predefined color and opacity transfer functions based on its intensity, and shaded according to its gradient. Next, the splatting algorithm projects the transferred voxels into image space. A blurring filter (reconstruction kernel) is used to compute the contribution of the voxel to an image buffer. For orthogonal viewing, the kernel can be calculated once and used for all the voxels with only an image plane offset. But for perspective viewing, a new oblique kernel has to be calculated for every voxel. This procedure can be sped up by maintaining a pre-computed footprint lookup table [Westover 1991]. The footprint is the projection of the kernel into the image buffer. At last, footprints are composited. The larger the footprint, the better is the suppression of the re-sampling artifacts. Larger footprints, however, are more costly to realize. After all the voxels have been processed, rendering is completed.

Early splatting elimination can be performed by dynamically maintaining an image occlusion map, which conservatively culls invisible splats early from the rendering pipeline [Mueller et al. 1999]. Although early splat elimination saves the cost of footprint rasterization for invisible voxels, their transformation still must be performed to determine their occlusion. Sobierajski et al. [1993] proposed a simplified approximation to the splatting method for interactive volume viewing in which only voxels comprising the object's surface are maintained. Laur and Hanrahan [1991] also proposed a method for efficient implementation of the splatting algorithm called hierarchical splatting, which uses a pyramid data structure to hold a multi-resolution representation of the volume.

The splatting algorithm is able to produce high-quality images similar to other algorithms. As the volume is processed slice-by-slice in the algorithm, it can provide users an incremental refined image. The major advantage of the splatting algorithm is that each voxel is considered only once and only the relevant voxels are considered every time, so this technique improved the speed of the ray-casting algorithm. On the other hand, because the projection of voxels is approximated by the kernel splat, rendering has a lower accuracy in comparison to ray-casting.

- Shear-warp algorithm

Lacroute and Levoy [1994] proposed the *shear-warp* algorithm, which is recognized as the fastest algorithm for software volume rendering to date [Meissner et al. 2000]. The algorithm is improved based on ray-casting scheme. It also takes advantage of object-space method, so shear-warp algorithm is often regarded as a hybrid method.

Two significant improvements are employed in shear-warp algorithm over ray-casting algorithm.

First, in shear-warp algorithm, the traditional model-view transformation in ray-casting is factorized to permutation-shear-warp transformation. It decomposes the 3D affine transformation into five 1D transformations, which significantly reduces the computational demand to only one floating point addition each [Hanrahan 1990]. An image plane called intermediate image or base plane is introduced. The base plane is chosen as overlapping with one of the six faces of the volume cube, and is the one onto which the image is projected to the largest area. Thus after transformed into shear space, the rays that cross the volume are perpendicular to the base plane. The step distance between two adjacent samples on a ray is defined as the slice distance so that samples fall exactly onto the slices and can be computed by bilinear interpolation instead of trilinear interpolation. After the intermediate image is produced, it is warped to the image plane, which is just a 2D transformation.

Second, run-length-encoding (RLE) is employed in shear-warp algorithm. During ray-casting in this algorithm, voxels are processed scanline by scanline, then slice by slice. Therefore, volume is encoded into RLE for skipping transparent voxels (space-leaping), and the image is also encoded into RLE for skipping opaque pixels (early ray-termination).

In shear-warp algorithm, the voxels can be accessed in object-order, or storage order. Consequently, the ray-casting process is accelerated remarkably by taking advantage of the continuous memory accessing. However, the system has to maintain three duplicated RLE-encoded volumes for each of the three principal viewing directions (x, y, and z). Two potential errors are introduced with the algorithm. First, it does not allow super-sampling along the ray, so Nyquist frequency is potentially violated for all but the axis-aligned views. Second, the 2D instead of 3D re-sampling may also result in artifacts.

2.3.2 Parallel Volume Rendering

Volume rendering acceleration can be done through algorithmic improvements. It can also be achieved by multiprocessor systems with parallel computing. The parallel approaches of volume rendering are mainly classified into image space parallelism and object space parallelism.

In image space parallel rendering methods, the screen is divided into several regions, and each processor is assigned a portion of the screen. If a processor finishes ray-casting and finds another region undone, it keeps on with that region. Nieh and Levoy proposed an efficient parallel ray-casting method that achieved good performance on a shared memory DASH machine in Stanford [Nieh and Levoy 1992]. Lacroute also parallelized the share-warp algorithm for shared memory architecture on a 16 processor Silicon Graphics Challenge [Lacroute 1995, 1996]. It gains a speed of 12 frames per second for a 256^3 dataset. Many other image space methods were reported over the years [Palmer et al. 1997]. They are carried out with a goal of partitioning image plane to achieve maximizing load balance and minimizing communication between multiple processors.

In object space parallel rendering methods, the volume data is divided into sub-cubes, and each processor renders a sub-cube of the volume data. The final image is obtained by compositing these sub-images of the sub-cubes in the right order. In this sort of methods, sub-image composition often consumes much time [Goldwasser et al. 1989]. Many approaches were proposed to solve such problem [Ma et al. 1994; Neumann 1994]. Splatting technique is also widely used in parallel volume rendering. Westover proposed and implemented the earliest parallel splatting renderer on a SUN network [1990]. Later, many other splatting-based parallel approaches were reported [Yagel and Machiraju 1995]. Beeson et al. [2003] also reported their parallel volume rendering solution using the perspective shear-warp volume rendering algorithm on a cluster network.

2.3.3 Hardware-Assisted Volume Rendering

To further enhance the performance of volume rendering, the use of specific hardware besides CPU has been investigated. In one approach, the traditional well developed volume rendering algorithms are specialized and implemented on the hardware adapter, called customized volume rendering hardware, which is able to render the volume on-the-fly. In the other approach, the capability of existing graphics card or graphics processing unit (GPU) is further exploited. The graphics card is able to compute volume rendering. Thereof, the texture-mapping is often used for volume rendering, where the graphics card implements the process similar to the ray-casting algorithm.

- Customized Volume Rendering Hardware

Special designed volume rendering customized hardware is distinguished in performance improvements. Notable examples include VolumePro series volume rendering acceleration board [Pfister et al. 1999] and VIZARD II, a reconfigurable interactive volume rendering system [Meissner et al. 2002]. In 1999, the publication of VolumePro graphics board by Mitsubishi has gained much attention in the volume rendering community. The VolumePro board implemented on the basis of Shear-warp algorithm provides high quality, real-time volume rendering on PCI-bus systems. Because of insufficient flexibility in rendering control, many customized applications such as integrated volume-polygon rendering cannot be easily implemented. As it supports only 3D volume rendering, higher dimensional and multimodality volume rendering cannot be accomplished either. Recently, the VolumePro graphics board has been upgraded by the employment of a shear-image order ray-casting algorithm [Wu et al. 2003], which achieves the image quality equivalent to the full image order volume rendering but with enhanced rendering performance. Graphics polygon embedded volume rendering is also supported.

- Texture-Based Volume Rendering

With the development of graphics hardware, texture-mapping technique comes mature and is widely used in hardware-assisted volume rendering (also referred to as GPU-based volume rendering). According to the types of the texture hardware, there are two texture-based volume rendering methods available, namely 2D and 3D texture-mapped volume rendering.

With the texture-mapping supported graphics hardware, volume rendering can achieve remarkable speedup. In the texture-based method, the 3D rasters (called texture maps) are mapped onto polygons in 3D with hardware interpolation. The series of polygons are rendered perpendicular to the viewing direction in front-to-back order, and blended into the frame buffer to produce the final image. The first non-shaded texture-based volume rendering method is proposed by Cabral et al. [1994], and later Gelder & Kim added in shading capabilities [1996]. Other researchers also proposed approaches to add more effects such as shadows [Behrens and Ratering 1998] or to improve the classification and shading methods [Meissner et al. 1999]. In [Boada et al. 2003], 3D texture-mapping is used for the visualization of the volume data integrated with the surface polygons based on a hybrid octree. Wilson et al. also reported a hybrid point-based volume rendering technique by using texture-mapping [Wilson et al. 2002]. Kruger and Westermann [2003] realized the early-ray termination and empty-space skipping techniques in 3D texture-based volume rendering so that the performance is further improved.

The texture-based volume rendering method theoretically produces images with qualities matching that produced by ray-casting algorithm. In comparison to ray-casting, texture-mapping method is much faster due to hardware acceleration. However, this method is normally OpenGL[2] dependent and its capabilities are limited by graphics hardware, for example, the supported volume data is often restricted by the texture memory on-board. Although there are many proposals, the extensive and efficient shading is still a sophisticated task in texture-mapped volume rendering.

Interested reader can refer to [Ma et al. 2003], in which authors reviewed more hardware-accelerated algorithms that have been introduced recently. There are several implementations of conventional ray-casting methods in the community. VTK is a popular software library for visualization in general [Schroeder et al. 1998].

2.4 Four-Dimensional Volume Rendering

Most research on visualization of 4D/time-varying volume data focuses on rendering acceleration and storage space reduction. Spatial coherence and temporal coherence are the

[2] OpenGL is the industry's foundation for high performance graphics and widely adopted graphics standard, *http://www.opengl.org*, SGI.

most important two characteristics of the time-varying volume data usually exploited for this purpose.

Shen et al. [1999] proposed the time-space partitioning (TSP) tree to capture both spatial and temporal coherence. The skeleton of a TSP tree is a complete octree that recursively subdivides the 3D space, while each octant contains a binary tree that bisects the time span. The coefficients of variation in space and time domains are used as the error metrics to evaluate the spatial and temporal coherence of a 4D volume dataset. Regions identified with little temporal variance are skipped from repeatedly rendering and their partial rendering results are reused to speedup overall rendering. The algorithm provides error control for users so that the image quality is possible to be traded off for rendering speed. The TSP tree is built as a supplementary data structure to the 4D volume. Therefore, it results in extra memory overhead. This method also cannot effectively reduce the space or I/O requirement as the original 4D volume data are kept and used simultaneously during rendering. Ellsworth et al. [2000] improved the TSP tree by using new color-based error metrics to enhance the capability of identifying coherent regions. The TSP tree can capture both the spatial and temporal coherence from a time-varying field of the whole sequence of volumes more effectively and the rendering performance is thus improved. However, the error metrics have to be recomputed once the transfer function is changed.

Since there is coherence between the time-varying volume data, the volume rendered images may not change significantly from one time step to another. Shen and Johnson [1994] proposed the differential volume rendering method, in which only the changed pixels are re-rendered in each time step. However, the process of determining the changed pixels may take long time especially when the amount of changed pixels is large. Liao et al. [2003] improved this process by using a two-level differential volume rendering method. The authors noted that some of the changed voxel positions could appear repeatedly between time steps. Therefore, the determination of the projected positions of these voxels can be omitted in rendering of successive time steps. They filter out the overlapped changes voxels and extract the difference of the changed voxels referred to as the second-order difference (SOD). Based on the SOD, their method saves the time to determine the positions of changed pixels and improves the rendering performance. However, this method cannot completely take advantage of the data coherence to further accelerate rendering. All the changed pixels have to be re-rendered from scratch in every time step no matter in the original or the improved algorithm. The information of differential voxels and the second-order differential voxels demand redundant memory and storage space, and when the amount of changed voxels is large, this redundancy can be significant. This method is designed for ray-casting renderer only and is hard to be extended to support other rendering algorithms such as texture-based volume rendering. Liao et al. [2004] further extended this method to support time-critical time-varying volume rendering. The volume rendered images can be produced within a time constraint at the reasonable price of image quality. It will be useful for some applications.

Some popular 3D volume rendering algorithms are further investigated to be suitable for 4D rendering. Anagnostou et al. [2000] proposed a time-varying volume rendering method based on shear-warp factorization algorithm. In this method, the RLE-encoded volumes are divided into slabs, and slabs are further subdivided into run blocks in the sheared object space. Temporal coherence is exploited and only run blocks changed over time are rendered.

Blending of the partial images of all the slabs produces the final image of each time step. In [Anagnostou et al. 2001], the authors reported their extended researches on shear-warp-based time-varying volume rendering algorithm. A change detection technique is adapted to process Poisson noise distributed datasets besides Gaussian noise. Neophytou and Mueller [2002] proposed a 4D splatting method based on 4D body-centered cubic (BCC) grids, which is able to provide more efficient sampling lattice than the usual Cartesian cubic (CC) grids. A hyper-slice approach is used to extract 3D volumes from the 4D lattice, and a splatting renderer is used for visualization.

Texture-based hardware-assisted volume rendering is an active research area recently in both 3D and 4D [Boada and Navazo 2003; Wilson et al. 2002]. Lum et al. [2001] reported a 2D texture-based solution for interactive time-varying volume data rendering. They use discrete cosine transform (DCT) and vector quantization techniques to compress time-runs of voxels into single byte indices that are then loaded into the texture memory. The compressed data result in reduced I/O cost between hard disk, system memory and texture memory. During rendering, a dynamic time-varying color palette is employed so that the indexed volumetric data are quickly decoded in hardware. Although artifacts may be introduced, this method provides users interactive 4D data exploration by fully utilizing the texture capability of a commodity graphics card. However, the spatial coherence of data is not exploited in this method for further performance improvement. The customized hardware-accelerated time-varying volume rendering methods are also investigated. For example, the VolumePro [Wu et al. 2003] graphics card is being extended to support 4D volume animation.

While some methods concentrate on rendering acceleration, other efforts emphasizing more on data compression are also reported. Ma et al. [1998] proposed an algorithm for encoding and rendering of time-varying volume data. They use quantization for voxel-level compression and octree encoding for spatial domain compression. The temporal domain compression is achieved by difference encoding. Wavelets transform [Dobashi et al. 1998; Guthe and Straßer 2001] is one of the most popular compression schemes used in time-varying or large-scale volume visualization. Sohn et al. [2002; 2004] proposed a combined compression scheme from wavelet transform and MPEG architecture. The spatial and temporal coherence of time-varying data are exploited by encoding only blocks containing significant features. The seed cells are inserted into the encoded data so that an online iso-surface extraction can be performed quickly. Finally, a combined rendering of iso-surface and volume is presented with geometric primitives and textures, where the iso-surface is used to represent the principal objects in details and volume rendering builds the surrounding effects. In all these algorithms, a lossy compression scheme is employed.

Besides trying to comprehend time-varying volume data through playback or animation, researchers also attempt to reveal the inherent information of time-varying data by other means. Tory et al. [2001] studied this problem in medical field. To discover abnormalities in a time-varying medical dataset with a more informative representation, they studied to visualize four different quantities of the dataset in 3D, i.e., intensity, temporal gradient, spatial gradient and changes of spatial gradient over time. This research provides a new insight to the time-varying datasets. Woodring and Shen [2003] proposed an alternative method for viewing time-varying volume data. A sequence of time-varying volumes is first integrated into a single volume, named chronovolume, which captures the essence of multiple time steps and is then visualized by a regular 3D renderer. To reveal different interests of time-varying data, several time integration functions are proposed. As noted by the authors, since many time steps contribute to one volume, the final image tends to clutter if users are not careful in manipulating the transfer functions.

Chapter 3

Dynamic Linear Level Octree for Time-Varying Volume Rendering

3.1 Introduction

The spatial data structures are used traditionally in the acceleration of 3D volume rendering to exploit the coherent property of volume images. When information of the volume is well organized in a spatial data structure, it is possible to perform volume rendering faster. For example, when a sub-volume is labeled as empty in a spatial data structure such as octree, the renderer can safely skip that region (space leaping) in the rendering process.

To accelerate 4D or time-varying volume rendering, many research attempts are made [Shen et al. 1999; Ellsworth et al. 2000; Anagnostou et al. 2001; Liao et al. 2003]. However, researchers found that it might not be a good idea to simply extend an octree to a hex-tree with the introduction of an additional temporal dimension. The granularity in the temporal dimension is totally different from that in spatial dimensions. The hex-tree, if it is used, will be very biased in certain dimension and it is hard to benefit rendering. Therefore, special attention is given to the temporal dimension. The improved data structure I proposed is able to represent the 4D volume data more naturally and effectively. It can be used efficiently to accelerate rendering.

As an introduction of the proposed time-varying volume rendering algorithm, we begin with the 3D algorithms and data structures as follows.

3.2 Linear Level Octree

3.2.1 Review of Octree in Volume Rendering

Octree is a hierarchical data structure. We start with a volume and determine if its description is sufficiently complex, in which case the volume is subdivided uniformly into eight congruent disjoint cubical regions (called *octants*), and then each octant is recursively checked and subdivided, until the complexity is sufficiently reduced and meets our predefined leaf criteria [Samet and Webber 1988]. Each leaf octant represents a homogeneous region. The homogeousness of regions suggests the spatial coherence, which is the characteristic frequently used for rendering acceleration.

The octree data structure is traditionally used to assist the intersection tests in the ray tracing algorithm [Kaplan 1985; Goldsmith and Salmon 1987; Whang et al. 1995] (Note: not ray-casting). A naïve ray-tracing algorithm would have to recursively test the rays of light emitted from the viewpoint against each of the surfaces of the objects, sort the resulting intersections, calculate and test the reflected rays to check if they intersect any other portions of the objects. The octree partitions the space recursively and forms non-overlapped minimum bounding boxes for every object. As a result, the ray-object intersection test can be performed more efficiently because the object is examined against the ray only when the bounding octant is found to intersect the ray.

Meagher [1982] is the first to propose a method that exploits the coherency in volume data by representing the 3D volume with an octree for geometric modeling. Years later, Levoy [1990b] extended the idea and employed the octree in ray-casting algorithm for volume rendering acceleration. In Levoy's method, each octant contains a flag with value zero or one to indicate whether the region occupied by the octant is empty or not respectively. During the ray casting, the flags of the octree are concurrently checked, and if a region is inside an octant with flag zero, the ray can advance quickly across the empty space; otherwise, samples will be drawn along the ray. In this method, the space leaping of ray-casting is well realized. However, the ability of octree to represent nonempty homogeneous regions is not yet explored to further speed up rendering.

In the shear-warp rendering algorithm, a pre-computed min-max octree, is employed for the interactive classification [Lacroute and Levoy 1994]. Each octant contains the extrema of the parameter values of the opacity transfer function. The opacity is then evaluated for all parameter points in the octant region and integrated with a supplementary multi-dimensional summed-area lookup table. The scan-line is thus recursively checked and transparent portions are determined. By this means, authors resolve the low-speed problem in unclassified-volume rendering and provide rendering with an interactive performance.

Researchers [Whang et al. 1995; Lee and Park 1997] also proposed the methods of building the octree by adaptively subdividing the volume space into non-uniform regions depending on the degree of spatial coherence of the volume data. In this way, a greater number of empty octants are generated and more time is saved while rays pass through these empty octants. However, the calculation complexity to identify the volume-ray intersection is inevitably increased, because such process cannot be done efficiently as before by using the regularity of the uniform structure.

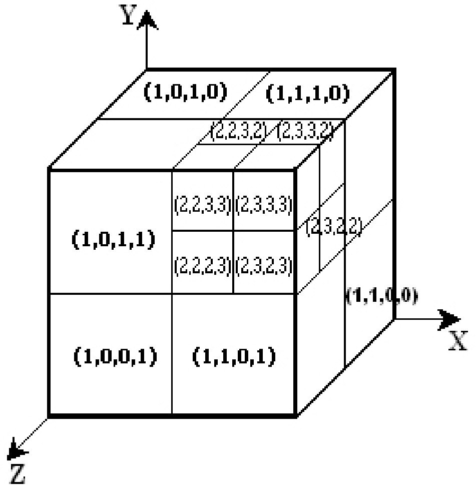
As mentioned in Chapter 2, time-space partitioning tree, an octree-variant, is proposed for the 4D volume rendering [Shen et al. 1999]. A comparison between this method and my method will be given Chapter 7.

So far octree is mostly used for surface rendering (as in ray-tracing algorithm) or as an auxiliary data structure used to describe the properties of regions in the volume, but not to rebuild the volume and make rendering directly based on it. When we represent volume data with an octree, the octree-based rendering is benefited by saving the cost of operations involving the octree and the volume.

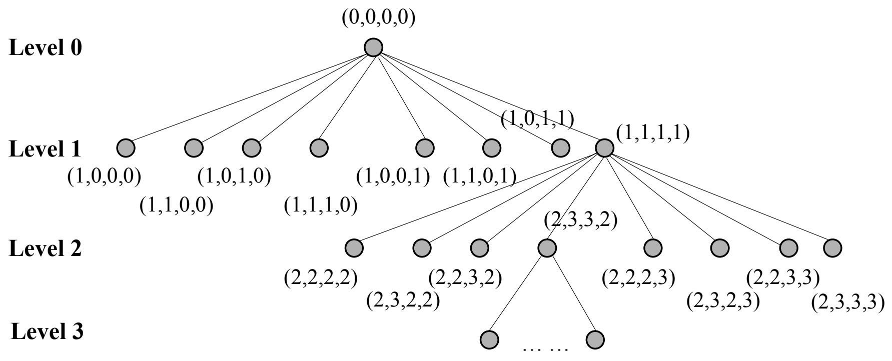
3.2.2 LLO Labeling Scheme

Octree can be labeled with different schemes. Linear Level Octree (LLO) [Chui et al. 1991], extended from the linear-octree scheme, is one of the labeling schemes. Chui et al. did a good early work to visualize the volume data in geometry by using the LLO. However, they did not propose a solution for LLO-based volume rendering.

A volume with size $2^n \times 2^n \times 2^n$ in a coordinate system is shown in Figure 3.1(a), where n is the resolution of the raster. The left-bottom-back corner/voxel of the volume is located at the origin, and its three edges are aligned with the X, Y and Z coordinate axes respectively. In the following description, we do not differentiate the octants and nodes of LLO.



(a) Decomposition of a volume



(b) Structure of the LLO corresponding to the volume

Figure 3.1 Linear level octree labeling scheme

In LLO, each octant/node is labeled with a unique *code key*: (L_i, x_i, y_i, z_i), where L_i is the level of the node, and x_i, y_i and z_i are the X, Y and Z level-coordinates of the node respectively.

Let the code key of the root node be (0, 0, 0, 0) and assume that an arbitrary node A at level L_i has a code key (L_i, x_i, y_i, z_i). Then the left-bottom-back sub-node of A at level $L_i + \Delta L$ (ΔL = 1, 2, …) will have a code key (L_{i0}, x_{i0}, y_{i0}, z_{i0}), where

$$\begin{aligned} L_{i0} &= L_i + \Delta L \\ x_{i0} &= x_i \bullet 2^{\Delta L} \\ y_{i0} &= y_i \bullet 2^{\Delta L} \\ z_{i0} &= z_i \bullet 2^{\Delta L} \end{aligned} \qquad \textbf{(3.1)}$$

The code keys of other sub-nodes of A at level $L_i + \Delta L$ are given as follows:

$$\begin{array}{l} L_{i0}, x_{i0}, y_{i0}, z_{i0} \\ L_{i1}, x_{i1}, y_{i1}, z_{i1} \\ L_{i2}, x_{i2}, y_{i2}, z_{i2} \\ L_{i3}, x_{i3}, y_{i3}, z_{i3} \\ L_{i4}, x_{i4}, y_{i4}, z_{i4} \\ L_{i5}, x_{i5}, y_{i5}, z_{i5} \\ L_{i6}, x_{i6}, y_{i6}, z_{i6} \\ L_{i7}, x_{i7}, y_{i7}, z_{i7} \end{array} = (L_{i0}, x_{i0}, y_{i0}, z_{i0}) + \begin{array}{l} 0, 0, 0, 0 \\ 0, 1, 0, 0 \\ 0, 0, 1, 0 \\ 0, 1, 1, 0 \\ 0, 0, 0, 1 \\ 0, 1, 0, 1 \\ 0, 0, 1, 1 \\ 0, 1, 1, 1 \end{array} \qquad \textbf{(3.2)}$$

Figure 3.1 demonstrates this LLO labeling scheme. The relationship between LLO nodes (Figure 3.1a) and their corresponding sub-volumes (Figure 3.1b) can be identified through their code keys.

We define the distance between two adjacent voxels in three axis directions as one unit. Based on the basic labeling scheme of LLO, we derive the following properties of an arbitrary node A with code key (L, x, y, z):

- $Size = 2^{\Delta L}$, where $\Delta L = n - L$.

This is the side length of node A, or in other words, it is the number of voxels contained in A along any one of the axis directions.

- $Location = (x \bullet 2^{\Delta L}, y \bullet 2^{\Delta L}, z \bullet 2^{\Delta L})$, where $\Delta L = n - L$.

This is the absolute location coordinate (not the level coordinate) of the left-bottom-back corner voxel of A in the object's (volume) coordinate system. It is also regarded as the location of node A.

The code key of upper level (ancestor level) node of A, say level L_w ($L_w < L$), is:

- $(L_w, \lfloor x/2^{\Delta L} \rfloor, \lfloor y/2^{\Delta L} \rfloor, \lfloor z/2^{\Delta L} \rfloor)$, where $\Delta L = L - L_w$ and $\lfloor\,\rfloor$ is an integer division

With these properties, we can identify the location, the size and the code key of either a parent node or a child node of an arbitrary octant easily with only a few simple arithmetic operations. Advantages of LLO include reduction of memory consumption, simple representation of octants, fast tree traversal and so on. LLO has all the advantages of linear-octree. For instance, only leaf nodes are stored, which greatly reduces both memory consumption and the number of nodes processed. The memory space required using LLO is less than that of linear-octree for the same object. It implicitly encodes the locations, the sizes of the nodes and the path from the root to the nodes. Therefore, it is more computation efficient to evaluate the relations between arbitrary spatial points and LLO nodes.

3.2.3 LLO Generation

To take advantage of the spatial coherence, volume datasets are converted into LLO representation. Without loss of generality, a volume dataset is assumed to have 2^n voxels in each dimension. Two leaf node criteria are adopted for the LLO generation:

- The minimum size (edge length) of an octant is 2^l, i.e., each dimension of the octant contains 2^l voxels at least, where l is an integer between 1 and n.
- The variance of voxel intensities contained in an octant is no more than v, a user predefined threshold.

The variance of an octant A at level L can be computed based on Equation 3.3 below:

$$\begin{aligned} m_A &= \frac{\sum_i v_i}{2^{3(n-L)}} \\ v_A &= \frac{\sum_i (v_i - m_A)^2}{2^{3(n-L)}} \end{aligned} \tag{3.3}$$

where v_i is the intensity value of the ith voxel of octant A, m_A is the mean and v_A is the variance.

However, Equation 3.3 could be computationally expensive when we compute the mean and variance of an octant at a high level (small *L*) of an octree. Voxels have to be accessed repeatedly for octants at each level. In fact, the mean and variance of a parent octant can be computed based on those of its child octants. Equation 3.4 gives the formulas without proof[3].

$$\begin{aligned} m_A &= \frac{1}{8}\sum_{i=1}^{8} m_i \\ v_A &= \frac{1}{8}\sum_{i=1}^{8}\left[v_i + (m_i - m_A)^2\right] \end{aligned} \qquad \mathbf{(3.4)}$$

where m_i and v_i are the mean and variance of the *i*th child of their parent octant *A* respectively. The mean and variance of a parent octant thus can be computed efficiently without accessing each voxel. In practice, we compute the mean and variance of the octants at level $(n - l)$ based on Equation 3.3 and those of their parent octants based on Equation 3.4. Each voxel is only accessed once in the whole process.

The first of the leaf node criteria is a necessary condition. Octants of size larger than the minimum size can be a leaf node only if it satisfies the criterion two. If all the voxels contained in a leaf octant are transparent, the octant is a *white* node and it is discarded. Otherwise, it is a *black* node, and saved. When the inequality $v_A \leq v$ is satisfied, the sub-volume data in octant *A* are regarded as homogeneous and is possible to be used for rendering acceleration.

[3] Interested readers can refer to Appendix C for more details.

To produce an LLO, all leaf octants are output together with their code keys, mean and variance. If an octant has variance no more than v, the sub-volume data contained in the octant will not be saved, otherwise, all the voxels are saved. After being converted into an LLO, the original volume dataset is not needed for rendering any longer.

Figure 3.2 gives the pseudo-code of the algorithm converting a volume to an LLO. The algorithm merges every eight adjacent octants into a bigger octant if they are all leaf nodes and the combined octant still satisfies the leaf node criteria. A supplementary data structure (`OctantArray`) is used to track the maximum level that the octants can be merged. According to the leaf node criteria, the smallest octant is of size 2^l. Therefore, the levels in `OctantArray` are initialized as $(n - l)$ that every octant in this level is potentially a leaf node.

```
Void CreateLLO()
{
  // The 3D array used to represent the properties of octants
  struct {
    int level;
    float variance, mean;
  } OctantArray[2^(n-l)][2^(n-l)][2^(n-l)];
  Initialize(OctantArray);

  // The distance of adjacent octants
  int D[8][3]={{0,0,0}, {1,0,0}, {0,1,0}, {1,1,0},
               {0,0,1}, {1,0,1}, {0,1,1}, {1,1,1}};
  int Step = 2;

  // Start the conversion from the lowest level
  L = n - l;
  while (L > 0) // Until root level
  {
    // Check all the octants in current level
    for (int k=0; k<2^(n-l); k+=Step) {
      for (int j=0; j<2^(n-l); j+=Step) {
        for (int i=0; i<2^(n-l); i+=Step)
        {
          // Check if all the eight brother octants are leaf nodes
          // with low variance
          for (int r=0; r<8; r++)
          {
            Brother =
            OctantArray[k+D[r][2]][j+D[r][1]][i+D[r][0]];
            if ((Brother.level > L) || (Brother.variance > ν))
              break;
          }

          V = variance of the eight octants;
          M = mean of the eight octants;

          // Check if the eight octants can be merged into a bigger
          // leaf octant at a upper level
          if (r == 8 && V <= ν)
          { // Yes
            OctantArray[k][j][i].level--;   // New octant level
            OctantArray[k][j][i].variance=V;// Update the variance
            OctantArray[k][j][i].mean = M;  // Update the mean
```

```
              }
              else
              { // No, the eight octants cannot be merged.
                Output the leaf octants whose level equals to L.
              }
            }
          }
        }
        // Update the distance between adjacent octants
        Update the elements in array D by multiplying 2;
        Step *= 2;

        // Repeat the process in an upper level
        L--;
      }
    }
```

Figure 3.2 Algorithm of LLO generation

Smaller size of leaf node allows smaller regions of data coherence to be exploited without sacrificing image quality. However, along with the benefit, the number of nodes increases significantly, which tends to incur substantial memory overhead. Furthermore, the increased number of rendering elements will also lead to extra graphics overhead during rendering. Therefore, l is selected to ensure that there will not be too many black leaf nodes generated in an LLO. The number of levels in an LLO will not exceed $(n - l + 1)$.

In the above description, we assume a volume dataset with 2^n voxels in each dimension. In practice, we will force the size of the volume to be the smallest power of 2 larger than the actual size and fill the extended portion with the background color. Because the extended portion of the volume is regarded as empty/transparent space, it will be processed as a few big white nodes that will not be stored or affect the algorithm performance at all.

3.3 LLO-Based 3D Volume Rendering

3.3.1 Overview

The flowchart of the LLO-based 3D rendering algorithm is given in Figure 3.3. An LLO is either encoded from a volume dataset for the first time or reloaded from storage media. During rendering, the LLO is traversed and octants are output one by one according to the current viewing direction. Since the LLO encodes only the non-transparent sub-volumes, every octant potentially contributes to the final image. A conventional 3D volume renderer, either a software-based such as a ray-caster or a hardware-accelerated such as a texture-mapped renderer, is employed for octant rendering. Finally, the partial images of octants are composited to produce the final image.

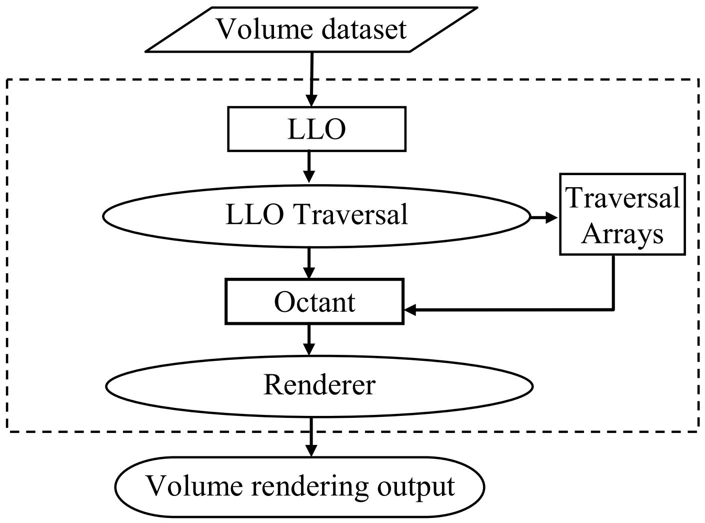


Figure 3.3 Flowchart of the LLO-based 3D volume rendering

Depending on the renderer selected, the LLO traversal order varies. For example, to utilize the texture-mapping technique for hardware-accelerated volume rendering, octants must be traversed in back to front order for proper texture blending, but if a ray-casting renderer is used, it is more efficient to traverse the LLO and composite partial images in front to back order.

In the following sub-sections, I will discuss this process in more detail.

3.3.2 LLO Traversal

An efficient traversal algorithm is critical for the LLO-based rendering algorithm. The LLO is traversed during rendering. Different tree traversal algorithms are used according to different projection methods, parallel or perspective projection. In parallel projection, the proposed algorithm can even avoid the cost of tree-traversal using a supplementary data structure. This algorithm is modified for the perspective projection with small traversal cost.

In parallel projection[4], LLO can be processed in a specific order without affecting the result image based on the viewing directions. There are eight distinct sets of such viewing directions. The child nodes of an octant are numbered as shown in Figure 3.4.

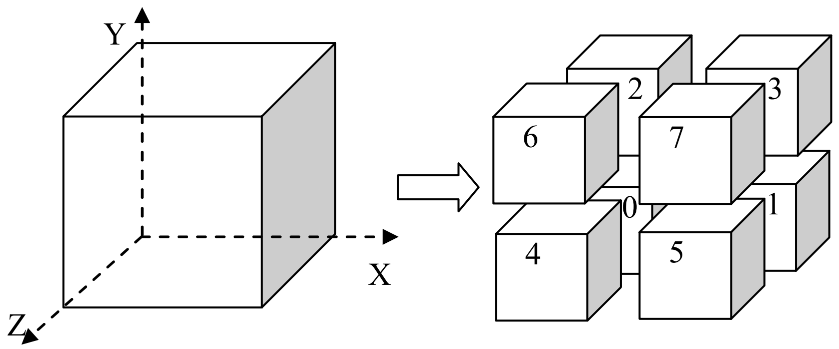


Figure 3.4 Numbering of child nodes

[4] The parallel projection is also called orthographical projection in some literature.

A vector (V_x, V_y, V_z) in object/model coordinate system is used to represent different viewing directions. The traversal sequence of nodes from different viewing directions is given in Table 3.1. The underlined nodes have the same traversal priority so that they can be processed in different order without affecting the result.

Table 3.1 Linear level octree traversal sequence

(V_x, V_y, V_z,)	**Traversal Sequence**
(<0, <0, <0)	$\underline{7}$, $\underline{3, 5, 6}$, $\underline{1, 2, 4}$, $\underline{0}$
(>0, <0, <0)	$\underline{6,}$ $\underline{2, 4, 7}$, $\underline{0, 3, 5}$, $\underline{1}$
(<0, >0, <0)	$\underline{5}$, $\underline{1, 4, 7}$, $\underline{0, 3, 6}$, $\underline{2}$
(>0, >0, <0)	$\underline{4}$, $\underline{0, 5, 6}$, $\underline{1, 2, 7}$, $\underline{3}$
(<0, <0, >0)	$\underline{3}$, $\underline{1, 2, 7}$, $\underline{0, 5, 6}$, $\underline{4}$
(>0, <0, >0)	$\underline{2}$, $\underline{0, 3, 6}$, $\underline{1, 4, 7}$, $\underline{5}$
(<0, >0, >0)	$\underline{1}$, $\underline{0, 3, 5}$, $\underline{2, 4, 7}$, $\underline{6}$
(>0, >0, >0)	$\underline{0}$, $\underline{1, 2, 4}$, $\underline{3, 5, 6}$, $\underline{7}$

Further examination of Table 3.1 shows that there are only four distinct traversal sequences, while others are just in inverse order of one another. Therefore, before rendering, the LLO can be traversed once for each of the four viewing directions, and the pointers of the leaf nodes are stored in four different traversal arrays (Figure 3.3). Traversal of the LLO during rendering is thus avoided.

During rendering, one of the four traversal arrays is selected according to the current viewing direction, and octants are processed from the beginning to the end or in inverse order of the arrays. Since the traversal of the hierarchical data structure is one of the most time consuming operations and it is also regarded as the major disadvantage of the hierarchical data structure [Yagel 1999], a substantial performance gain is achieved by this method.

In perspective projection, the LLO traversal order is not only dependent on the viewing direction but also the viewing point and the field of view (FOV). Therefore, the traversal sequence varies for different portions of a volume and need to be carefully considered.

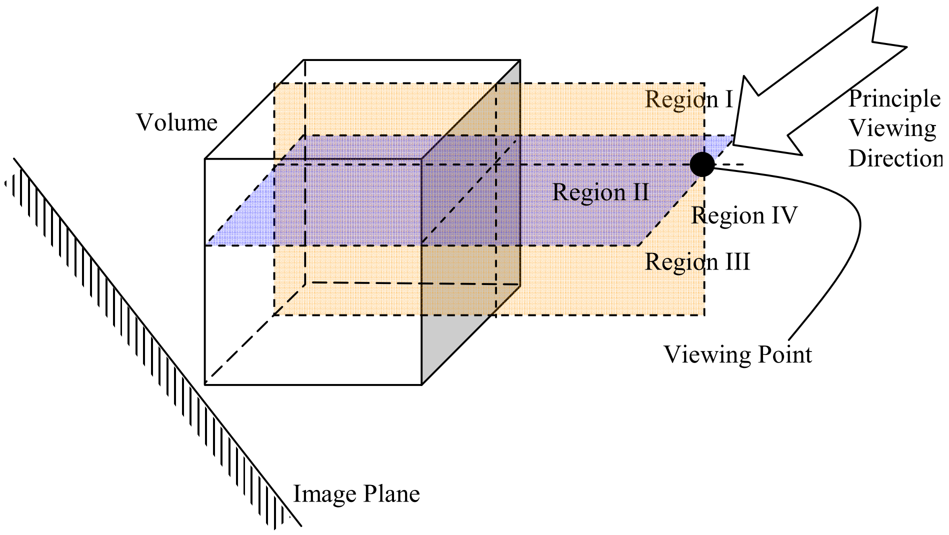


Figure 3.5 FOV Regions for Perspective Projection

To identify the traversal order in different sub-volumes, the volume space is divided into regions in the object coordinate system. Three planes parallel to the three sides of the volume and passing through the viewing point are generated. We consider only the case that the viewing point is outside of the volume. There are at most two planes that can intersect with the volume. No matter what is the viewing direction, the field of view is divided into four regions as shown in Figure 3.5.

We noticed that, in the same FOV region, all the viewing directions are conformable and belong to the same input entry of the Table 3.1 as suggested in the parallel projection. Therefore, the traversal order in the same FOV region will be consistent as well. The following rules are derived to identify the LLO traversal order for perspective projection.

- If an octant fully resides in a FOV region, the traversal sequence is looked up based on the viewing direction of the region and Table 3.1.
- If an octant straddles multiple FOV regions, the traversal sequence is determined by $\vec{V} = \vec{P}_c - \vec{P}_{vp}$ and looked up in Table 3.1, where $\vec{P}_c$ is the center-point of the octant and $\vec{P}_{vp}$ is the viewing point.

Figure 3.6 gives a 2D example of the LLO traversal in the perspective projection. The field of view is divided into two regions, where all the viewing direction vectors (v_x, v_y) in Region I conform to $v_x < 0$ and $v_y > 0$ and viewing vectors in the region II conform to $v_x > 0$ and $v_y > 0$. Therefore, octants in the different regions are traversed in different orders, and the octants in the same region are traversed in the same order. The numbers on sub-volumes indicate the sequence that octants are accessed and processed in front-to-back order.

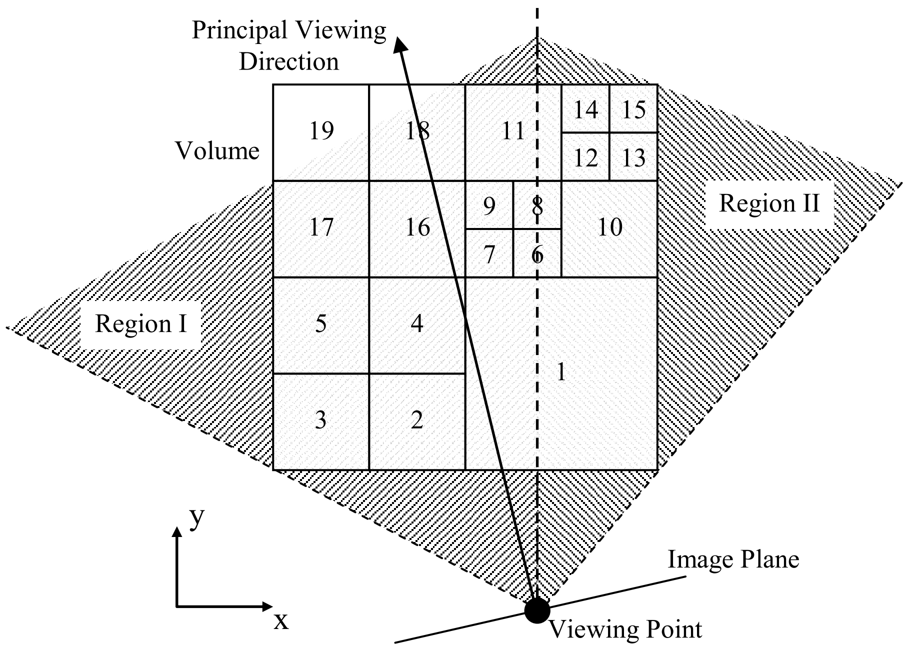


Figure 3.6 Octant traversal order in perspective projection

LLO-based perspective rendering is thus achieved within the same framework as parallel rendering. However, it is no longer possible to avoid the traversal process.

3.3.3 Adaptive Rendering

Since the mean and variance are encoded together with each octant in generation of an LLO, adaptive volume rendering can be realized through user-specified error tolerances τ during rendering. While rendering each of the octants, variance is compared with the error tolerance and if it is smaller, the mean value is used instead of the octant voxels for either octant reconstruction or interpolation. A low error tolerance is chosen when image quality is required to be high (e.g., medical examination), and a high error tolerance is chosen for fast rendering for the price of image quality. Users thus are able to control the rendering quality for different applications.

This LLO-based volume rendering algorithm is independent of any specific volume renderer. Volume rendering algorithms such as ray-casting, splatting, shear-warp and texture-mapping can be employed. Here, I briefly introduce how the two typical renders, ray-caster and texture-mapped renderer, are used in LLO-based volume rendering.

- Ray-casting Renderer

When we use the ray-casting algorithm, LLO is traversed and octants are accessed in front to back order according to the viewing direction. Given the transformation of the parent octant, the transformation of its sub-octants can be efficiently computed and the corresponding projection area in the image plane can be identified efficiently. Rays are cast through non-opaque pixels in the image plane. Samples are taken and shaded in the octant and accumulated to the pixel until it is opaque enough. Normally, it is necessary to have one layer of voxel overlap at the boundary of octants for proper tri-linear interpolation and two layers of voxel overlap for gradients computation. When a leaf node with low variance is encountered, sampling in the octant can be avoided and the mean is used instead. Since all sample points in this octant have identical intensity (I_0) and opacity (α_0) values, the integration of sample points can be optimized and the volume rendering equation is simplified to:

$$\begin{aligned} I &= \sum_{i=0}^{n-1} I_i \prod_{j=0}^{i-1} (1-\alpha_j) \\ &= \sum_{i=0}^{n-1} I_0 (1-\alpha_0)^i \end{aligned} \qquad \textbf{(3.5)}$$

and the integrated opacity is computed as:

$$\alpha = \sum_{i=0}^{n-1} \alpha_0 (1 - \alpha_0)^i \tag{3.6}$$

where *n* is the number of samples along a ray within this octant.

In the LLO-based ray-casting algorithm, it is no longer needed to consider the space-leaping either, as only non-transparent octants are passed to the renderer. Rendering is completed when all the octants are processed.

- Texture-mapped Renderer

When texture-mapping is employed for octant rendering, the algorithm is able to benefit from the hardware acceleration. Depending on the type of graphics hardware available, leaf octants are rendered through 2D or 3D texture-mapping techniques. In the case of 2D texture-mapped octant rendering, software sampling is required to create the texture images for the three major orientations. To minimize the cost, textures can be pre-computed and saved at the expense of system memory. 3D texture-mapping is superior to the 2D technique in its capability to perform fast 3D interpolation. Therefore, software sampling is avoided. For leaf octants with low variance, textures are reconstructed from their mean values in 2D or 3D accordingly. The low variance octants can also be rendered using flat-shaded polygons instead, as most graphics systems render flat-shaded polygons faster than texture polygons [Ellsworth et al. 2000]. This approach also saves the texture memory and texture downloading time. After octant images are downloaded to the 2D/3D hardware texture memory, texture polygons are transferred to the correct locations based on their code keys. No matter which texture hardware is used, textures must be mapped in back to front order. To ensure that boundaries between octants do not result in image artifacts, it is necessary to have one layer of voxels overlap between neighboring octants.

3.4 Dynamic Linear Level Octree

3.4.1 Overview

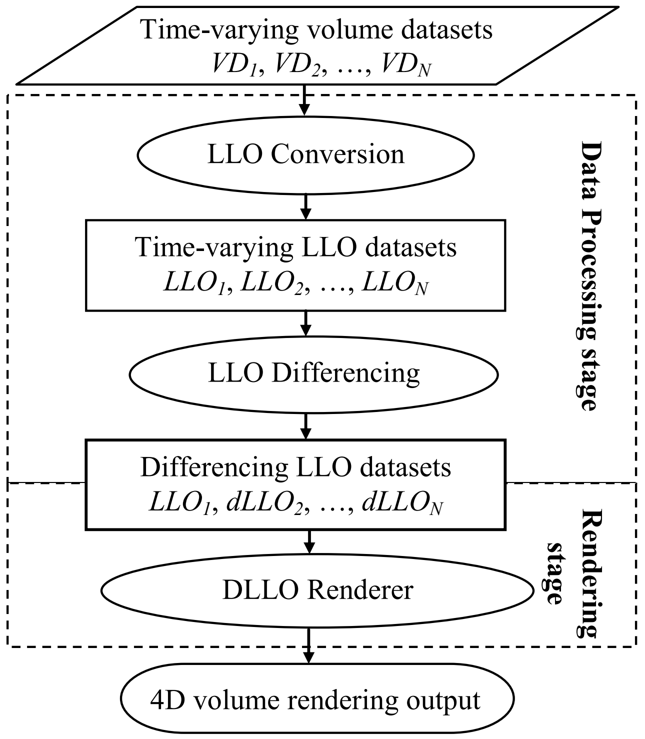

Time-varying volume datasets
$VD_1, VD_2, \ldots, VD_N$
LLO Conversion
Time-varying LLO datasets
$LLO_1, LLO_2, \ldots, LLO_N$
LLO Differencing
Differencing LLO datasets
$LLO_1, dLLO_2, \ldots, dLLO_N$
DLLO Renderer
4D volume rendering output
Data Processing stage
Rendering stage

Figure 3.7 Flowchart of DLLO-based 4D volume rendering

Extended from the 3D LLO-based volume rendering algorithm, 4D volume rendering is performed with a dynamic LLO. During rendering, a working LLO is updated by new datasets. A change detection algorithm is employed to exploit the temporal coherence between datasets, and 3D volume rendering is applied only to the octants changing over time. This is the essential idea on the dynamic linear level octree (DLLO).

4D volume rendering is divided into two steps, data-processing stage and rendering stage as shown in Figure 3.7. In the first stage, a time sequence of volume datasets is converted into a dynamic LLO. In the next stage, 4D volume rendering is performed. These two stages are introduced in the following sub-sections.

3.4.2 DLLO Generation

A 4D medical volume dataset is normally composed of sets of 3D volume datasets, which are acquired at a sequence of time steps, so the 4D volume is often referred to as time-varying volume data as well. We denote the N sets of time-varying volume data as (VD_1, VD_2, …, VD_N), which are inputs to our 4D volume rendering system. In the data processing stage, the input is converted into LLO representations so that the spatial coherence is exploited inside each of the volumes. We denote them as (LLO_1, LLO_2, …, LLO_N). Second, an LLO differencing algorithm is employed to exploit the temporal coherence between datasets. After this process, these time-varying LLO datasets are converted into the representations, named differencing LLO datasets (LLO_1, $dLLO_2$, …, $dLLO_N$). Finally, they are passed to the rendering stage.

The LLO differencing algorithm is designed to detect the difference between two consecutive datasets. In the algorithm, two LLOs are traversed and compared simultaneously. The second LLO is clipped into a differencing LLO (*d*LLO), while the first one remains intact.

Only changed nodes are retained in the *d*LLO while the homogeneous leaf nodes between two datasets are cropped. The details of LLO differencing algorithm is given in Figure 3.8.

```
int Differencing(Octant_A, Octant_B)
{
      Check the termination conditions in Table 3.2. If octants A
      and B satisfy an entry, the value is returned and additional actions
      are applied as suggested by the table. Then this function is
      terminated; otherwise, go ahead.

      // Both Octant A and B are intermediate nodes.
      int EmptyChildNodeCount = 0;
      for (int i=0; i<8; i++)
      {
            // Traverse each child of octant A and B simultaneously.
            EmptyChildNodeCount +=
            Differencing(Octant_A.Child[i], Octant_B.Child[i]);
      }

      // Check whether all eight child nodes of octant B are empty.
      if (EmptyChildNodeCount == 8)
      {     // Yes
            remove Octant_B;
            return 1;
      }
      else
      {     // No
            return 0;
      }
}
```

Figure 3.8 Differencing algorithm

The two parameters of the differencing algorithm are octants from the same location of two LLOs, LLO *A* and LLO *B*. Initially, the algorithm will be invoked with their roots as parameters:

```
Differencing(LLO_A_root, LLO_B_root);
```

The algorithm clips LLO B into *d*LLO B without changing LLO *A*. If octant *B* is an empty node or has been removed from LLO *B*, the algorithm will return 1, otherwise, return 0. This value returned is used to remove intermediate nodes that have no descendants.

Table 3.2 Termination conditions of the differencing algorithm

Octant *A*	**Octant *B***	**Value Returned**	**Additional Actions**
E	*E*	1	*NIL*
E	*L*	0	*NIL*
E	*I*	0	*NIL*
L	*E*	0	*Replace Octant B with a pseudo empty node*
L	*L*	1	*Remove Octant B (Octants A and B are similar)*
		0	*NIL (Octants A and B are dissimilar)*
L	*I*	0	*NIL*
I	*E*	0	*Replace Octant B with a pseudo empty node*
I	*L*	0	*NIL*

The algorithm is recursive, and the termination conditions are given in Table 3.2, where *E*, *L* and *I* stand for empty, leaf and intermediate nodes respectively. The first two columns are possible states of octants *A* and *B*, and the third and fourth columns suggest the return value and additional actions needed

As suggested in Table 3.2, when Octant *B* is an empty node but Octant *A* is a non-empty node, a special pseudo empty node is generated to replace Octant *B*. It is used to remove the descendants of the working DLLO from this point while rendering. In addition, when both octants are leaf nodes, a change detection algorithm is employed to determine whether the two octants are similar. We utilize the normalized Euclidean distance between two octants for this purpose.

$$NED_{AB} = \sqrt{\sum_i \left(a_i^A - a_i^B\right)^2} \Big/ (2^r)^3 \qquad \textbf{(3.7)}$$

where a_i^A and a_i^B are the *i*th voxel intensities of octants *A* and *B* respectively, and 2^r is the edge size of the octant, where *r* is an integer between *l* and *n*. In practice, we use the normalized square Euclidean distance instead for less computation. The computation cost of this method can also be relieved by down-sampling both octants.

As discussed in section 3.2.3 on LLO Generation, when the variance of either Octant *A* or Octant *B* is no more than *v*, the sub-volume data contained in the octant(s) are not saved. Then it will not be possible to calculate the *NED* of them based on Equation 3.7, since the voxel values of at least one octant are unavailable. However, we can estimate their *NED* using Equation 3.8[5].

$$NED_{AB} = \sqrt{\frac{v_A + v_B + (m_A - m_B)^2}{\left(2^r\right)^3}} \qquad \textbf{(3.8)}$$

where v_A, m_A, v_B and m_B are the variances and means of octants *A* and *B* respectively. In this way, we avoid accessing voxels contained in both octants. Since the variance and mean values of each octant have been computed during LLO generation, the *NED* between two octants can be evaluated very fast.

If the *NED* between two octants is less than an error tolerance ε, they are regarded as similar, otherwise, dissimilar. More complicated change detection algorithm could detect sudden changes caused by noise which should be ignored.

To convert a sequence of LLOs into *d*LLOs correctly, the differencing algorithm is applied to the datasets from the last to the first. The algorithm will first be applied between LLO_{N-1} and LLO_N, and finally between LLO_1 and LLO_2. Eventually, except the first dataset that is still in the original LLO format, others are converted to *d*LLOs. The output differencing datasets can be organized and saved into a file for distribution or reuse.

If there is a high degree of data coherence between successive volumes, considerable storage space savings can be achieved by representing the 4D volume datasets with *d*LLOs.

[5] Interested readers can refer to Appendix C for more details.

3.4.3 DLLO-Based 4D Volume Rendering

With *d*LLO data, a dynamic linear level octree (DLLO) can be maintained during rendering. Initially the DLLO is the same as the first LLO dataset. The DLLO at new time steps is constructed from the current DLLO and subsequent *d*LLOs by traversing from the roots of DLLO and *d*LLO simultaneously. Suppose that an octant *A* from DLLO and an octant *B* from *d*LLO are examined currently.

- If octants *A* and *B* are both intermediate nodes, examine each of the child octants.
- If octant *B* is a pseudo empty node, octant *A* is removed.
- If octant *B* is an empty node, octant *A* is left intact, and the algorithm stops traversal of this branch.
- Otherwise, we replace octant *A* with octant *B*.

The DLLO at any specific time step is essentially an LLO. Therefore, it is possible to use the LLO-based volume rendering algorithm introduced in previous sections to visualize the DLLO and produce animated images over the time. Note that octant rendering and DLLO construction can be performed in the same traversal. The overhead of the octree traversal is thus not high.

The performance of the DLLO-based 4D volume rendering algorithm can be significantly further improved by taking advantage of the temporal coherence of the time-varying volume data in the case that the model-view transformation and transfer functions are not changed between time steps.

Rendering is accelerated by reusing intermediate volume rendered results of sub-DLLO at previous time steps. Instead of rendering and compositing all the leaf nodes into the image plane directly, partial images are saved in image buffers associated with each leaf node, where the partial images are the volume rendered images of the leaf node. If the current model-view transformation and transfer functions have not been changed since last time step, only leaf nodes from *d*LLO (except pseudo empty nodes) are re-rendered during the update of DLLO and over the change of time steps. Compositing all the partial images based on current projection, which may be a bi-linear interpolation process with little computation cost, produces the final image. Such process is repeated to update the final image output to users.

Similar to the 3D rendering algorithm, the DLLO-based 4D volume rendering algorithm is not restricted to a certain volume renderer. Most conventional volume rendering algorithms ranging from the classical CPU-based ray-casting method to the GPU-based texture-mapped method can be employed. I do not repeat the detailed discussion of them here.

This 4D volume rendering algorithm is mostly based on the assumption that not much change takes place between successive volumes and this assumption is true especially in medical imaging. For instance, a series of time-varying volume datasets which records the contrast injection procedure of brain vasculature has very little change between successive volumes. Based on this algorithm, the results obtained also supports the assumption.

3.5 Results and Discussion

A time-varying volume rendering algorithm using DLLO data structure is implemented with ray-casting and texture-mapping as the underlying renderer. All the tests were conducted using a desktop PC with an Intel Pentium IV 2.52GHz processor and 1 gigabyte system memory equipped with NVIDIA Quadro®4 700 XGL graphics card with 64 megabyte onboard memory. The experiments are to study the performance gains that can be achieved using the DLLO in terms of space reduction and speed acceleration. The experiments also studied the trade-off relations between the error tolerance and the image quality.

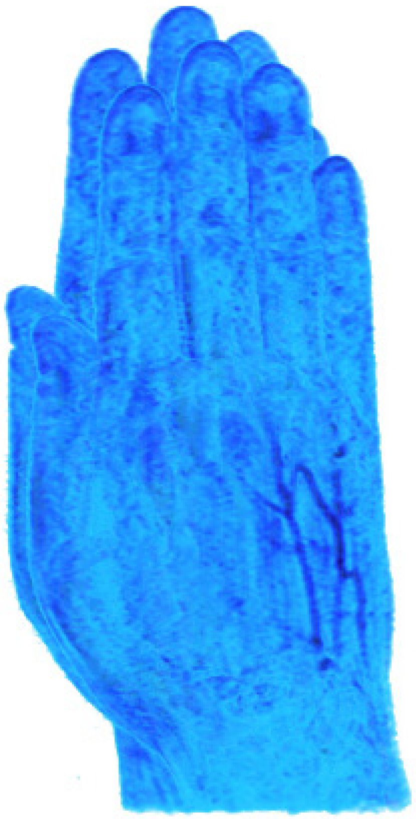
Five time-varying volume datasets were used for testing of the proposed algorithm (Table 3.3). All of them are medical data from the radiology department of a local hospital using a SIEMENS dynamic MRI system. In the first dataset, contrast dyes were used on patient's hands. It was a representative study for vascular flow malfunction. The pre and post contrast procedure include 5 time steps each of which comprises 136 4.0mm slices with an imaging matrix of 512 × 512 pixels and an in-plane resolution of 0.39 × 0.39 mm^2. The breast dataset was a representative one for investigation on breathing sequence. This MRI dataset includes 5 steps each of which comprises 26 4.2mm slices with an imaging matrix of 256 × 256 pixels and an in-plane resolution of 1.25 × 1.25 mm^2. Two heart MRI datasets were used in this testing. Both of them include 20 steps. The first one was scanned along axis each of which comprises 27 8.0mm slices with an imaging matrix of 192 × 156 pixels and an in-plane resolution of 1.67 × 1.67 mm^2. The other cardiac dataset was scanned along the short axis. Each volume comprises 16 8.0mm slices with an imaging matrix of 156 × 192 pixels and an in-plane resolution of 1.77 × 1.77 mm^2. The last dataset was a study of the urinary system of a female patient. This abdomen MRU[6] dataset includes 39 time steps each of which comprises 12 5.0mm slices with an imaging matrix of 256 × 256 pixels and an in-plane resolution of 1.02 × 1.02 mm^2. All five datasets are thus essentially anisotropic. The color plate shows the selected volume rendered images for each of the datasets using the DLLO-based ray-casting renderer.

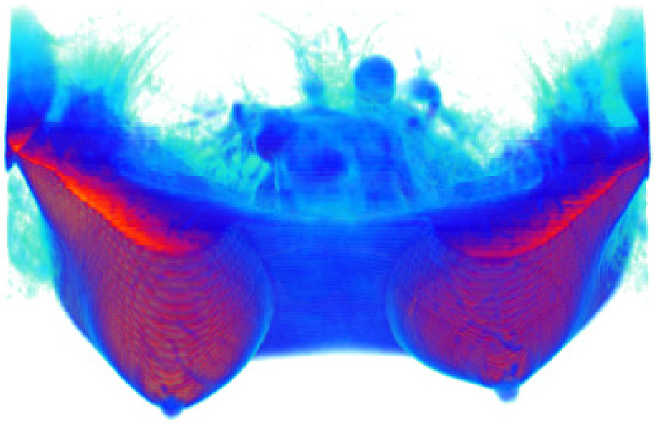
Table 3.3 Experimental time-varying volume datasets

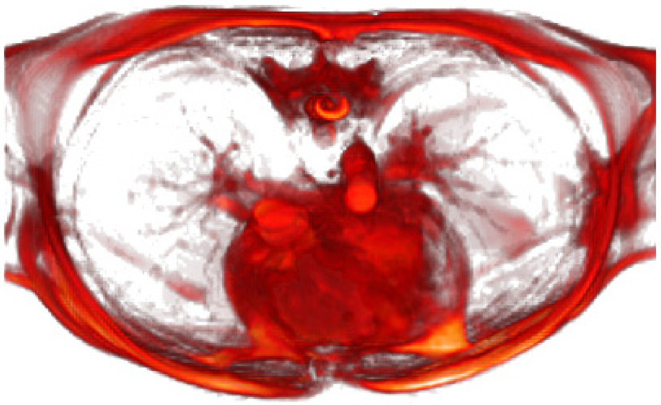
Dataset	Dimensions	Time Steps	Size (MB)	Modality
HAND	512 × 512 × 136	5	171.25	MRA
BREAST	256 × 256 × 26	5	8.13	MRI
HEART I	192 × 156 × 27	20	15.42	MRI
HEART II	156 × 192 × 16	20	9.14	MRI
ABDOMEN	256 × 256 × 12	39	29.25	MRU

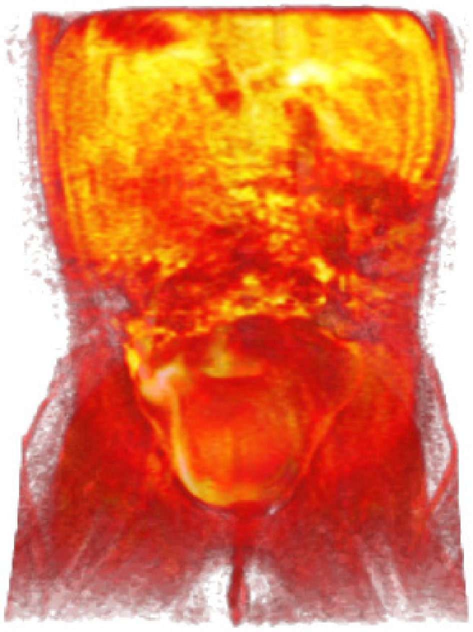
All five experimental time-varying volume datasets were converted into DLLO representations. The minimum octant size of the leaf node criteria was chosen to be 16 × 16 × 16 to detect data coherence in small regions. The octant variance threshold of the leaf node criteria, or spatial error tolerance, was set as zero in DLLO conversion. Since the spatial error tolerance can also be set dynamically during rendering, this conversion parameter will help us concentrate on analyzing the effectiveness of the proposed algorithm for 4D

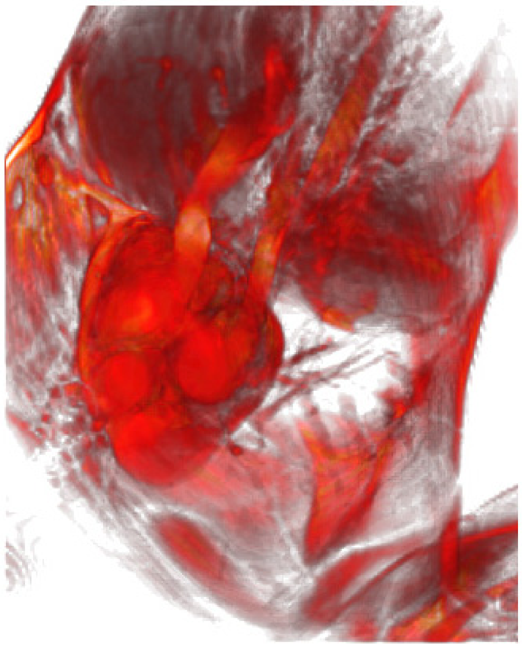
[6] Magnetic Resonance Urinography


Step 4 of *HAND*


Step 3 of *BREAST*


Step 12 of *HEART I*


Step 20 of *ABDOMEN*


Step 12 of *HEART II*

Color Plate: Selected volume rendered images using the DLLO-based renderer

rendering. We are interested to test the performance of the algorithm under different temporal error tolerances (i.e., *NED* thresholds).

The compression ratio (*R*) of a DLLO file is defined as

$$R = (1 - \frac{\text{DLLO Size}}{\text{Raw Data Size}}) \times 100\% \qquad \textbf{(3.9)}$$

which gives 100% compression if nothing remains after compression and 0% if the size remains unchanged. Note that the DLLO-based method is not meant for data compression. The compression ratio is just employed to measure the performance of the proposed method in space reduction. There are not any compression-specialized algorithms employed in the proposed method either.

Parameters used for DLLO conversion of the *HAND*, *BREAST*, *HEART I*, *HEART II* and *ABDOMEN* datasets and compression ratios achieved are shown in Table 3.4 to Table 3.8 respectively. Higher error tolerances were employed for the conversion of the datasets that are less coherent. I have attempted three different types of conversions. The conversion scheme is as labeled by *A*, *B* and *C*, and the letter is appended to the name of the dataset. For example, *Hand A*, *Hand B* and *Hand C* refer to *HAND* dataset that has undergone conversion schemes *A*, *B* and *C* respectively.

Table 3.4 DLLO conversion of the *HAND* dataset under three different temporal error tolerances (spatial error tolerance was 0.0)

Dataset Name	*NED* Threshold	Time Cost (Seconds)	Size (MB)	Compression Ratio
Hand A	0.05	23	66.29	61.0%
Hand B	0.10	22	52.79	68.9%
Hand C	0.15	21	24.63	85.5%

Table 3.5 DLLO conversion of the *BREAST* dataset under three different temporal error tolerances (spatial error tolerance was 0.0)

Dataset Name	*NED* Threshold	Time Cost (Seconds)	Size (MB)	Compression Ratio
Breast A	0.05	1	5.83	28.2%
Breast B	0.10	1	4.75	41.5%
Breast C	0.20	1	3.13	61.5%

Table 3.6 DLLO conversion of the *HEART I* dataset under three different temporal error tolerances (spatial error tolerance was 0.0)

Dataset Name	*NED* Threshold	Time Cost (Seconds)	Size (MB)	Compression Ratio
Heart I A	0.05	2	7.71	50.0%
Heart I B	0.10	2	3.72	75.9%
Heart I C	0.12	2	3.12	79.8%

Table 3.7 DLLO conversion of the *HEART II* dataset under three different temporal error tolerances (spatial error tolerance was 0.0)

Dataset Name	*NED* Threshold	Time Cost (Seconds)	Size (MB)	Compression Ratio
Heart II A	0.05	1	3.83	58.1%
Heart II B	0.08	1	2.47	72.9%
Heart II C	0.10	2	2.10	77.0%

Table 3.8 DLLO conversion of the *ABDOMEN* dataset under three different temporal error tolerances (spatial error tolerance was 0.0)

Dataset Name	*NED* Threshold	Time Cost (Seconds)	Size (MB)	Compression Ratio
Abdomen A	0.10	4	32.81	-12.2%
Abdomen B	0.15	4	22.91	21.7%
Abdomen C	0.20	4	15.84	45.8%

We observed from the tables above that the compression ratio is directly proportional to the *NED* threshold. A high compression ratio is achieved when more temporal errors can be tolerated during the conversion. The compression performance is also dependent on the temporal coherence of the dataset. It can be seen from the tables that we achieved higher overall compression ratio (up to 85%) for the *HAND* dataset due to its higher degree of temporal coherence, while the overall compression ratio is lower for the *ABDOMEN* dataset due to its lower temporal coherence. Because the size of an octant must be the same in all three dimensions[7], there may be space overhead in extending the original volume. When the space reduction gained by exploiting the data coherence is less than the space overhead, the DLLO produced could be larger than the raw data (e.g., *Abdomen A*). However, based on the testing results, we achieved satisfactory compression performance through DLLO on most medical datasets under reasonable error tolerance.

The temporal error tolerance showed little influence on DLLO construction time. The time cost of DLLO conversion is dominated by the size of the time-varying volume data. In this experiment, most DLLO conversions can be done in a few seconds. Therefore, an online interactive DLLO conversion is possible for small or medium size 4D volume datasets. A user can first select a high error tolerance in DLLO conversion for fast browsing the 4D data, and then re-convert the dataset with small error tolerance for further investigation.

The reduced size of the DLLO data will in turn benefit the rendering speed for the reduced I/O throughput and the reduced number of rendering elements. The ray-casting algorithm is implemented as the underlying renderer of the proposed DLLO-based time-varying volume rendering method. In the following, I analyze speedups of the proposed algorithm by comparing its performance with conventional ray-casting algorithm termed *regular ray-casting* algorithm. This implementation of the conventional ray-casting algorithm can be substituted with that of VTK[8] software library or other software tools. In the regular ray-casting renderer, early-ray-termination technique is employed for speed acceleration. The

[7] In Chapter 4, another time-varying volume rendering method will be introduced. It adopts a more flexible volume decomposition scheme. The space overhead is thus possibly avoided.

[8] The Visualization ToolKit (VTK) is an open source, freely available software system for 3D computer graphics, image processing, and visualization. *http://public.kitware.com/VTK*

following experiments are to analyze their relative performance. The DLLO-based renderer and the regular ray-casting renderer are implemented based on the same piece of fundamental codes. Any optimization of the implementation will improve the performance of both renderers.

The experiments are conducted based on the following procedures. After a dataset is loaded into the system, it is rendered repeatedly for 20 cycles while rendering timing of each time step is recorded. The timing results of the last 10 cycles are then averaged and reported. This design of the experiment ensures the timing results are stable and renderers can benefit from the I/O cache if possible. The performance results are illustrated in Figure 3.9 to Figure 3.13, where regular ray-casting rendering of the raw data is denoted as *Regular RC* and rendering of different DLLO datasets is denoted with the *dataset name* directly.

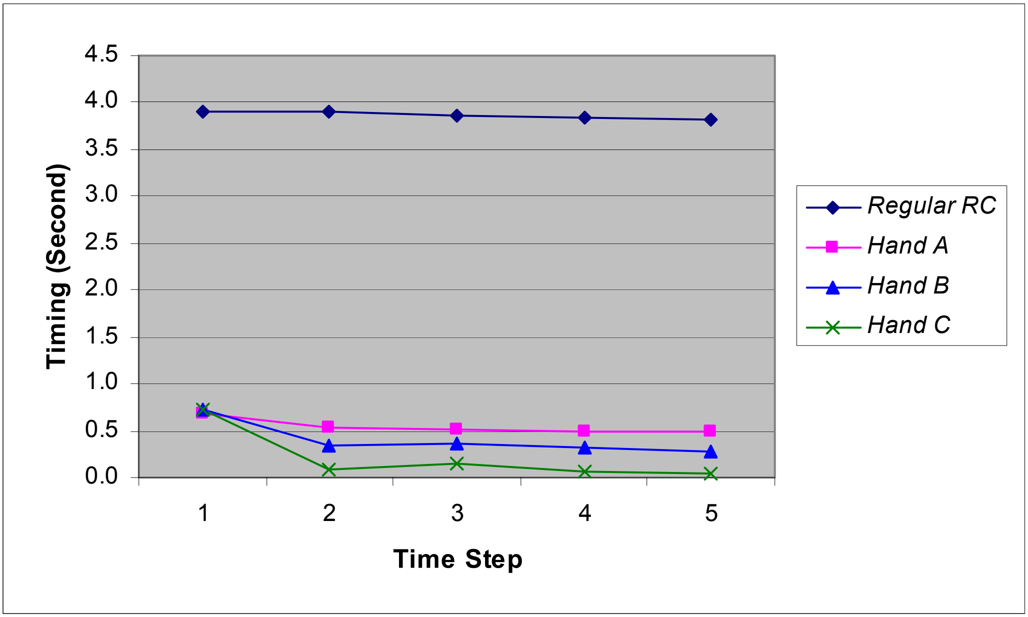


Figure 3.9 Comparison of the time-varying volume rendering speed between regular ray-casting rendering and DLLO-based rendering under three different temporal error tolerances of the *HAND* dataset

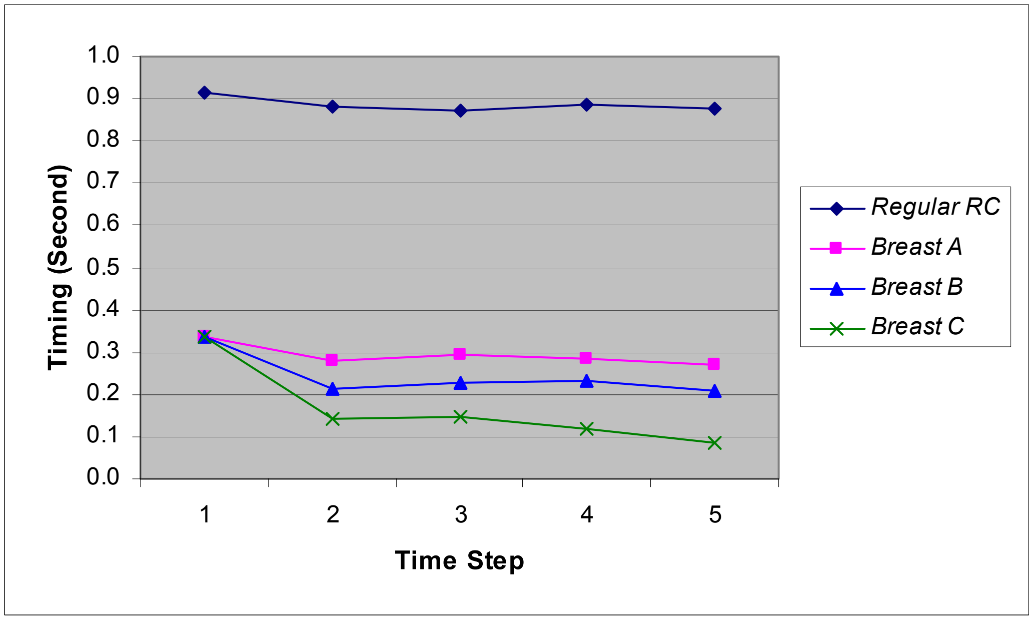


Figure 3.10 Comparison of the time-varying volume rendering speed between regular ray-casting rendering and DLLO-based rendering under three different temporal error tolerances of the *BREAST* dataset

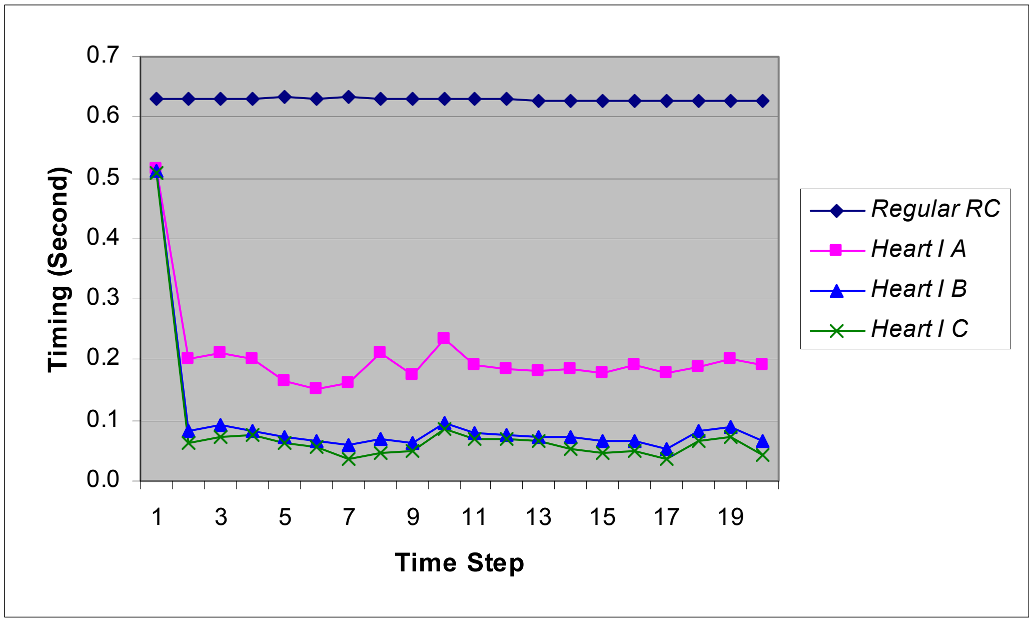


Figure 3.11 Comparison of the time-varying volume rendering speed between regular ray-casting rendering and DLLO-based rendering under three different temporal error tolerances of the *HEART I* dataset

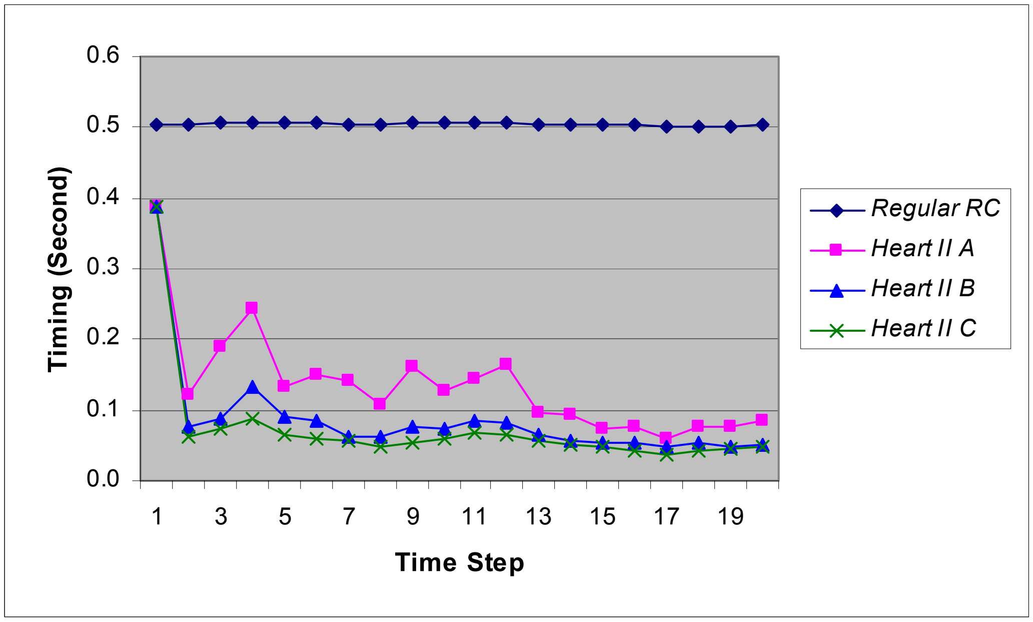


Figure 3.12 Comparison of the time-varying volume rendering speed between regular ray-casting rendering and DLLO-based rendering under three different temporal error tolerances of the *HEART II* dataset

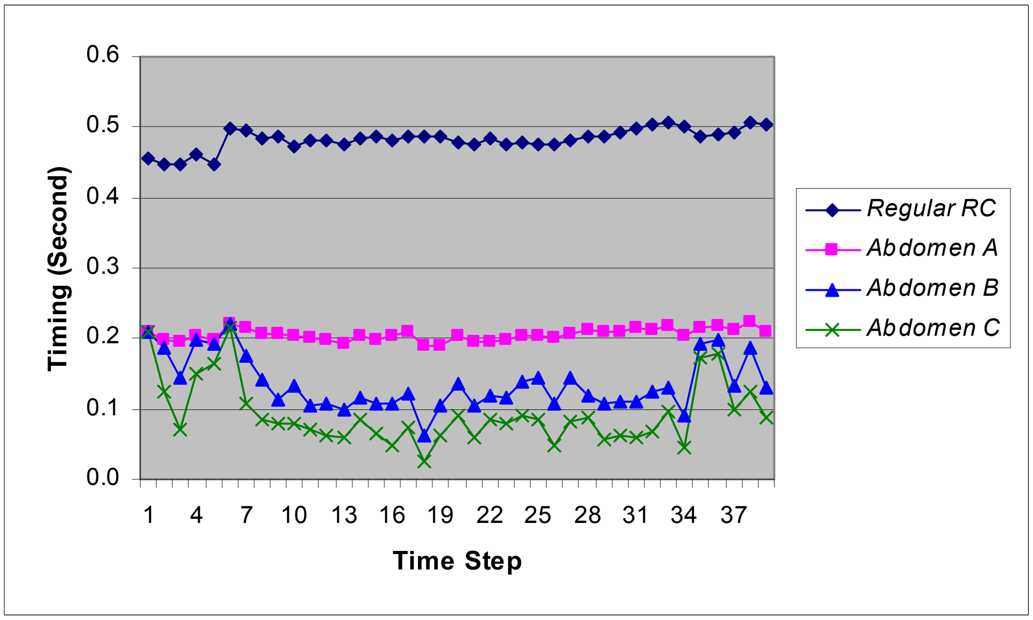


Figure 3.13 Comparison of the time-varying volume rendering speed between regular ray-casting rendering and DLLO-based rendering under three different temporal error tolerances of the *ABDOMEN* dataset

From the figures above, we can observe that, for the first time step of each time-varying dataset, a complete LLO-based rendering was performed, which cannot benefit from the temporal coherence, so it took similar time to render this step no matter what the values of the temporal error tolerance are used in the DLLO conversion. However, for the subsequent time steps, DLLO datasets with higher temporal error tolerance result in faster rendering. The rendering time of subsequent time steps is significantly shorter as compared to the first time step for datasets with high data coherence (e.g., *HAND*, *HEART I* and *HEART II* datasets). Although rendering of the first step cannot be accelerated by using temporal coherence, we still observed significant speedup for DLLO-based rendering compared with regular ray-casting rendering. The DLLO-based renderer demonstrated its remarkable speed at all time steps for all the testing datasets.

Next, I will quantitatively compare the speedup of the DLLO-based rendering method over the regular ray-casting method in terms of cycle timing which is the average of 10 rendering cycles of each dataset. As it usually takes longer time for the DLLO-based method to render the first volume step than the subsequent steps, for the 4D datasets with only a few time steps, the speed of the first time step will have much more influence on the calculation of the speedup factors than that of the 4D datasets with many time steps. Therefore, in this experiment, two types of the speedup factors are used. The first one is calculated by dividing the cycle timing of the regular ray-casting method by the cycle timing of the DLLO-based method. It shows how many times the DLLO-based method is faster than the traditional method in rendering a full cycle. The second speedup factor, denoted as *Speedup**, is also calculated based on the cycle timing but excluding the first time step. This factor serves as a more effective means to measure the performance gains through exploiting the temporal coherence and is more meaningful when comparing different datasets.

The speedup results of each dataset are shown in Table 3.9 to Table 3.13 and the comparisons of cycle timing are illustrated in Figure 3.14 to Figure 3.18, where regular ray-casting rendering is denoted as *Regular RC* and the DLLO *dataset names* are used to represent corresponding DLLO-based rendering of each dataset. The proposed algorithm is able to provide adaptive volume rendering based on user-specified error tolerances during rendering. To evaluate how much the error tolerance (τ) of the octant variance, or spatial error tolerance, can improve the performance of DLLO-based rendering, the testing results are reported under different error tolerances (τ=0 and τ=10) in this experiment.

Table 3.9 Cycle timing (in seconds) and speedup of DLLO-based rendering under different error tolerances (*HAND* dataset)

		Regular RC	*Hand A*	*Hand B*	*Hand C*
	Total Time	19.330	2.710	2.026	1.089
$\tau = 0$	**Speedup**	–	7.13	9.54	17.75
	Speedup*	–	7.63	11.85	42.73
	Total Time	19.330	2.575	1.931	0.990
$\tau = 10$	**Speedup**	–	7.51	10.01	19.53
	Speedup*	–	7.95	12.34	50.86

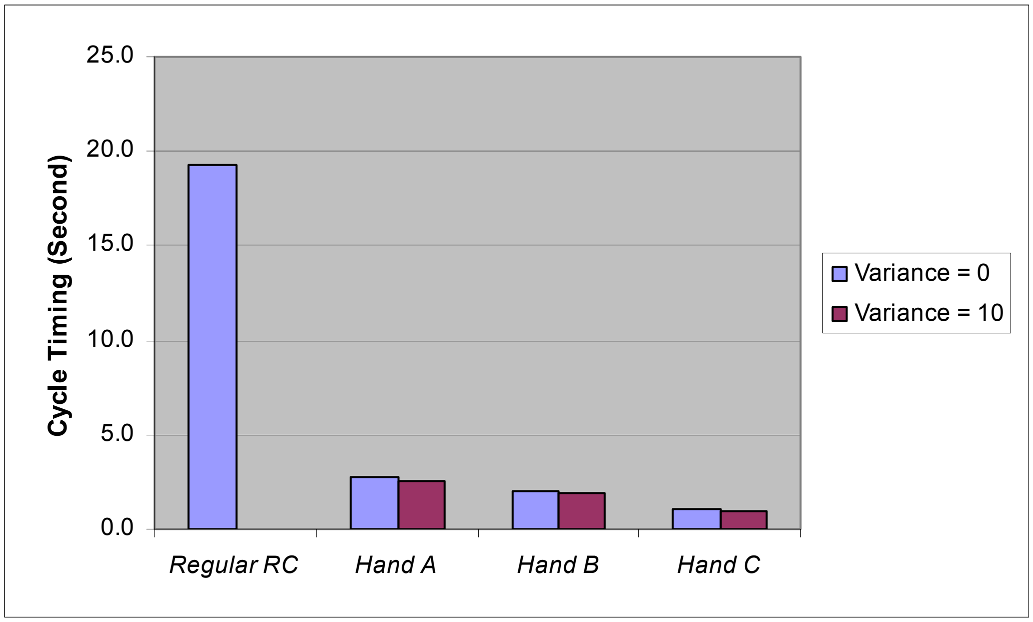


Figure 3.14 Comparison of the cycle rendering time between the DLLO-based method and the regular ray-casting method (*HAND* dataset)

Table 3.10 Cycle timing (in seconds) and speedup of DLLO-based rendering under different error tolerances (*BREAST* dataset)

		Regular RC	*Breast A*	*Breast B*	*Breast C*
	Total Time	4.433	1.464	1.212	0.831
$\tau = 0$	**Speedup**	–	3.03	3.66	5.34
	Speedup*	–	3.12	4.02	7.10
	Total Time	4.433	1.401	1.161	0.777
$\tau = 10$	**Speedup**	–	3.17	3.82	5.71
	Speedup*	–	3.18	4.07	7.31

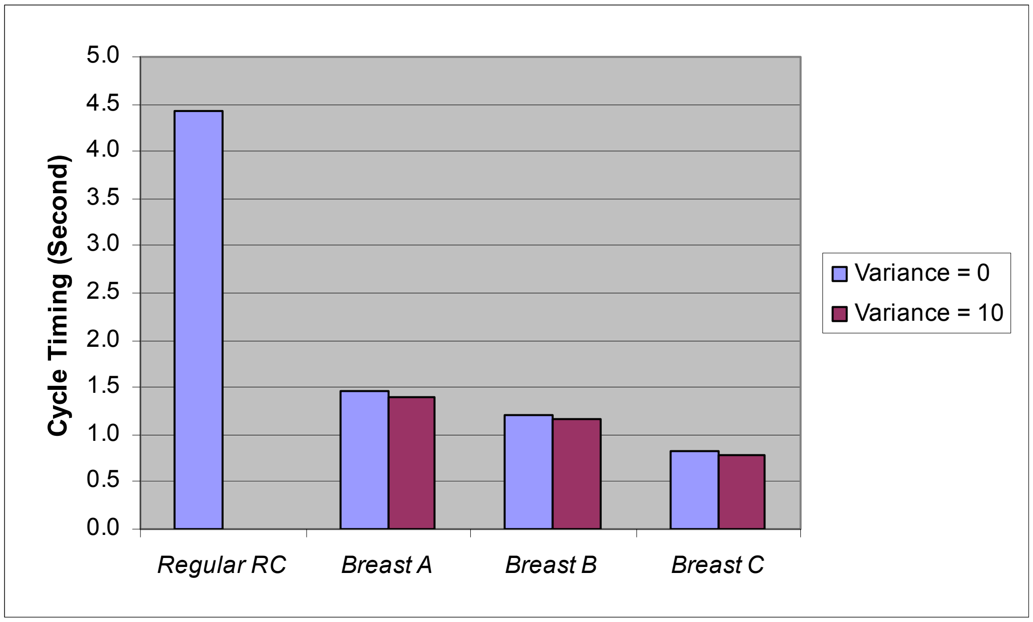


Figure 3.15 Comparison of the cycle rendering time between the DLLO-based method and the regular ray-casting method (*BREAST* dataset)

Table 3.11 Cycle timing (in seconds) and speedup of DLLO-based rendering under different error tolerances (*HEART I* dataset)

		Regular RC	*Heart I A*	*Heart I B*	*Heart I C*
	Total Time	12.600	4.107	1.916	1.628
$\tau = 0$	**Speedup**	–	3.07	6.58	7.74
	Speedup*	–	3.33	8.53	10.71
	Total Time	12.600	3.811	1.625	1.359
$\tau = 10$	**Speedup**	–	3.31	7.75	9.27
	Speedup*	–	3.54	10.00	12.92

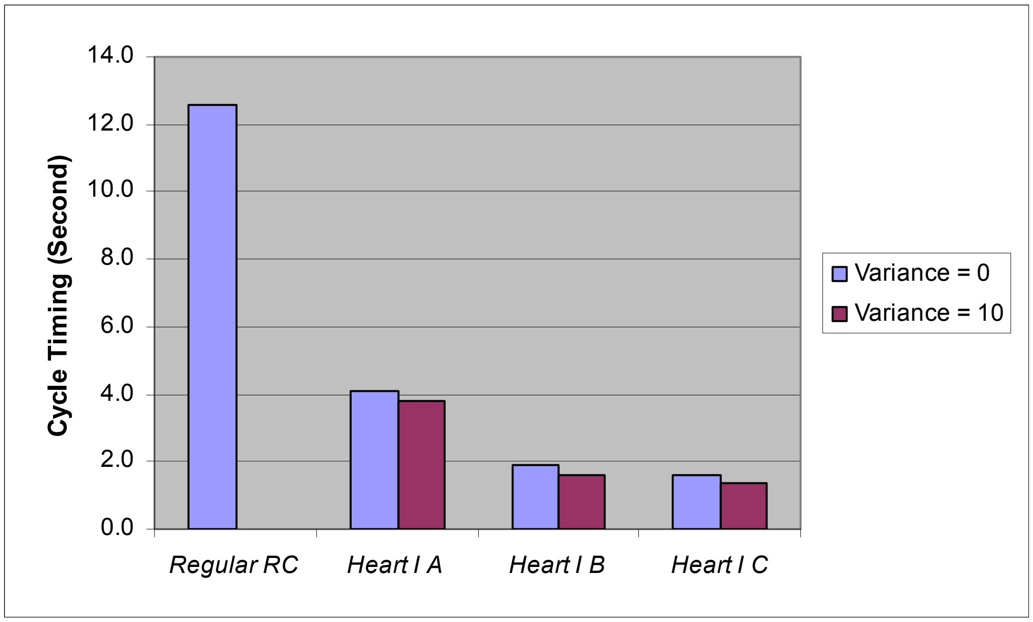


Figure 3.16 Comparison of the cycle rendering time between the DLLO-based method and the regular ray-casting method (*HEART I* dataset)

Table 3.12 Cycle timing (in seconds) and speedup of DLLO-based rendering under different error tolerances (*HEART II* dataset)

		Regular RC	*Heart II A*	*Heart II B*	*Heart II C*
	Total Time	10.092	2.707	1.724	1.458
$\tau = 0$	**Speedup**	–	3.73	5.85	6.92
	Speedup*	–	4.13	7.17	8.95
	Total Time	10.092	2.670	1.705	1.428
$\tau = 10$	**Speedup**	–	3.78	5.92	7.07
	Speedup*	–	4.17	7.21	9.08

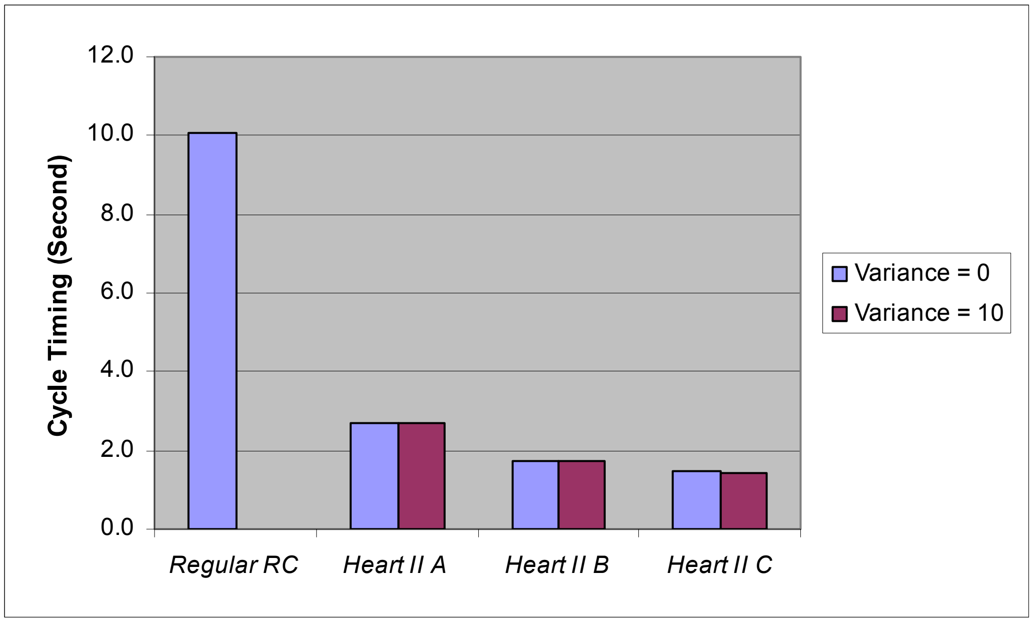


Figure 3.17 Comparison of the cycle rendering time between the DLLO-based method and the regular ray-casting method (*HEART II* dataset)

Table 3.13 Cycle timing (in seconds) and speedup of DLLO-based rendering under different error tolerances (*ABDOMEN* dataset)

		Regular RC	*Abdomen A*	*Abdomen B*	*Abdomen C*
	Total Time	18.827	8.018	5.289	3.589
$\tau = 0$	**Speedup**	–	2.35	3.56	5.25
	Speedup*	–	2.35	3.62	5.44
	Total Time	18.827	7.900	5.228	3.553
$\tau = 10$	**Speedup**	–	2.38	3.60	5.30
	Speedup*	–	2.39	3.66	5.49

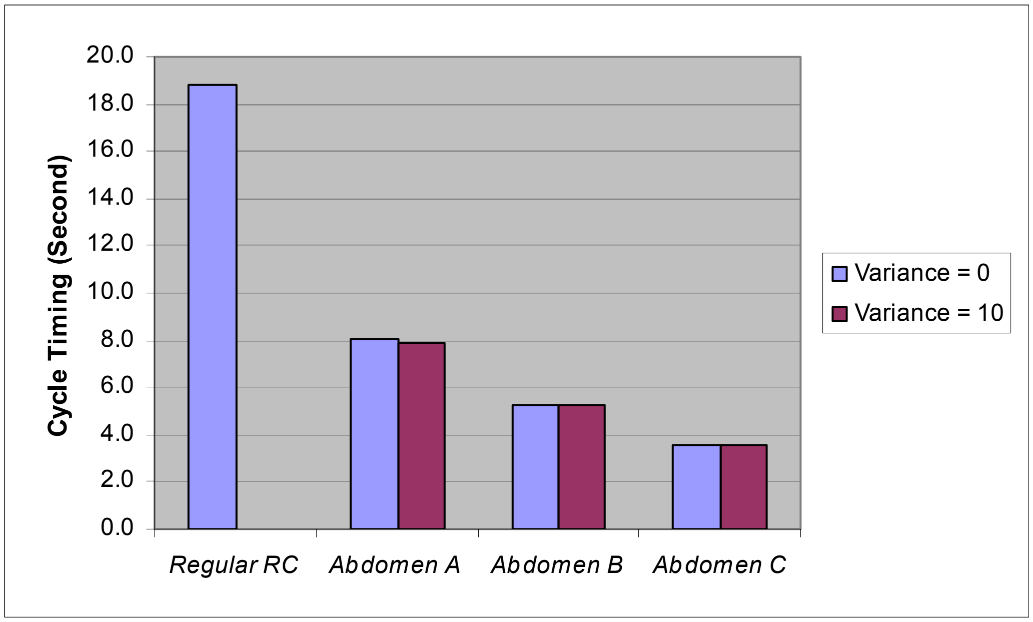


Figure 3.18 Comparison of the cycle rendering time between the DLLO-based method and the regular ray-casting method (*ABDOMEN* dataset)

The introduction of the proposed DLLO-based rendering method can speed up rendering by more than 19 times. If we consider this performance improvement by excluding the first time step, the speedup could be over 50 times. For different datasets, the speedup achieved varies for their different data coherence. But even for the testing datasets with least data coherence (e.g., *ABDOMEN* dataset), the speedup is between 2 and 5 times. The influence of the spatial error tolerance to the rendering performance is insignificant. Based on the statistics (not included), the spatial error tolerance of 10 accelerated rendering by 2% to 20% for different test datasets. This performance improvement is also dependent on the characteristics of the spatial coherence of the dataset. Overall, the DLLO-based method successfully achieves interactive rendering even with a ray-casting implementation.

To explore the possibility of embedding DLLO with the current GPU, the DLLO-based method is also implemented by using 2D texture-mapping techniques. The performance of DLLO-based rendering (τ =0) is compared with regular texture-mapped rendering. Similar to the previous experiment, the speedup results are calculated based on the averaged cycle timings and are given in Table 3.14 to Table 3.18, where regular texture-mapped rendering is denoted as *Regular TM* and the DLLO *dataset names* are used to represent corresponding DLLO-based rendering of each dataset.

We can observe that the rendering speed is significantly enhanced by taking advantage of GPU techniques. Compared to regular texture-mapped rendering, the DLLO-based method achieves high speedup and real-time rendering is successfully fulfilled for most of the datasets.

Table 3.14 Cycle timing (in seconds) and speedup results of DLLO-based rendering using 2D texture-mapping based on *HAND* dataset

	Regular TM	*Hand A*	*Hand B*	*Hand C*
Total Time	30.339	3.744	3.062	1.619
Speedup	–	8.103	9.907	18.743
Speedup*	–	8.302	10.838	30.712

Table 3.15 Cycle timing (in seconds) and speedup results of DLLO-based rendering using 2D texture-mapping based on *BREAST* dataset

	Regular TM	*Breast A*	*Breast B*	*Breast C*
Total Time	1.614	0.307	0.263	0.196
Speedup	–	5.264	6.144	8.232
Speedup*	–	5.408	6.627	10.055

Table 3.16 Cycle timing (in seconds) and speedup results of DLLO-based rendering using 2D texture-mapping based on *HEART I* dataset

	Regular RC	*Heart I A*	*Heart I B*	*Heart I C*
Total Time	1.165	0.455	0.268	0.241
Speedup	–	2.561	4.340	4.824
Speedup*	–	2.726	5.025	5.724

Table 3.17 Cycle timing (in seconds) and speedup results of DLLO-based rendering using 2D texture-mapping based on *HEART II* dataset

	Regular RC	*Heart II A*	*Heart II B*	*Heart II C*
Total Time	0.670	0.217	0.150	0.132
Speedup	–	3.088	4.466	5.083
Speedup*	–	3.386	5.264	6.226

Table 3.18 Cycle timing (in seconds) and speedup results of DLLO-based rendering using 2D texture-mapping based on *ABDOMEN* dataset

	Regular RC	*Abdomen A*	*Abdomen B*	*Abdomen C*
Total Time	6.044	1.369	0.966	0.694
Speedup	–	4.417	6.258	8.713
Speedup*	–	4.426	6.348	8.984

Since error tolerance was used in the DLLO conversion of the time-varying volume data, it is necessary to analyze the resultant visual quality. The regression testing method from VTK is employed for this purpose. The regression testing compares a test image that is produced with an algorithm being evaluated with a "valid" image that is assumed to be correct. The comparison takes into account dithering and anti-aliasing effects, and creates an output image representing the difference between the test image and valid image [Schroeder et al. 1998]. The difference of the two images is quantified in terms of the *absolute error* (E_A) and *thresholded error* (E_T), which are calculated based on the equations below:

$$D_i = \frac{\left|r_i^v - r_i^t\right| + \left|g_i^v - g_i^t\right| + \left|b_i^v - b_i^t\right|}{3}$$
$$E_A = \frac{\sum_i D_i}{L_c - 1} \qquad \textbf{(3.10)}$$

$$A_i = \begin{cases} D_i - T & D_i - T > 0 \\ 0 & \mathit{Otherwise} \end{cases}$$
$$E_T = \frac{\sum_i A_i}{L_c - 1} \qquad \textbf{(3.11)}$$

where (r_i^v, g_i^v, b_i^v) and (r_i^t, g_i^t, b_i^t) are the *i*th color pixel value (red, green and blue) of the valid image and test image respectively; D_i is the difference of the *i*th pixel between the two images; L_c is the number of color levels of a channel; T is a threshold tolerance for pixel differences. Thus, the absolute error is actually the total error in comparing the two images, and the thresholded error is the error for a given pixel minus the threshold and clamped at a minimum of zero. The latter will be more effective in representing the noticeable differences between two images.

In this implementation, all the images are generated in color with red, green and blue channels, and each channel has 8 bits, i.e., $2^8 = 256$ levels ($L_c = 256$). To avoid misunderstanding of images with the introduction of pseudo-colors[9], pixels are assigned with the same value for all three channels and the images thus appear in gray. A threshold

[9] The radiology images used in this thesis all do not have intrinsic color values assigned to the voxels, only intensities. Therefore, an image rendered in color can only be achieved through mapping of pseudo-colors.

tolerance of 5 is used in the analysis of the image quality. It is less than 2% of the maximum pixel difference and normally is not noticeable by human eyes.

The images generated by the regular ray-casting method are employed as the valid images, and the quality of DLLO-based rendering is evaluated based on the following procedures. For each dataset, the regression testing is applied to a pair of corresponding images of each time step and the E_A and E_T of this time step are calculated. After the regression testing is finished for all time steps, results are averaged. The averaged E_A and E_T are then used to represent the error of DLLO-based rendering for this dataset. Based on the images generated by the regular ray-casting method, the regression testing is also applied between the images at successive time steps. The testing results are averaged and used to indicate the inter-step differences. This value provides us an effective reference to evaluate the rendering quality we have achieved. The inter-step error also serves as a good index of the data coherence of the dataset.

The error analysis of DLLO-based rendering was performed to all the testing datasets and the results are shown in Table 3.19 to Table 3.23. It should be noted that all the images must be generated in the same size so that the regression errors are comparable with each other. In this experiment, all the images have size of 500 × 500 pixels.

Table 3.19 Error analysis of DLLO-based rendering of *HAND* dataset

Name	**Absolute Error (E_A)**	**Thresholded Error (E_T)**
Inter-Step	444.723	63.057
Hand A	0.634	0.001
Hand B	13.089	0.976
Hand C	86.131	16.345

Table 3.20 Error analysis of DLLO-based rendering of *BREAST* dataset

Name	**Absolute Error (E_A)**	**Thresholded Error (E_T)**
Inter-Step	1370.249	683.178
Breast A	3.664	0.227
Breast B	40.862	3.823
Breast C	168.292	27.682

Table 3.21 Error analysis of DLLO-based rendering of *HEART I* dataset

Name	**Absolute Error (E_A)**	**Thresholded Error (E_T)**
Inter-Step	302.296	42.240
Heart I A	18.232	0.811
Heart I B	66.057	18.032
Heart I C	90.236	29.351

Table 3.22 Error analysis of DLLO-based rendering of *HEART II* dataset

Name	Absolute Error (E_A)	Thresholded Error (E_T)
Inter-Step	661.847	119.967
Heart II A	84.436	12.684
Heart II B	168.240	54.908
Heart II C	193.506	70.650

Table 3.23 Error analysis of DLLO-based rendering of *ABDOMEN* dataset

Name	Absolute Error (E_A)	Thresholded Error (E_T)
Inter-Step	2616.277	1526.589
Abdomen A	3.697	0.125
Abdomen B	209.025	51.155
Abdomen C	549.032	200.331

It is clear that the DLLO-based algorithm achieves very good rendering quality as all the results of error analysis are small compared to inter-step difference. The rendering quality does not degrade much when the error tolerances for DLLO conversion increase. The inter-step errors also provide us a good way to evaluate the degree of data coherence. The *HAND* dataset has the highest degree of data coherence for the smallest inter-step error while the *ABDOMEN* dataset has the lowest degree of data coherence for the largest inter-step error. This coincides with the performance difference in speed acceleration and space reduction for different datasets in that the DLLO-based rendering algorithm is data coherence dependent. Selected images are shown in Figure 3.19 to Figure 3.23. The loss of the image quality is visually tolerable for all the test datasets. The quantified errors of each pair of images are given in the figures as well.

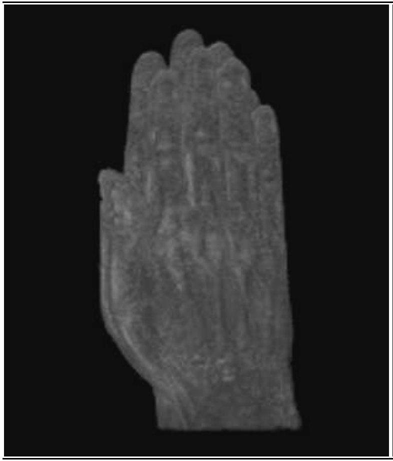
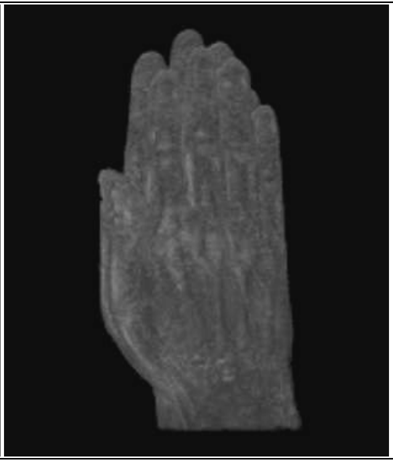
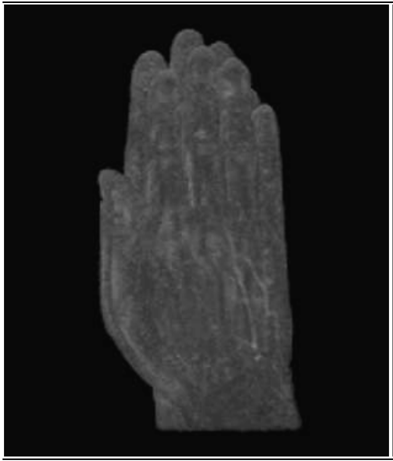
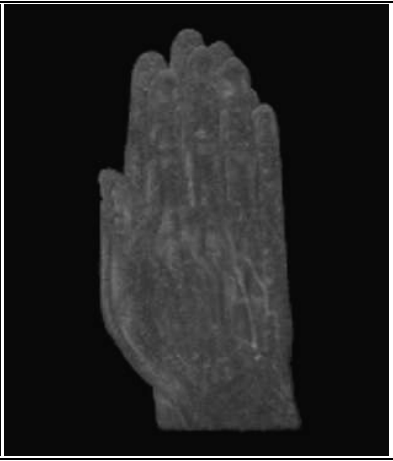
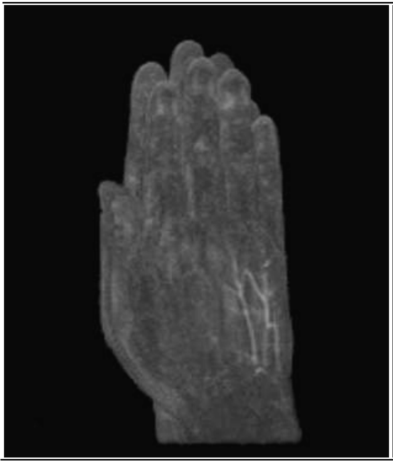
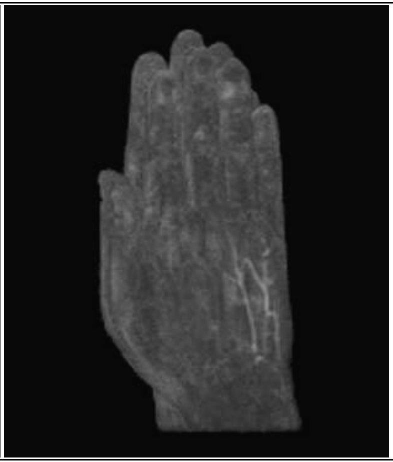
(a) Valid image	**(b) DLLO rendered image**	**(c) Regression Error**
		**E_A = 0.00** **E_T = 0.00** **(Step 1)**
		**E_A = 7.25** **E_T = 0.06** **(Step 3)**
		**E_A = 35.1** **E_T = 3.80** **(Step 5)**

Figure 3.19 Comparison of the image quality between regular ray-casting and DLLO-based rendering of the *HAND* dataset (*NED* Threshold = 0.1)

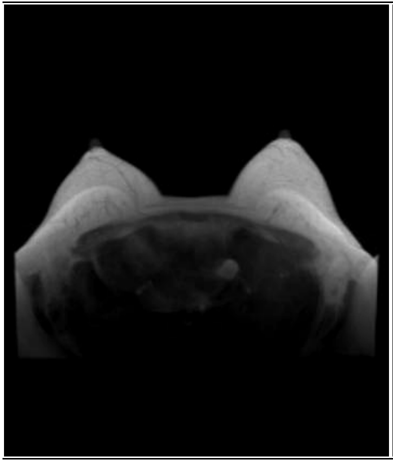
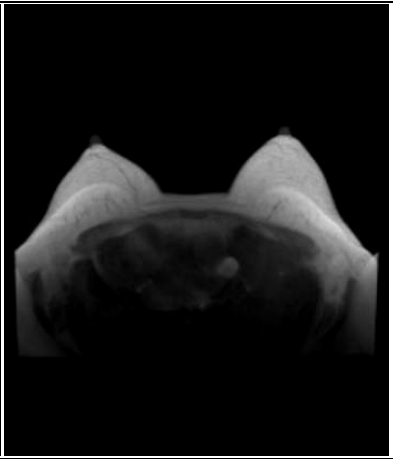
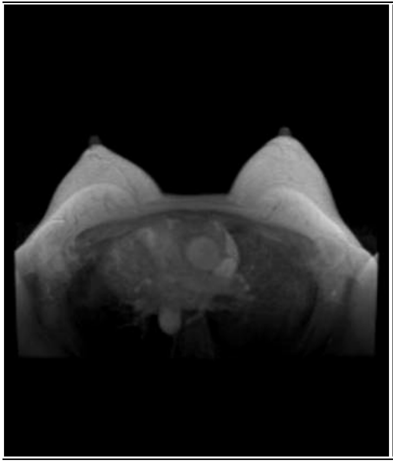
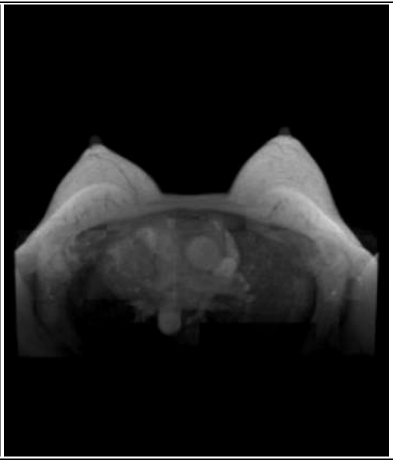
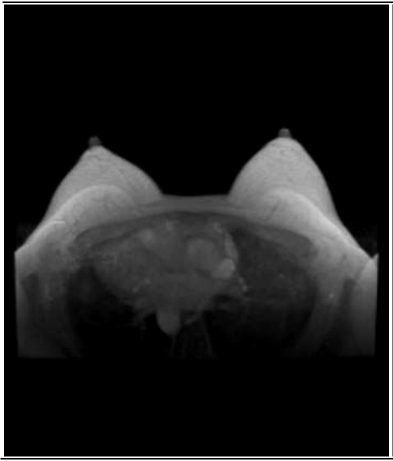
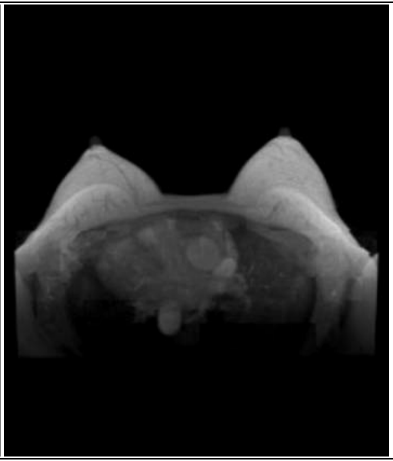
(a) Valid image	**(b) DLLO rendered image**	**(c) Regression Error**
		$\boldsymbol{E_A}$ **= 0.00** $\boldsymbol{E_T}$ **= 0.00** **(Step 1)**
		$\boldsymbol{E_A}$ **= 195** $\boldsymbol{E_T}$ **= 32.2** **(Step 3)**
		$\boldsymbol{E_A}$ **= 308** $\boldsymbol{E_T}$ **= 54.7** **(Step 5)**

Figure 3.20 Comparison of the image quality between regular ray-casting and DLLO-based rendering of the *BREAST* dataset (*NED* Threshold = 0.2)

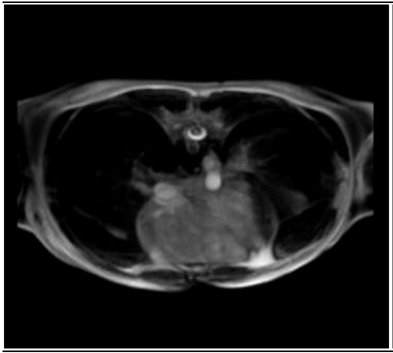
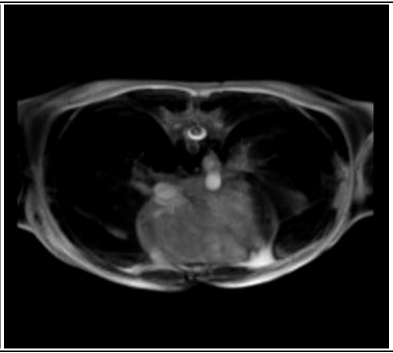
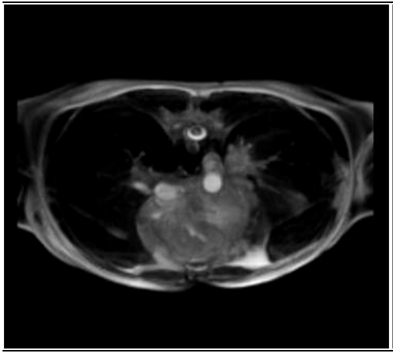
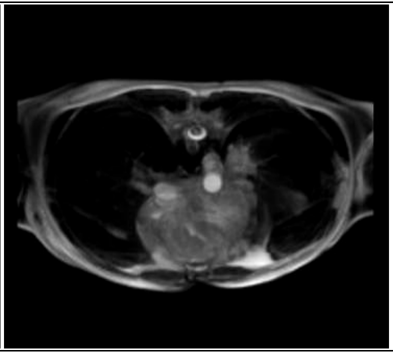
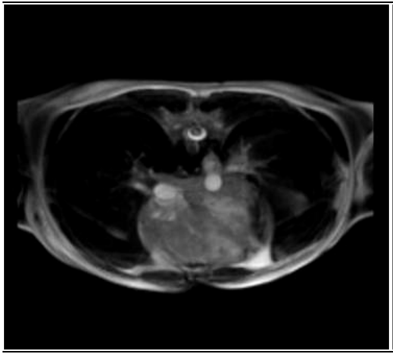
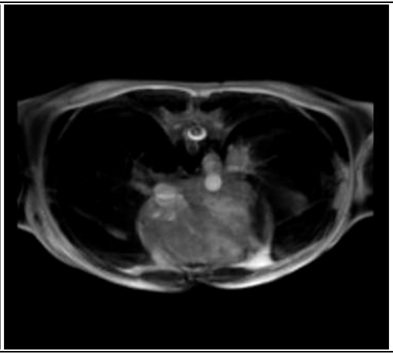
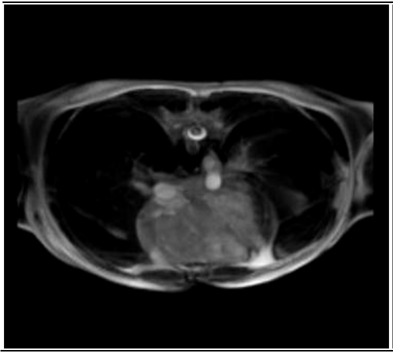
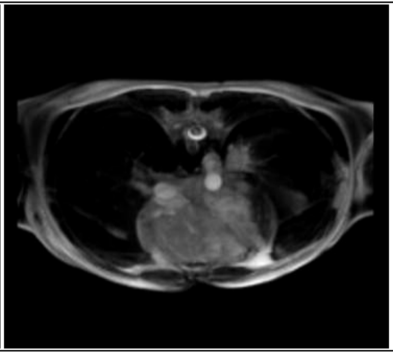
(a) Valid image	**(b) DLLO rendered image**	**(c) Regression Error**
		***E_A* = 0.00** ***E_T* = 0.00** **(Step 1)**
		***E_A* = 124** ***E_T* = 41.2** **(Step 7)**
		***E_A* = 90.5** ***E_T* = 33.2** **(Step 13)**
		***E_A* = 92.9** ***E_T* = 33.2** **(Step 20)**

Figure 3.21 Comparison of the image quality between regular ray-casting and DLLO-based rendering of the *HEART I* dataset (*NED* Threshold = 0.12)

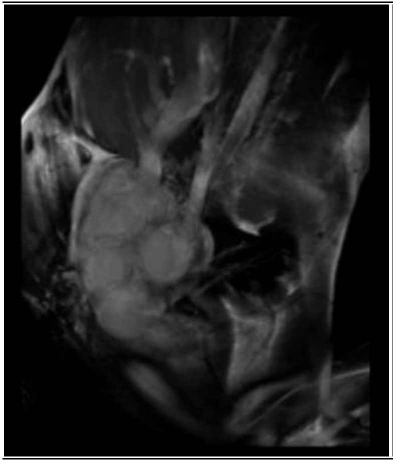
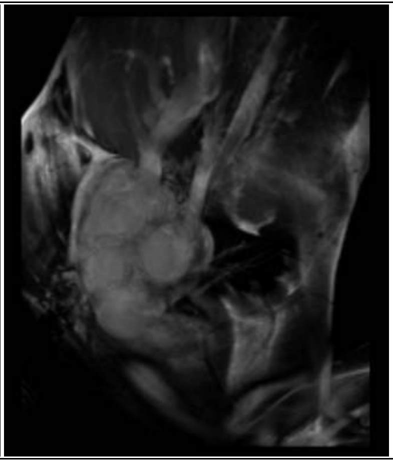
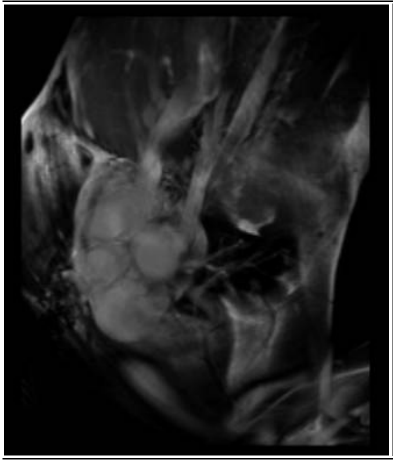
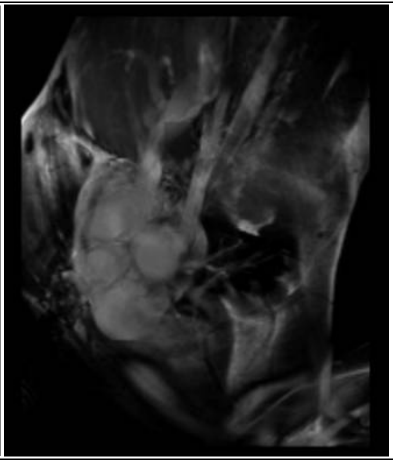
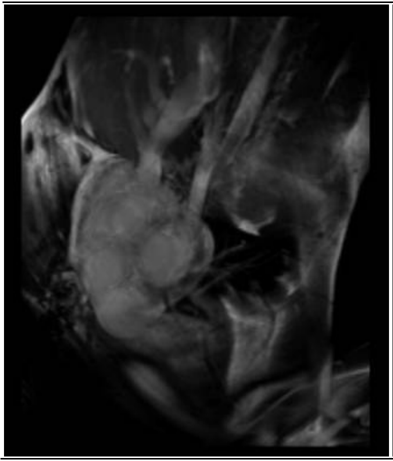
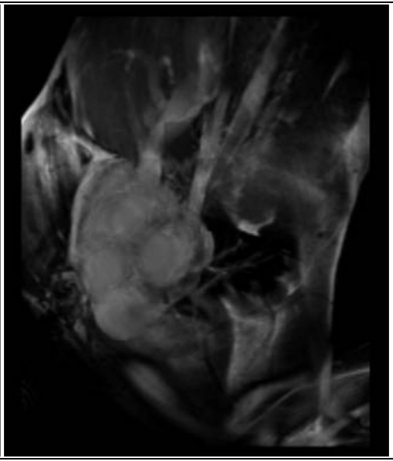
(a) Valid image	**(b) DLLO rendered image**	**(c) Regression Error**
		***E_A* = 0.00** ***E_T* = 0.00** **(Step 1)**
		***E_A* = 203** ***E_T* = 62.9** **(Step 11)**
		***E_A* = 168** ***E_T* = 58.8** **(Step 20)**

Figure 3.22 Comparison of the image quality between regular ray-casting and DLLO-based rendering of the *HEART II* dataset (*NED* Threshold = 0.08)

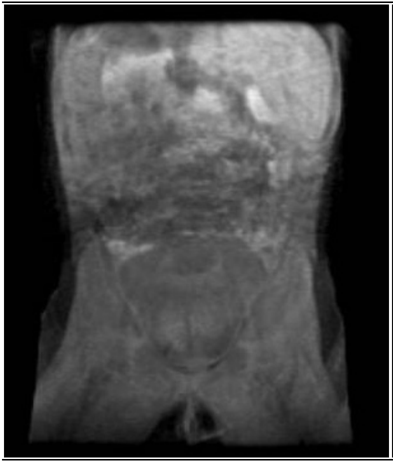
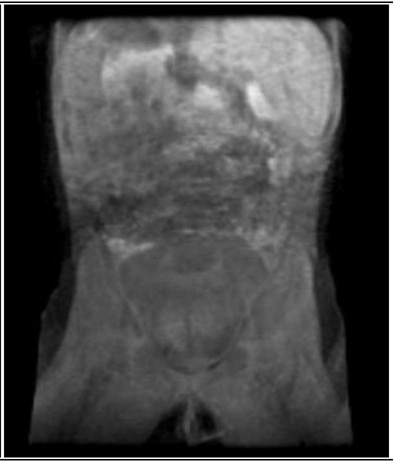
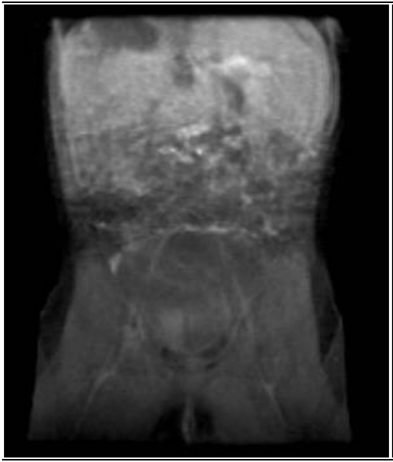
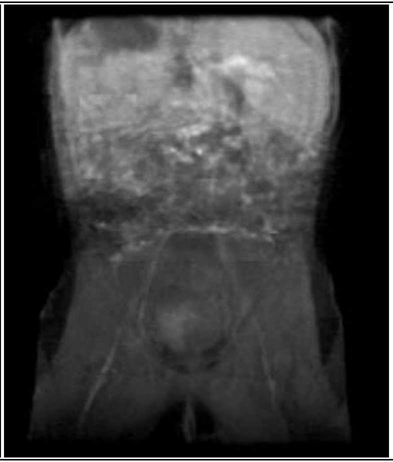
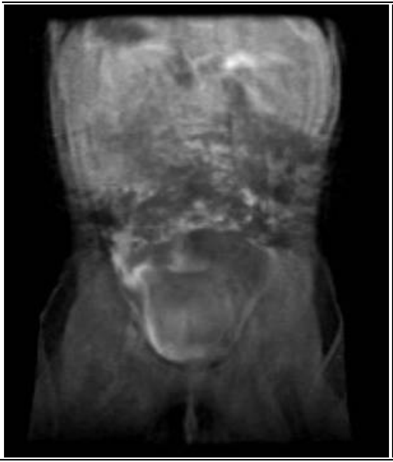
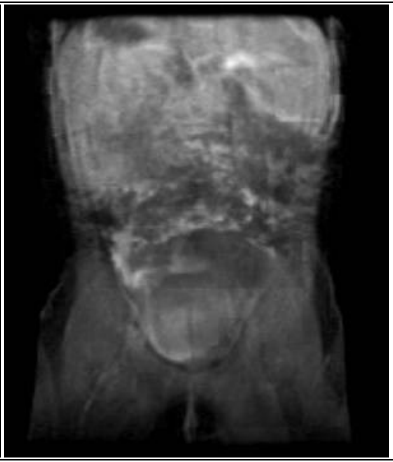
(a) Valid image	**(b) DLLO rendered image**	**(c) Regression Error**
		**E_A = 0.00** **E_T = 0.00** **(Step 1)**
		**E_A = 681** **E_T = 190** **(Step 19)**
		**E_A = 420** **E_T = 135** **(Step 39)**

Figure 3.23 Comparison of the image quality between regular ray-casting and DLLO-based rendering of the *ABDOMEN* dataset (*NED* Threshold = 0.2)

3.6 Summary

In this chapter, I first introduced a spatial data structure, namely linear level octree, which is more computationally efficient than conventional octree. It is employed for the acceleration of steady-state volume rendering, and adaptive rendering can be provided with user-specified error tolerances. Based on LLO, a new data structure called dynamic linear level octree was proposed to represent the time-varying volume data and accelerate 4D rendering. The DLLO takes advantage of the spatial and temporal coherence of the time-varying volume data to provide interactive or real-time 4D volume rendering for PC-based medical simulations. Since it has no restriction on the underlying renderers, most conventional 3D volume rendering methods can be mended for 4D rendering with enhanced performance through the use of DLLO.

In comparison with the regular ray-casting algorithm and the regular texture-mapped rendering algorithm, the proposed DLLO-based rendering method presented fast rendering speed and reduced space requirement. The compression ratio obtained could be up to 85% and more than 19 times speedup can be achieved. The DLLO-based algorithm also achieves a good rendering quality according to the regression testing results as well as through the visual inspection.

In addition, the proposed LLO-based volume rendering algorithm is also capable of supporting multimodality rendering. Interested reader can refer to Appendix B for more details.

Chapter 4

Cluster-Based Time-Varying Volume Rendering

4.1 Introduction

The problem of rendering time-varying volume data is challenging. Traditionally, algorithms of 3D volume rendering are to process and encode the volume data into other representations or use auxiliary data structures before every rendering. The overall performance has been improved even though the online data preprocessing takes time. As compared with the steady-state 3D volumes, the size of time-varying volume data increases dramatically. The expense of preprocessing online ten or even hundred times of data is no longer affordable in 4D rendering. Moreover, the preprocessing is over-sophisticated and tedious for common users such as clinicians, geologists or artists. The users are mainly concerned with the rendering results.

An offline data preprocessing, therefore, will be more suitable for time-varying volume rendering. An appropriate framework for time-varying volume rendering could be based on the raw volume data at all times that have been encoded once in a preprocessing stage, and subsequent rendering could be viewed as decoding the data. This mechanism is indeed very similar to that of video compression and playing. For example, the unprocessed data in video and time-varying volume are both very large so that compression is normally expected. Scenes between successive frames or volumes usually have similar contents. Fast playing or rendering is commonly required. Therefore, the techniques in video processing could be useful in time-varying volume rendering.

In the basic scheme of the MPEG[10] standard, motion is predicted from frame to frame in the temporal direction, and DCTs (discrete cosine transforms) are used to organize the redundancy in the spatial directions. The similar scheme can also be introduced in 4D volume rendering to resolve the data redundancy problem by exploiting the coherence of data in both spatial and temporal dimensions.

This chapter describes a new 4D volume rendering algorithm to exploit redundant regions of time-varying data via cluster analysis of the spatial block decomposition of a multi-dimensional dataset. Besides the exploitation of the spatial coherence and temporal coherence, this algorithm takes advantage of the global coherence of the time-varying data,

[10] MPEG is the *Moving Picture Experts Group*, working under the joint direction of the International Standards Organization (ISO) and the International Electro-Technical Commission (IEC). This group works on standards for the coding of moving pictures and associated audio. *http://www.mpeg.org*

which is a new concept introduced. Experiments and results in the last section demonstrate its superiority over other algorithms for time-varying volume rendering.

4.2 Overview of the Algorithm

Figure 4.1 shows the framework of the proposed time-varying volume rendering algorithm where the data preprocessing procedure named encoding stage is separated from the rendering stage. The encoding of the 4D volume data thus can be performed offline. It may take a long time with a sophisticated encoder, but the output can provide a satisfactory rendering.

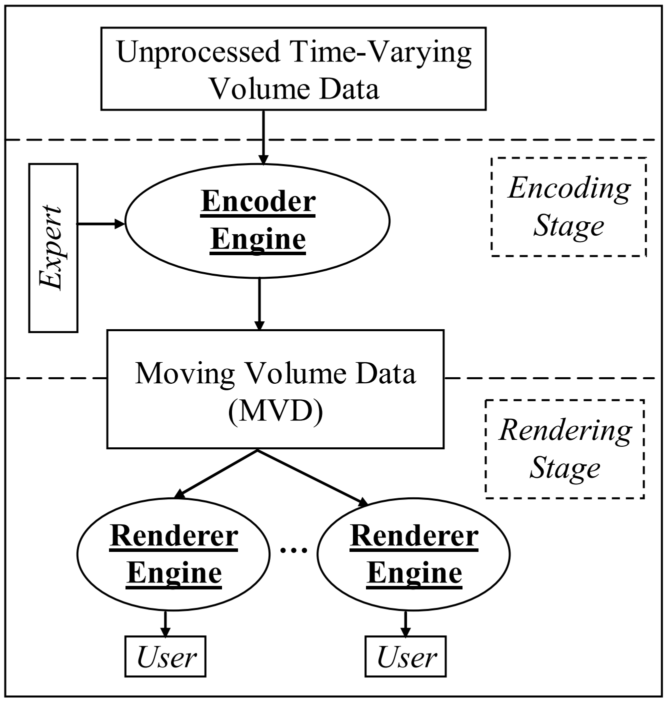


Figure 4.1 The Framework of time-varying volume rendering

In the encoding stage, a time-varying volume dataset is passed to an encoder engine and converted into an encoded volume, named *Moving Volume Data* (MVD), under the supervision of a 4D volume-processing specialist. The expert decides on the encoder and its corresponding parameters such as error tolerance based on the application need. In the encoder engine, a cluster analysis is applied to exploit natural coherence in the time-varying dataset and an MVD file is produced and passed to the next stage.

Since an MVD is produced as a file, it is reusable and distributable to users. In the rendering stage, an MVD is rendered as a decoding process with an MVD-based renderer engine. Users are then free to use their client renderers to view and manipulate the time-varying volume data at any time.

This framework is actually independent on the specific encoder and renderer (decoder). It is generally suitable for time-varying volume rendering with offline data processing. Various pairs of 4D volume coding/decoding (codec) algorithms can be developed and employed. For an encoded 4D volume, the corresponding decoder is automatically selected and serves as a plug-in of the user's renderer. Similar to the scheme used in video codec, this process could be transparent to users viewing 4D rendering.

In the following sections, I will describe the algorithms used in the encoder and its corresponding renderer as part of my solution for fast time-varying volume rendering.

4.3 Encoding

To convert a time-varying volume dataset into an MVD, the volumes are processed in three steps, namely division, clustering and data output as shown in Figure 4.2. They will be introduced in the following sub-sections.

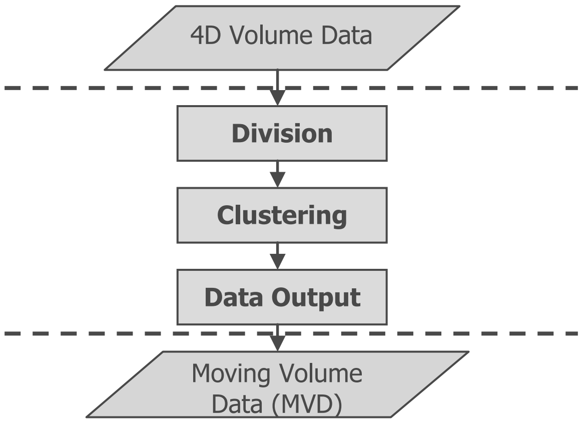


Figure 4.2 Flowchart of the encoding process

4.3.1 Division

Volumes are collections of points with intensity values, i.e., voxels, arranged on a rectangular lattice. The rows, columns and planes of the lattice are parallel to the global x-y-z coordinate system [Schroeder et al. 1998]. For a 3D volume dataset, the number of voxels in each dimension is assumed to be the same for simplicity in the following description, although they are variable in most datasets.

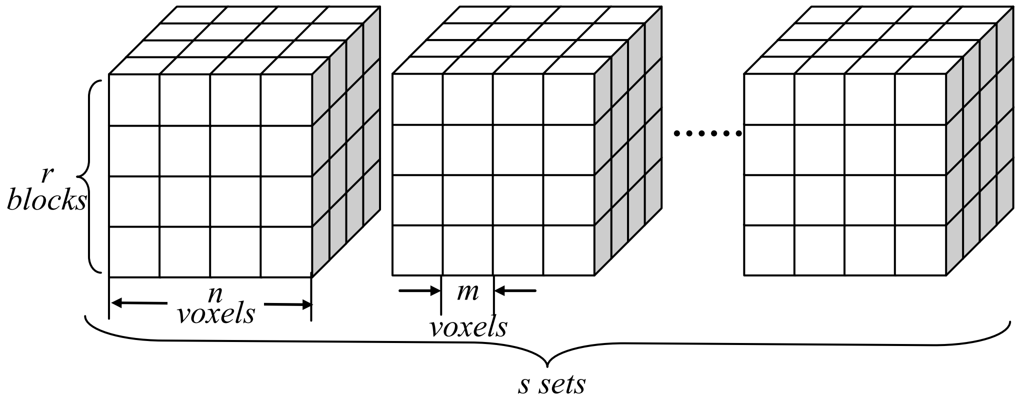


Figure 4.3 Division of time-vary volume data

A time-varying volume dataset usually contains a sequence of 3D volumes numbered from 1 to *s* with *n* voxels in each dimension as illustrated in Figure 4.3. Each 3D volume is uniformly divided into r^3 blocks, while each block contains $M = m^3$ voxels, i.e., $n = r \times m$. Then the time-varying volume dataset is divided into $S = r^3 \times s$ blocks. To keep the memory requirement low, the size (*m*) of the blocks should be carefully selected based on the size (*n*) of the volume.

4.3.2 Clustering

Clustering technique is widely used in data mining to group items. The goal is to partition a set of entities into groups such that entities within a group are similar to each other and entities that belong to two different groups are dissimilar [Ramakrishnan and Gehrke 2000]. Each of the groups is called a cluster. The similarity between two entities is measured by a *distance function* which takes two input entities and returns a value that is a measure of their similarity. Usually, the output of a clustering algorithm consists of a summarized representation of each cluster and its size.

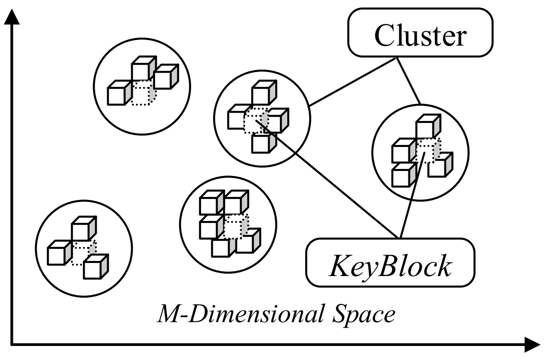


Figure 4.4 Clusters of blocks in *M*-dimensional space

The clustering technique is employed to exploit the homogeneousness of the 4D volume data. The blocks from all time steps are divided and grouped into different clusters, and each cluster is summarized by its center (also called mean) and radius. In each cluster of blocks, a *KeyBlock* is generated to represent the cluster. An example of clusters is shown in Figure 4.4. Note that a *KeyBlock* is generated by considering all the contributions of the blocks in a cluster. It is not necessary to be any one of the existing blocks.

Let us denote block *i* as an *M*-dimensional vector:

$$\begin{gathered} Block_i = \begin{bmatrix} a_i^1 & a_i^2 & \cdots & a_i^k & \cdots & a_i^M \end{bmatrix} \\ (M = m^3, i \in [1, S],\ k \in [1, M],\ a_i^k \in [0, 2^b - 1]) \end{gathered} \tag{4.1}$$

where a_i^k is the intensity value of the *k*th voxel in block *i*, and *b* is the number of bits used to store a voxel intensity.

The distance function between two blocks *Block_i* and *Block_j* is defined as the following:

$$\begin{aligned} D_{ij} = D_{ji} &= \frac{\left\| Block_i - Block_j \right\|}{M} \\ &= \frac{\sqrt{(a_i^1 - a_j^1)^2 + (a_i^2 - a_j^2)^2 + \cdots + (a_i^M - a_j^M)^2}}{M} \end{aligned} \tag{4.2}$$

which is actually the normalized Euclidean distance (*NED*) of the two blocks. In practice, we use the normalized square of the Euclidean distance instead because it involves less computation.

The distance can be estimated more efficiently with smaller number of dimensions for the price of accuracy. A common dimension-selection scheme can be employed for dimension reduction. For instance, we access voxel values at only even-number dimensions of each block for faster distance estimation. A random dimension-selection scheme is also effective. The dimensions can also be reasonably reduced by applying the principal component analysis (PCA) on all the blocks. Only the voxels at the first *x* most principal dimensions are accessed and evaluated.

The distance functions can also be defined in some other ways such as the one given in Equation 4.3. However, in this chapter, we are constrained to use the *NED* for the similarities measurement.

$$D_{ij} = D_{ji} = size(\{k \,|\, |a_i^k - a_j^k| > \varepsilon, k \in [1, M]\}) \qquad \textbf{(4.3)}$$

With the distance function, blocks can be organized in clusters. We name the set of all blocks as *RSet* and use BIRCH (Balanced Iterative Reducing and Clustering using Hierarchies) algorithm [Zhang et al. 1996] to cluster blocks. The I/O cost of BIRCH algorithm is linear in the size of the dataset, and a single scan of the dataset is able to yield a good clustering.

In BIRCH algorithm, a cluster is denoted as (C_i, R_i), where C_i is the center and R_i is the radius of cluster *i*. An entity belongs to a cluster when the distance between the entity and the center of the cluster is less than the radius of the cluster. In our algorithm, a distance threshold D_{thd} is selected for all the clusters. Therefore, we represent the clusters in our algorithm as ($KeyBlock_i$, D_{thd}).

The BIRCH algorithm reads the blocks from the *RSet* sequentially, puts the first block as the first cluster and processes the following block, namely *B*, based on the rules:

- Compute the distance between block *B* and each of the existing clusters. Let *i* be the cluster index such that the distance between *B* and $KeyBlock_i$ is the smallest.

- Compute the value of the new radius $R_i^{'}$ of the *i*th cluster under the assumption that *B* is inserted into it. If $R_i^{'} \leq D_{thd}$, we assign *B* to the *i*th cluster by updating its center and setting its radius to $R_i^{'}$. If $R_i^{'} > D_{thd}$, we create a new cluster containing only the Block *B*.

The height-balanced tree used in BIRCH algorithm is similar to an R-tree and hence the closest cluster of a given block can be quickly identified.

When a new block is inserted into a cluster, the center and radius of the cluster must be updated. A naïve method for the computation of the new *KeyBlock* and the radius of cluster *i* could be:

$$KeyBlock_i^{'} = \frac{\sum_j Block_j}{T_i} = \frac{(\sum_j a_j^1, \sum_j a_j^2, ..., \sum_j a_j^M)}{T_i} \qquad \textbf{(4.4)}$$

$$R_i^{'} = \max_j \left(\left\| Block_j - KeyBlock_i^{'} \right\| \Big/ M \right) \tag{4.5}$$

where $Block_j$ is the jth block in cluster i and T_i is the total number of blocks included in cluster i.

Obviously, this method is very computationally expensive. All the existing blocks of a cluster have to be accessed repeatedly during block insertion. To reduce the cost of computation, an approximate method is developed for the radius estimation.

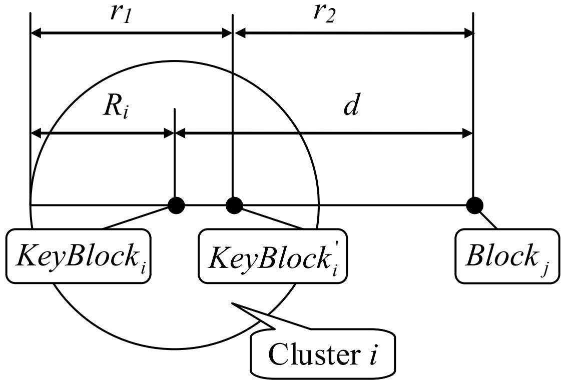


Figure 4.5 Estimation of the center and radius of a cluster for a trial insertion of a block

As illustrated in Figure 4.5, an existing cluster i has (T_i – 1) blocks and is centered at $KeyBlock_i$ with radius R_i. To include a new block $Block_j$, the new center $KeyBlock_i^{'}$ of the cluster is computed as:

$$KeyBlock_i^{'} = \frac{(T_i - 1)KeyBlock_i + Block_j}{T_i} \tag{4.6}$$

and the new radius $R_i^{'}$ is estimated based on the following formulae:

$$\begin{aligned} r_1 &= R_i + \frac{d}{T_i} \\ r_2 &= \frac{d \cdot (T_i - 1)}{T_i} \\ R_i^{'} &= \max(r_1, r_2) \end{aligned} \tag{4.7}$$

where *d* is the distance between the trial block and the center of the cluster. r_1 and r_2 are the two candidate radii, and the greater one will be chosen as the radius of the updated cluster. The mathematical model of it mimics the linear interpolation between two weighted points in the *M*-dimensional space. If a block is inserted into a cluster, the new center of the cluster will be pulled towards the inserted block, and the displacement is inversely proportional to the weights of the two *M*-dimensional points, which are the number of blocks represented respectively.

It is easy to prove that Equations 4.4 and 4.6 produce the same results, so there is no error introduced in the computation of the *KeyBlock*. However, Equation 4.7 tends to produce a value greater than that of Equation 4.5, i.e., the radius could be over estimated. The cluster is actually denser than that implied by the estimated radius. Therefore, this method is effective in producing clusters strictly under the pre-defined error-tolerance. It is also computation efficient as it avoids accessing blocks that are already in the cluster. Furthermore, in contrast to Equation 4.5, the computation of the radius in Equation 4.7 is independent of the *KeyBlock*. Thus, only the radius is evaluated in trying to insert a block into a cluster, and the *KeyBlock* is updated only when the radius satisfies the cluster criterion. This implementation significantly improves the performance of the clustering process.

To exploit the spatial coherence of a time-varying volume dataset, the variance is used to identify those homogeneous *KeyBlock*s that have very small changes in intensity. The mean and the variance of a *KeyBlock* are computed as:

$$mean = \frac{\sum_i a_i}{M} \tag{4.8}$$

$$var = \frac{\sum_i (a_i - mean)^2}{M} \tag{4.9}$$

where a_i is the intensity of the voxel i and M is the number of voxels contained in the *KeyBlock*.

If the variance of a *KeyBlock* is less than a predefined tolerance τ, then it is regarded as a block with homogeneous contents. This *KeyBlock* will be represented only by the *mean* of all the voxels, and it in turn will benefit the performance of the algorithm by reducing the I/O requirement.

```
RSet: the set of blocks.  It initially includes all the blocks.
CSet: the set of clusters.  It is initially empty.

void Clustering()
{
      While (RSet != Empty)
      {
            select Block∈Rset;

            if (CSet != Empty)
            {
                  // Find the cluster with minimum distance to Block
                  Cluster∈CSet, min(distance(Block, Cluster));

                  // Try to insert the block into the cluster
                  Compute radius $R_i$' based on Equation 4.7;

                  // Judge if the insertion is appropriate
                  if ($R_i$' <= $D_{thd}$)
                  {
                        // Yes, update this cluster
                        Cluster.radius = Ri';
                        Compute Cluster.KeyBlock based on Equation 4.6;
                        Cluster.size++;
                        continue;
                  }
            }

            // Create a new cluster and add to cluster set
            create Cluster = {Block};
            CSet = CSet + Cluster;
            RSet = RSet - Block;
      }

      // Identify the property of spatial coherence
      for (every Cluster in CSet)
      {
            Compute Cluster.KeyBlock.mean;
            Compute Cluster.KeyBlock.variance;
      }
}
```

Figure 4.6 Clustering algorithm

In the above description, for simplicity, we assume each volume has *n* voxels identically in each dimension, and each block divided from volumes has *m* voxels identically in each dimension as well. In practice, the number of voxels contained in each dimension of a volume is usually different. To better fit the size of a volume, a block can also be divided to contain the different number of voxels in each dimension accordingly. Such division will not affect the algorithm.

Figure 4.6 gives the pseudo-code of the clustering algorithm.

4.3.3 Data Output

At the last step of the encoding procedure, a moving volume data file is generated and distributed to users. The file will be directly fed into MVD-based time-varying volume renderers. The file structure/format used to save the moving volume data is, therefore, critical for efficient 4D volume rendering. First, in an MVD file, it is crucial that volume data can be read sequentially over the change of time so as to avoid loading the data of all times, and the expired data can be duly released to avoid memory overhead. Second, to reduce the I/O bandwidth, only the sub-volumes with significant contributions (*KeyBlock*s) are saved in the file. An effective data structure registering the relationship of sub-volumes is then important for fast reconstruction of volumes at each time step. Finally, it is also necessary to include other related information of the time-varying volume dataset in an MVD file.

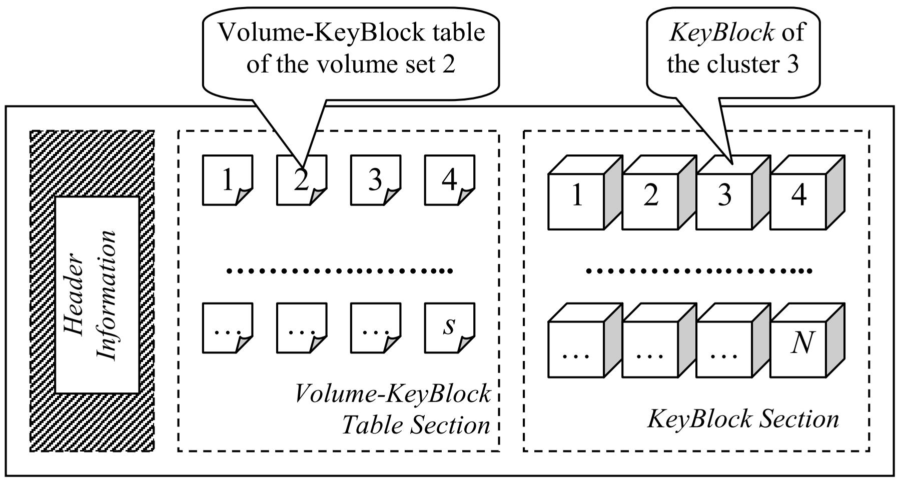

Volume-KeyBlock table of the volume set 2
KeyBlock of the cluster 3
Header Information
1
2
3
4
…
s
1
2
3
4
…
N
Volume-KeyBlock Table Section
KeyBlock Section

Figure 4.7 Structure of an MVD file

With these requirements in mind, an MVD file structure is proposed as illustrated in Figure 4.7. There are three logical sections in an MVD file, namely the header information section, the Volume-KeyBlock table section and the *KeyBlock* section.

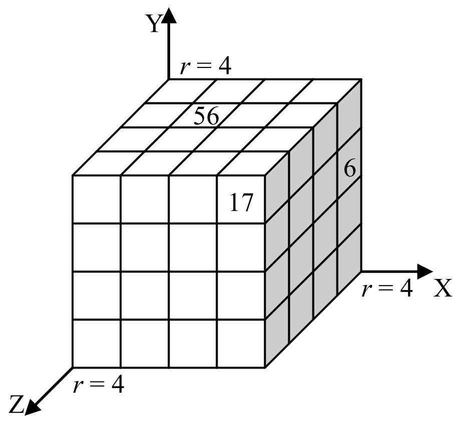


Figure 4.8 Graphical representation of a Volume-KeyBlock table

An MVD file begins with a header information section that identifies the file with a signature followed by the information such as the resolution of the individual volume, the number of volumes, voxel format, data descriptions and pointers to the other two sections in the file etc. of the time-varying volume.

The Volume-KeyBlock table section consists of Volume-KeyBlock tables, a collection of lookup tables corresponding to the 3D volumes, one table for each time step. As each volume is decomposed into blocks and each block is organized in one cluster represented by a *KeyBlock*, the lookup table is thus used to reconstruct the volume of each time step from the *KeyBlock*s. It can be treated as a 3D array with size r^3. The corresponding graphical representation of the table below is shown in Figure 4.8.

Table 4.1 A Volume-KeyBlock table

Block Location (*x*, *y*, *z*)	***KeyBlock* Index**
(0, 0, 0)	…
(1, 0, 0)	…
…	…
(3, 2, 0)	6
…	…
(1, 3, 1)	56
…	…
(3, 3, 3)	17

An entry of a block in the lookup table indicates the index number of the cluster, which the block belongs to. This number is also the index of the corresponding *KeyBlock* in the *KeyBlock* section. During rendering, the block of a volume could be reconstructed by using this *KeyBlock*. Table 4.1 demonstrates the structure of a Volume-KeyBlock table with a few example entries.

In the *KeyBlock* section, all the *KeyBlock*s generated from the clustering step are labeled with index numbers and saved in the MVD file based on the following rules:

- The *KeyBlock*s generated from the blocks of earlier volumes are given smaller index numbers.
- The *KeyBlock*s are stored lexicographically according to the x, y and z dimensions.
- The *KeyBlock*s and the clusters they represented adopt the same indices.

For efficient memory management, each *KeyBlock* is associated with a *Last Volume Number* (LVN), which is the number of the last volume that contains blocks belonging to the cluster represented by the *KeyBlock*. The LVN indicates the life period during which a *KeyBlock* will be used to reconstruct volume(s) from time to time and should reside in the memory, and it is also the expiring time after which a *KeyBlock* should be released. The *KeyBlock*s, therefore, are not released one by one as the order they are loaded in. A dynamic memory management scheme should be employed during the implementation.

In this way, *KeyBlock*s are stored so that they can be properly loaded and released as the sequence of volume being processed.

4.3.4 Additional Processing

Encoding of an MVD could be varied to cater for different renderers. For example, if a shear-warp algorithm is used as the renderer engine, then *KeyBlock*s need to be RLE-encoded. For a linear-level-octree (LLO) renderer [Wang et al. 2002b], *KeyBlock*s are encoded in LLOs.

Just as traditional 3D volume rendering, operations of image-processing such as noise filtering and segmentation [Hua et al. 2000] normally can improve the performance of rendering. They can be applied before the encoding without affecting the structure of the algorithm.

Classification can be applied at either encoding stage (if the transfer function is already available) or rendering stage at run-time. If a tri-linear interpolation scheme (e.g., ray-casting) is used, the adjacent blocks need to have overlapping boundaries at the time of encoding.

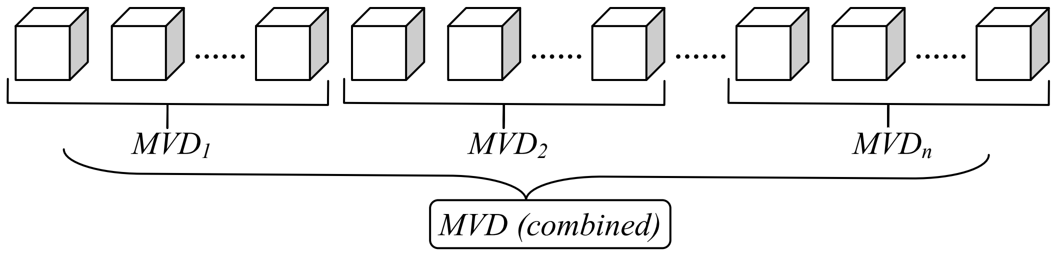


Figure 4.9 The scheme of encoding time-varying volume dataset with many time steps

For time-varying dataset with many time steps, it is better to divide them into multiple groups in time order, apply the encoding algorithm to each group and combine the results into a single MVD file (Figure 4.9). This will ease the search of a certain time step in the whole MVD file or the reconstruction of an intermediate time step from the beginning and it also avoids the superimposition of too many artifacts.

4.4 Rendering – the Decoding Process

In the rendering stage, an MVD is rendered in a decoding process. In this section, I will focus on the rendering algorithm designed to work on an MVD file, and will not go into details of the underlying volume rendering algorithms. Various existing volume rendering techniques could be used in this algorithm directly or after an optimization. Interested reader can refer to [Elvins 1992] and [Meissner et al. 2000], which give detailed introduction and evaluation of a variety of popular volume rendering algorithms.

4.4.1 MVD Rendering Algorithm

As illustrated in the 4D volume rendering framework (Figure 4.1), under the supervision of a specialist, a time-varying volume dataset is converted into an MVD file. It is then distributed to users through various storage media or networks and viewed through client renderers. The renderer decodes the MVD, reconstructs the volume at every time step and updates the output images to users over the change of time steps. An iterative algorithm is thus employed in the renderer engine as below.

Let us denote the working volume as q. Initially, the volume of the first time step is used as the working volume ($q = 1$) and the following steps are executed:

1. *KeyBlock*s whose LVNs are less than *q* are released together with their associated partial-image buffers. The final image of the current time step is initialized.

2. *KeyBlock*s are read from the MVD file in turn. Each *KeyBlock* is associated with a partial-image buffer, and *KeyImage*, the rendering result of each *KeyBlock*, is saved into the partial-image buffer. After all the *KeyBlock*s contained in volume *q* are loaded, they are rendered according to the three rules:

    - If it is the first volume, all the *KeyBlock*s are rendered.

    - If the current model-view transformation or transfer functions are changed as compared to that in the previous time step, then all *KeyBlock*s are re-rendered.

    - If the current model-view transformation and transfer functions have not been changed, only *KeyBlock*s that are newly loaded are rendered.

3. The *KeyImage*s of the *KeyBlock*s are composited in 2D space according to the Volume-KeyBlock table of volume *q* and the final image is constructed as follows:

    - According to the current viewing direction, blocks in volume *q* are accessed in front-to-back order. With the Volume-KeyBlock table, *KeyBlock*s can be easily located.

- The *KeyImage* of the *KeyBlock* is composited into the final image at the corresponding projection area.

After all blocks of volume q are processed, the final image is produced and displayed to users.

4. To proceed to the volume of the next time step, q is increased by one ($q = q + 1$).

The above steps are repeated until the whole sequence is processed.

In the above algorithm, once the *KeyImage*s are produced, the final image is generated by compositing their colors and opacities in front-to-back order based on the theory of partial ray compositing. The final image can be composed from the *KeyImage*s by using the *over* operator as in the following equation:

$$\boldsymbol{I_q = KeyImage_1 \ \ over \ \ KeyImage_2 \ \ over \ \ldots} \qquad \textbf{(4.10)}$$

where I_q is the final image of volume q.

2D re-sampling of the *KeyImage*s may be required if the sampling rate of the *KeyImage* is different from that of the final image or when they are not sampled along the same set of rays. In any case, the early-ray-termination is still possible for both *KeyBlock* rendering and *KeyImage* composition, where samples in *KeyImage*s can be safely skipped when pixels of the final image are already opaque enough.

4.4.2 Underlying Volume Renderers

In the cluster-based time-varying volume rendering algorithm proposed in this chapter, users are allowed to select the underlying 3D renderers for *KeyBlock* rendering. Employment of different 3D volume rendering algorithms as the kernel of the renderer engine does not affect the essence of the algorithm. Most 3D rendering algorithms such as ray-casting, shear-warp, splatting and texture-mapping algorithms can be used. In the following, I summarize the advantages and disadvantages of the different renderers if they are used in the algorithm.

A ray-casting renderer can fit well in this algorithm. It is hardware independent. Extensive shading is supported. However, such a renderer could slow down the overall performance of rendering. The shear-warp algorithm is relatively complicated as compared with the ray-casting algorithm. It is fast and independent of the hardware. Extensive shading is also supported.

Splatting is another good choice. It is hardware-independent and extensive shading is supported. Its rendering speed is slower than the shear-warp method. However, this method will potentially reduce the rendering artifacts resulted by *KeyBlock* boundaries. The inherent characteristic of splatting algorithm makes the boundary of the objects (*KeyBlock*s) blurred in the rendered image (although this can be regarded as another kind of artifacts.).

Since the GPU-based computer graphics methods are becoming popular these years, it is necessary for us to discuss about texture-based volume rendering. A texture-mapping algorithm is dependent on the graphics hardware. Nowadays, 2D texture-mapping is supported by most graphics cards. But software sampling may be required to create the texture images for the three major orientations. 3D texture-mapping is capable of performing the fast 3D interpolation through hardware so that software sampling can be avoided. If 3D texture-mapping is supported by the graphics card, it can be employed by using the OpenGL and its extensions[11]. When texture-mapping method is used in this 4D rendering algorithm, partial-image rendering is no longer necessary and the algorithm is slightly varied.

Texture-based rendering can be achieved at two levels, namely block level and volume level. At the block level, each of the blocks is texture-mapped and transferred to a proper location. The volume rendering of a time step is completed after all the blocks of the volume are processed. Since the blocks are represented by the *KeyBlock*s, the texture memory can be reused for homogeneous blocks and the requirement of texture memory is reduced. However, if the size of blocks is too small, there will be too many agent polygons used for texture-mapping, and it will affect the rendering performance. At the volume level, the volume of each time step is first reconstructed from the *KeyBlock*s and entirely downloaded to the texture memory in the graphics hardware. The volume is then texture-mapped and rendered on the screen. The implementation of this method is normally simpler than that of the block level method. For this cluster-based algorithm, the data throughput between time steps is low and hence textures can be updated very quickly. However, the texture memory must be large enough to, at least, fit an entire set of a 3D volume.

[11] On Windows® platform, the OpenGL only supports to version 1.1, in which the 3D texture-mapping is not yet included until version 1.2. Therefore, the 3D texture-mapping can only be accessed through the OpenGL extension, if the graphics card supports it. *http://www.opengl.org/resources/features/OGLextensions/*

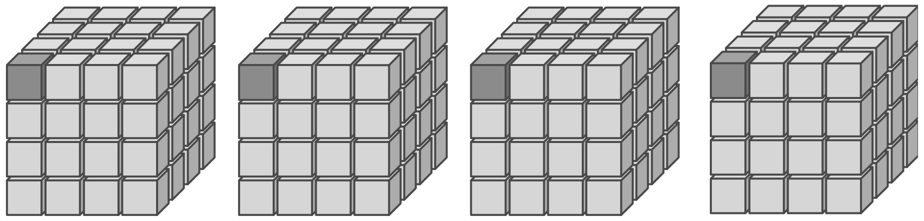
4.5 Global Coherence

The rendering algorithm in this chapter identified and exploited a new kind of coherence, *global coherence*, of the time-varying volume data, which is not reported or utilized by previous 4D volume rendering methods. We observed that 3D volume datasets may have similar regions at different locations (e.g., blood vessels) and that sub-volumes may change repeatedly from one time step to another (e.g., heart beating and contrast injection). The global coherence of time-varying volume data refers to the fact that sub-volumes from arbitrary locations in space or time may contain similar voxels. The similar regions can be grouped together and represented once only, resulting in savings in both space and rendering time.

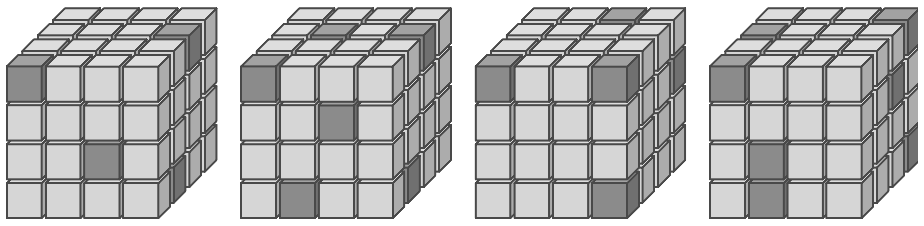
The spatial coherence of 3D volume data, which refers to the fact that voxels in adjacent regions tend to have similar values, can be effectively identified while exploiting the 4D volume global coherence. In particular, the empty regions can be grouped into one cluster and represented by a blank *KeyBlock*. Thus, volume blocks represented by the blank *KeyBlock* will be simply skipped during rendering.

The temporal coherence is exploited based on the observation that voxels will not change drastically from one time step to the next. Therefore, regions from the same 3D location but different time steps may be regarded similar. Apparently, the temporal coherence is subsumed by the global coherence. With the employment of clustering technique, volume regions that are grouped into a cluster may come from any portions of a time-varying volume dataset in both space and time directions. Similar regions can be represented by the same

KeyBlock and rendered only once if rendering conditions are unaltered. The savings in the rendering time, therefore, could be significant as compared to the 4D volume rendering algorithms exploiting only temporal coherence. In Figure 4.10, a group of similar blocks is identified by exploiting two types of data coherence and labeled in dark grey. Obviously, more savings are possible if global coherence is exploited in rendering.



(a) Temporal coherence used



(b) Global coherence used

Figure 4.10 Comparison of temporal coherence and global coherence

In the proposed algorithm, each cluster is represented as a single block (*KeyBlock*). When global coherence is used, the total number of different blocks used to represent the entire time-varying volume is significantly smaller than the original number of blocks, which results in remarkable savings in space requirement. The savings are meaningful especially for online data distribution and when the I/O bandwidth is critical.

4.6 Results and Discussion

The experiments were done on a desktop PC with an Intel Pentium IV 2.52GHz processor and 1 gigabyte system memory equipped with NVIDIA Quadro®4 700 XGL graphics card with 64 megabyte onboard memory. The experiments are designed to study the performance gains of the proposed algorithm in terms of space reduction and speed acceleration. The degradation of the image quality due to the error introduced during the MVD encoding is also quantitatively measured and analyzed. The same set of medical time-varying volume datasets as used in Chapter 3 are employed for testing of the newly proposed algorithm in this chapter. Table 4.2 re-lists the details of these datasets.

Table 4.2 Experimental time-varying volume datasets

Dataset	Dimensions	Resolution (mm^3)	Time Steps	Size (MB)	Modality
HAND	512×512×136	0.39×0.39×4.0	5	171.25	MRA
BREAST	256×256×26	1.25×1.25×4.2	5	8.13	MRI
HEART I	192×156×27	1.67×1.67×8.0	20	15.42	MRI
HEART II	156×192×16	1.77×1.77×8.0	20	9.14	MRI
ABDOMEN	256×256×12	1.02×1.02×5.0	39	29.25	MRU

These time-varying volume datasets were encoded into MVD presentations. It is interesting to study the performance of the proposed algorithm under different encoding error tolerances. The compression ratio is thus employed to measure its performance in space reduction. The compression ratio (R) of an MVD file is defined as

$$R = (1 - \frac{\text{MVD Size}}{\text{Raw Data Size}}) \times 100\% \qquad \textbf{(4.11)}$$

which gives 100% compression if nothing remains after compression and 0% if the size remains unchanged. Note that the cluster-based method is not meant for data compression. We did not employ any compression-specialized algorithms in the proposed method as well.

Parameters used for MVD encoding of the *HAND*, *BREAST*, *HEART I*, *HEART II* and *ABDOMEN* datasets and their compression ratios achieved are shown in Table 4.3 to Table 4.7 respectively.

Table 4.3 MVD encoding of the *HAND* dataset under three different cluster *NED* thresholds

Dataset Name	Block Size	Cluster *NED* Threshold	Time Cost (Seconds)	Size (MB)	Compression Ratio
Hand A	16×16×17	0.10	3953	36.27	78.7%
Hand B	16×16×17	0.15	3044	26.29	84.5%
Hand C	16×16×17	0.20	2525	20.94	87.7%

Table 4.4 MVD encoding of the *BREAST* dataset under three different cluster *NED* thresholds

Dataset Name	Block Size	Cluster *NED* Threshold	Time Cost (Seconds)	Size (MB)	Compression Ratio
Breast A	16×16×13	0.10	13	3.40	58.2%
Breast B	16×16×13	0.15	12	2.96	63.5%
Breast C	16×16×13	0.20	10	2.34	71.2%

Table 4.5 MVD encoding of the *HEART I* dataset under three different cluster *NED* thresholds

Dataset Name	Block Size	Cluster *NED* Threshold	Time Cost (Seconds)	Size (MB)	Compression Ratio
Heart I A	12×13×27	0.10	22	3.53	77.1%
Heart I B	12×13×27	0.15	16	2.27	85.3%
Heart I C	12×13×27	0.20	13	1.61	89.6%

Table 4.6 MVD encoding of the *HEART II* dataset under three different cluster *NED* thresholds

Dataset Name	Block Size	Cluster *NED* Threshold	Time Cost (Seconds)	Size (MB)	Compression Ratio
Heart II A	13×13×16	0.05	21	4.58	49.9%
Heart II B	13×13×16	0.10	13	2.79	69.5%
Heart II C	13×13×16	0.15	10	1.90	79.3%

Table 4.7 MVD encoding of the *ABDOMEN* dataset under three different cluster *NED* thresholds

Dataset Name	Block Size	Cluster *NED* Threshold	Time Cost (Seconds)	Size (MB)	Compression Ratio
Abdomen A	16×16×12	0.10	305	20.76	29.0%
Abdomen B	16×16×12	0.15	264	17.46	40.3%
Abdomen C	16×16×12	0.20	236	14.93	49.0%

With the increase of cluster *NED* threshold, higher compression ratio could be achieved. The maximum compression ratio achievable under a certain error tolerance is mainly dependent on the nature of global coherence in each dataset. We achieved up to 90% compression ratio for the datasets with high global data coherence (e.g., *HAND* and *HEART I*). The compression ratio of the *ABDOMEN* dataset is relative low due to its low data coherence.

Table 4.8 Time cost of MVD encoding of the *HAND* dataset with three different block sizes

Block Size	Cluster *NED* Threshold	Time Cost (Seconds)	Compression Ratio	Absolute Error (E_A)	Cycle Rendering Time (s)
16×16×17	0.15	3381	84.5%	17.938	3.479
32×32×34	0.04	295	79.5%	18.102	4.747
64×64×68	0.01	102	69.4%	15.278	4.582

The time cost of MVD encoding decreases with the increase of cluster *NED* threshold. Since the cluster-based 4D rendering method is designed for offline encoding, it is normally acceptable with relative long conversion time. However, the conversion time is also dependent on the block size. The time can be significantly reduced when large blocks are used. Table 4.8 gives three examples of the MVD encoding of the *HAND* dataset with different block sizes. It shows that the MVD encoding of the *HAND* can also be done within two minutes with the similar rendering quality (E_A).

Therefore, appropriate block size should be used according to the time or space available in practice. When there is an urgent need to view a 4D dataset, larger block size should be used for an instant encoding and rendering, whereas smaller block size should be used to produce an MVD file with smaller file size for further distribution. A relative small cluster *NED* threshold should be used when the block size is large, because the significant differences between voxels could be averaged out when many voxels are involved in the computation of the *NED*.

To show advantages of global coherence over temporal coherence, I investigated rendering workload, which can be quantified by the number of blocks needed to be processed, with and without exploiting two types of data coherence. The respective number of blocks for each dataset is shown in Table 4.9. The same error tolerance of 0.2 is used in exploring both temporal coherence and global coherence. It is clear that the saving achieved with global-coherence is significantly higher than what can be achieved by exploiting the temporal coherence alone for most test datasets.

Table 4.9 Saving due to global coherence as compared with temporal coherence in the number of blocks needed to be processed

	HAND	*BREAST*	*HEART I*	*HEART II*	*ABDOMEN*
G	5023	736	398	433	5086
T	10443	1220	453	442	6896
S	5420	484	55	9	1810
R	52%	40%	12%	2%	26%

G: The number of blocks that must be processed by exploiting global coherence
T: The number of blocks that must be processed by exploiting temporal coherence
S: The number of blocks saved due to global coherence ($S = T - G$)
R: Ratio of saving due to global coherence as compared with temporal coherence ($R = S / T$)

The compressed MVD files result in the reduced I/O throughput during rendering. For a cluster-based renderer, the I/O throughput varies for different time steps. In contrast, for a traditional volume renderer used for 4D rendering, the I/O throughput will be constant for all time steps. In the following, I compared the I/O throughput between MVD files and raw data. The comparison results of the *HAND*, *BREAST*, *HEART I*, *HEART II* and *ABDOMEN* datasets are illustrated in Figure 4.11 to Figure 4.15, respectively.

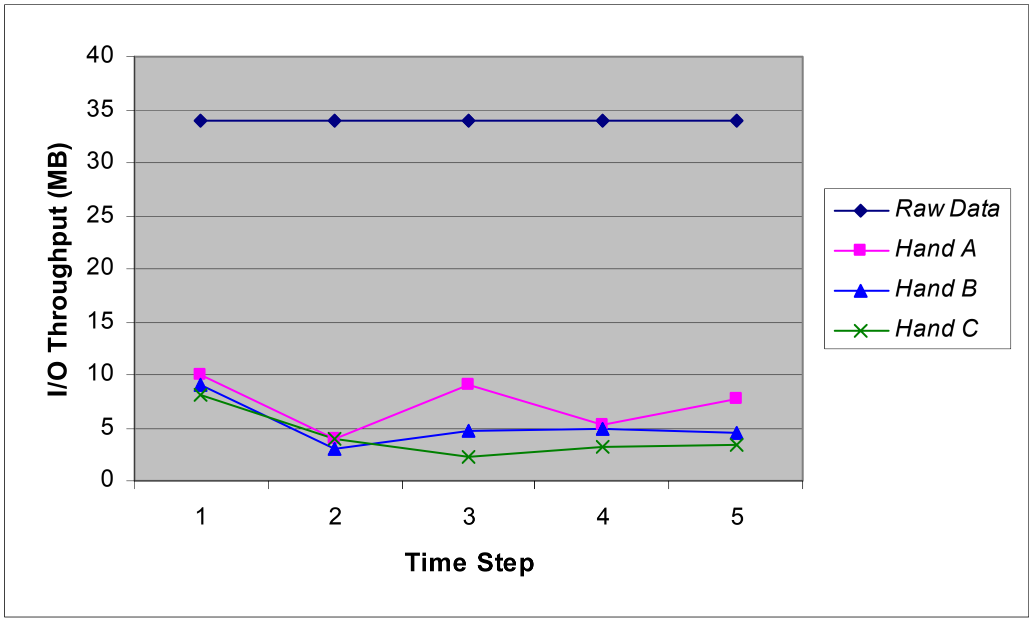


Figure 4.11 Comparison of the I/O throughput between MVD and raw data (*HAND* dataset)

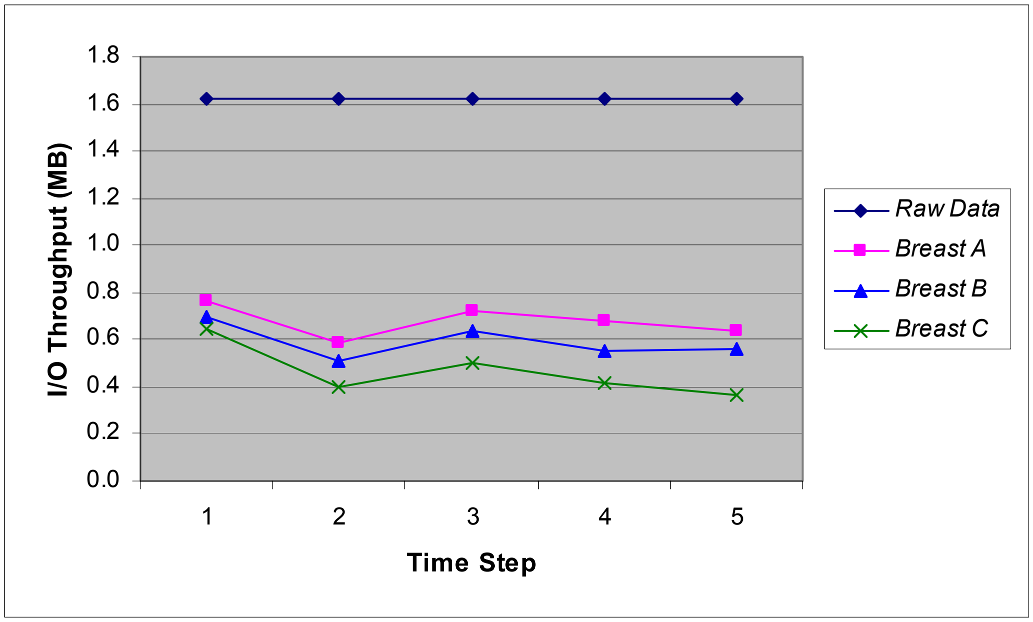


Figure 4.12 Comparison of the I/O throughput between MVD and raw data (*BREAST* dataset)

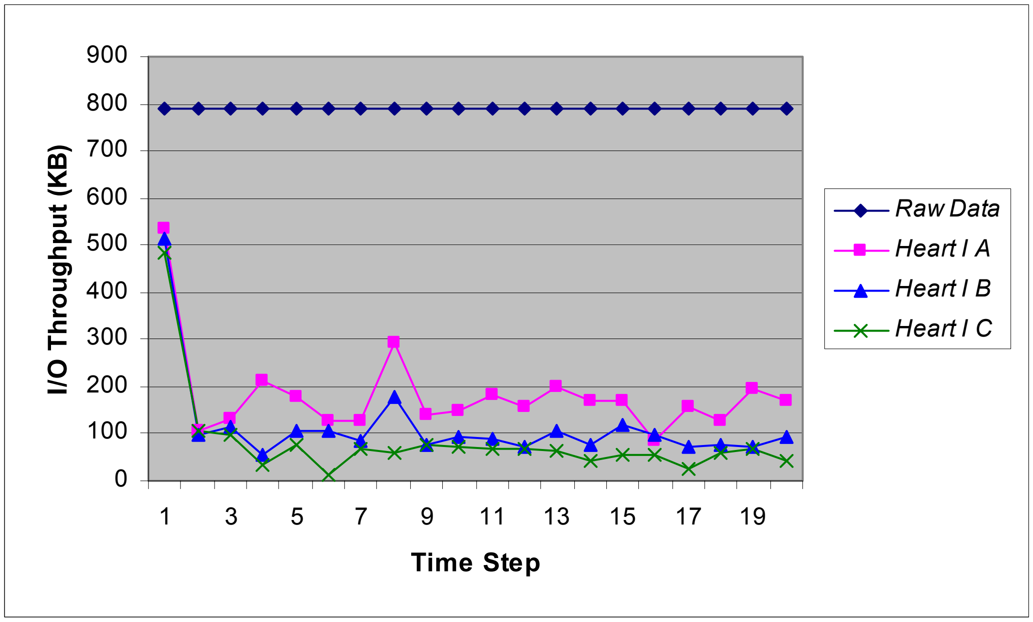


Figure 4.13 Comparison of the I/O throughput between MVD and raw data (*HEART I* dataset)

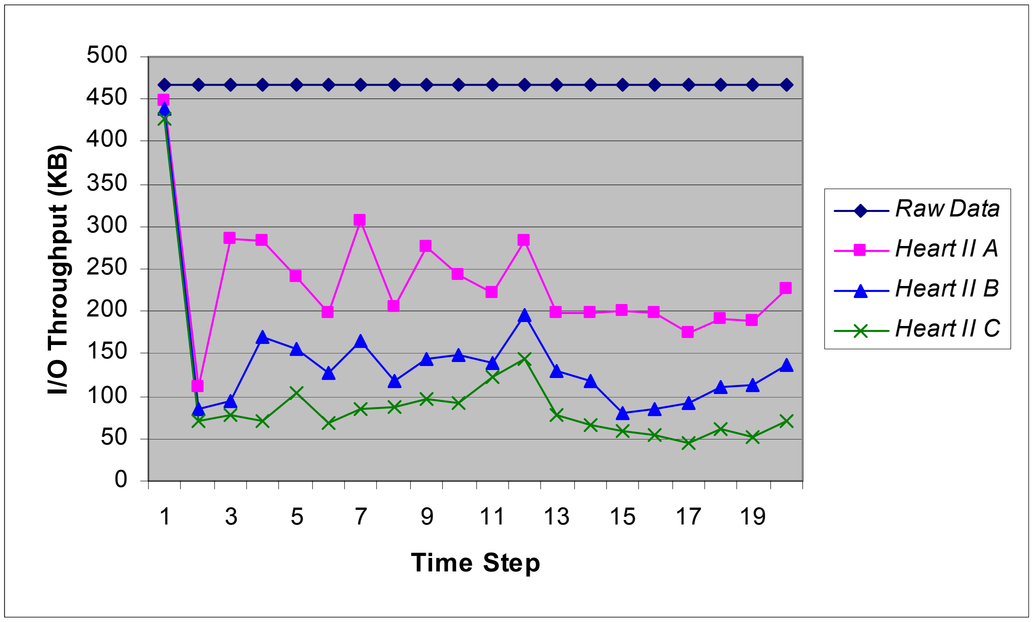


Figure 4.14 Comparison of the I/O throughput between MVD and raw data (*HEART II* dataset)

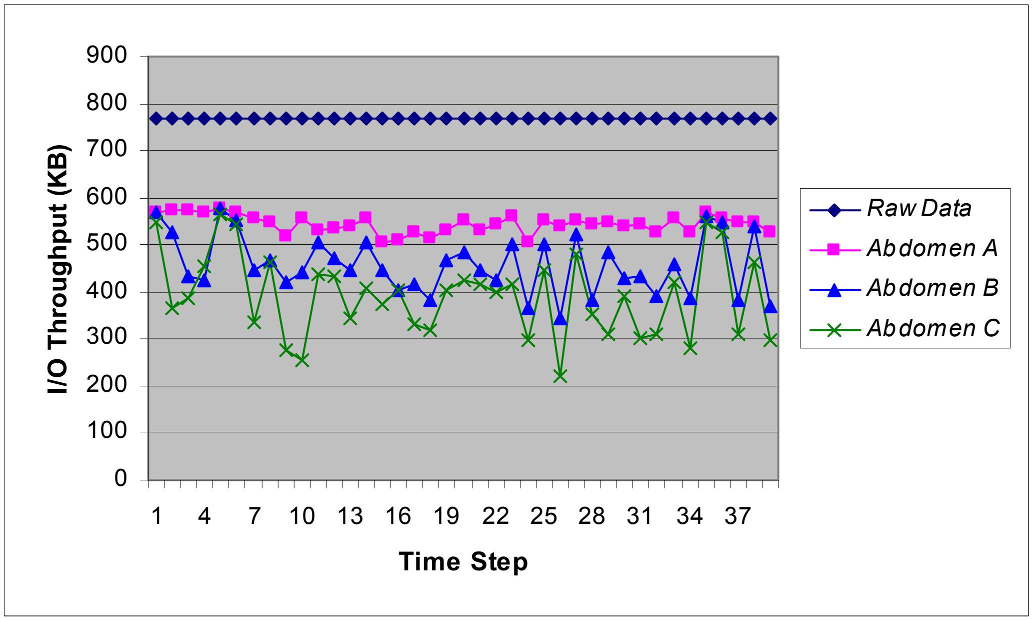


Figure 4.15 Comparison of the I/O throughput between MVD and raw data (*ABDOMEN* dataset)

It is clear that the MVD files have much smaller I/O throughput than the raw data. An MVD normally has larger I/O throughput at the first time step than the subsequent steps as the volume is reconstructed from scratch initially. The volumes of the subsequent time steps are updated by reusing the *KeyBlock*s of the previous steps. When there is high data coherence between steps, the I/O throughput is low. Next, we will investigate how the reduced I/O throughput affects the rendering performance.

The proposed cluster-based time-varying volume rendering algorithm is implemented with three underlying renderers, namely ray-caster, 2D texture-mapper and 3D texture-mapper. Due to the inherent characteristics of the ray-casting algorithm, it could be slow for us to discern the performance gains of the proposed method, although this implementation also achieves satisfactory results as compared with the regular ray-casting algorithm. The capability of supporting texture-based rendering is an important feature of current volume rendering algorithms. The texture-based implementation of this algorithm enables us to compare its performance with hardware accelerated rendering that is regarded as the fastest method commonly achievable on personal computers. Therefore, in the following, I am constrained to compare the rendering performance of the 2D/3D texture-based implementation of the cluster-based rendering method with the regular 2D/3D texture-mapped rendering method. We are interested to analyze their relative performance. The cluster-based renderer and the regular texture-mapping renderer are implemented based on the same piece of codes. Any optimization of the implementation will improve the performance of both renderers.

The experiment is conducted based on the following procedures. After a dataset is loaded into the system, it is rendered repeatedly for 20 cycles while rendering timing of each time step is recorded. The timing results of the last 10 cycles are then averaged and reported as the performance results of this dataset. The design of the experiment ensures the timing results obtained have become stable and renderers can benefit from the I/O cache if possible. The rendering speeds are measured for 2D and 3D texture-based implementation separately. In the following, performance results are illustrated in Figure 4.16 to Figure 4.21, where the regular texture-mapped rendering of the raw data is denoted as *Regular TM* and cluster-based rendering of MVD datasets is denoted with the *dataset name*s instead.

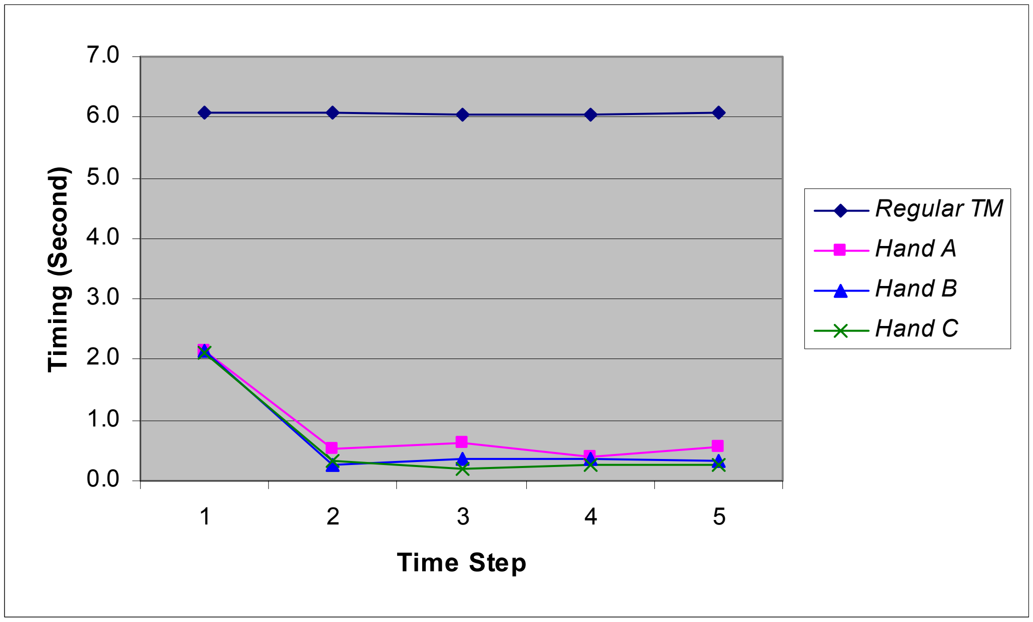


Figure 4.16 Speed comparison between regular texture-mapped rendering and cluster-based rendering of the *HAND* dataset using 2D texture-mapping

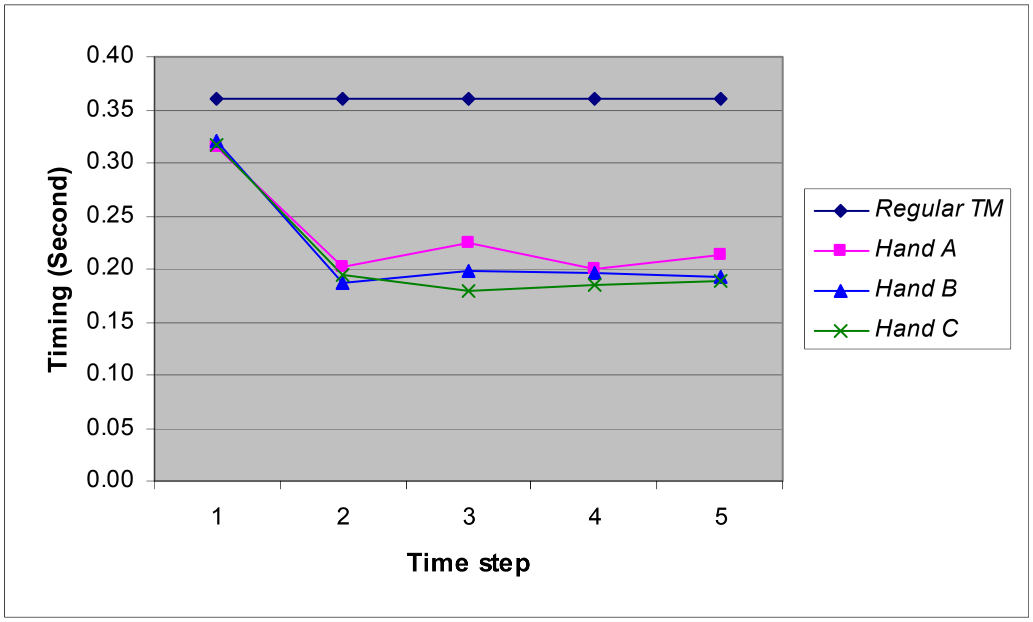


Figure 4.17 Speed comparison between regular texture-mapped rendering and cluster-based rendering of the *HAND* dataset using 3D texture-mapping

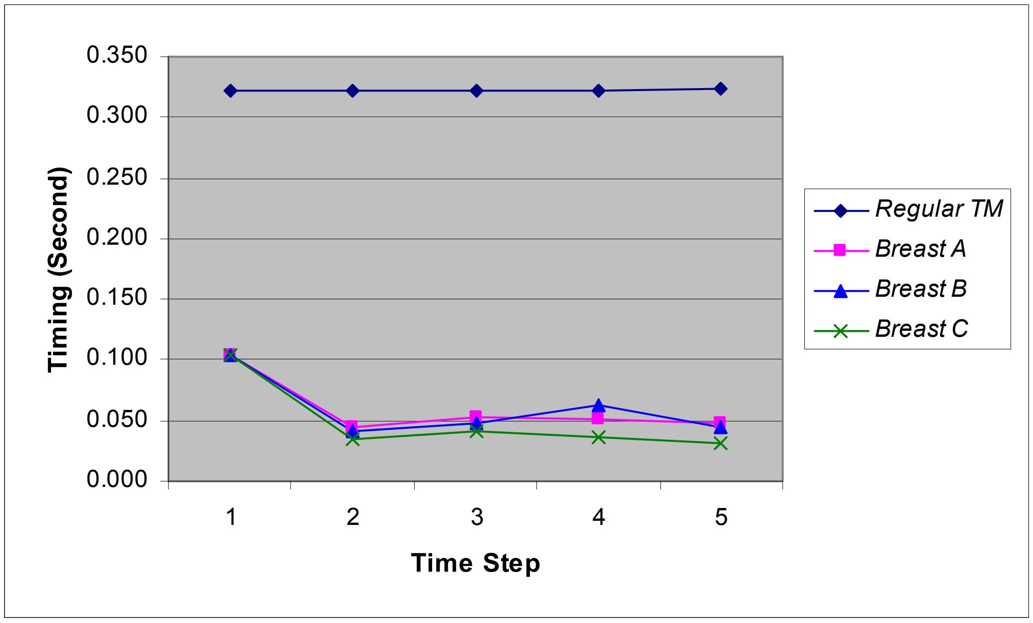


Figure 4.18 Speed comparison between regular texture-mapped rendering and cluster-based rendering of the *BREAST* dataset using 2D texture-mapping

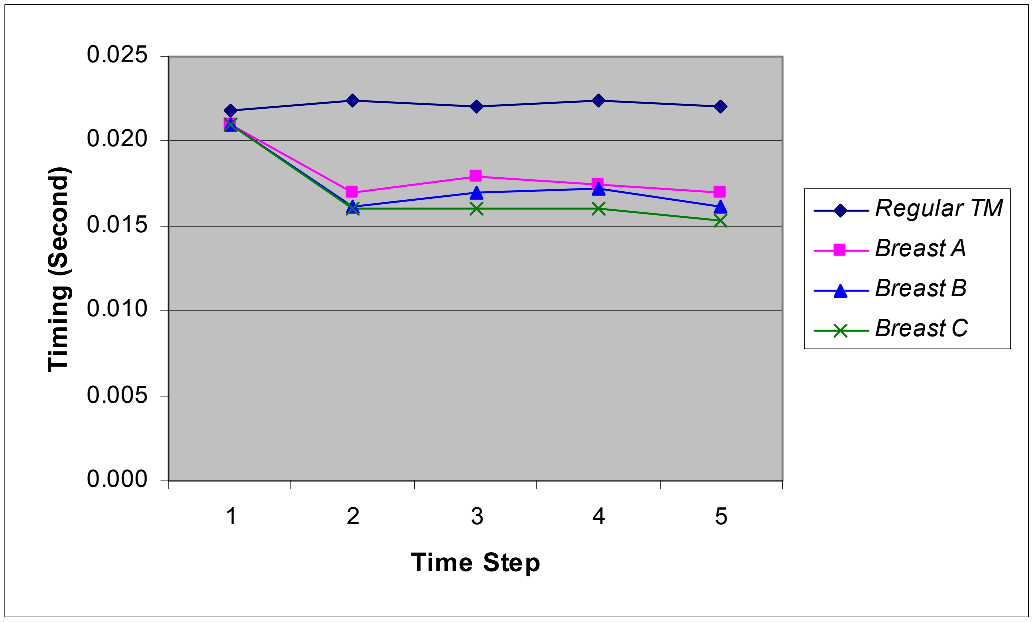


Figure 4.19 Speed comparison between regular texture-mapped rendering and cluster-based rendering of the *BREAST* dataset using 3D texture-mapping

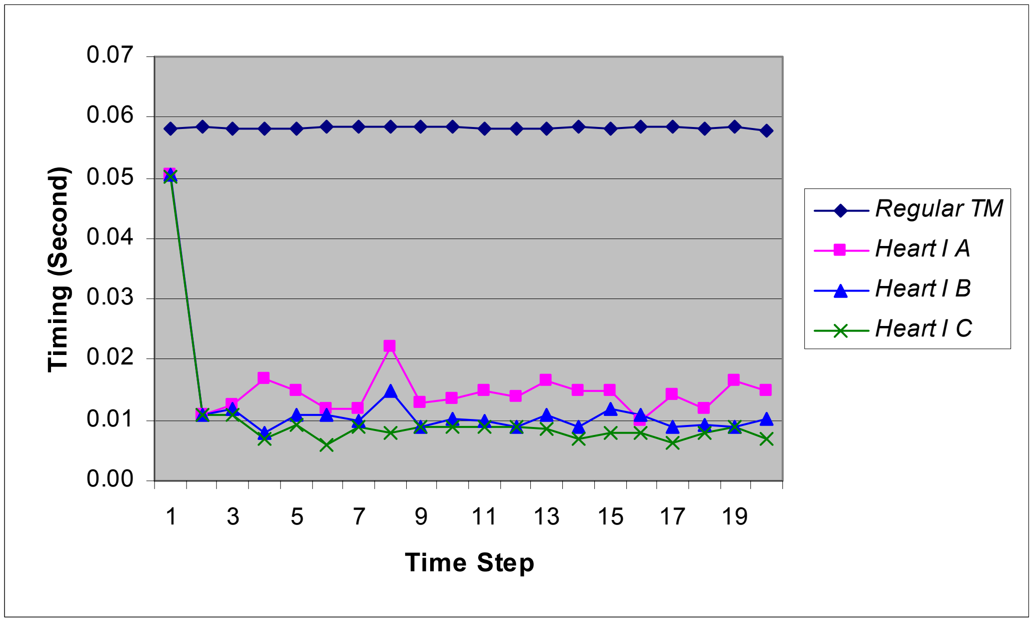


Figure 4.20 Speed comparison between regular texture-mapped rendering and cluster-based rendering of the *HEART I* dataset using 2D texture-mapping

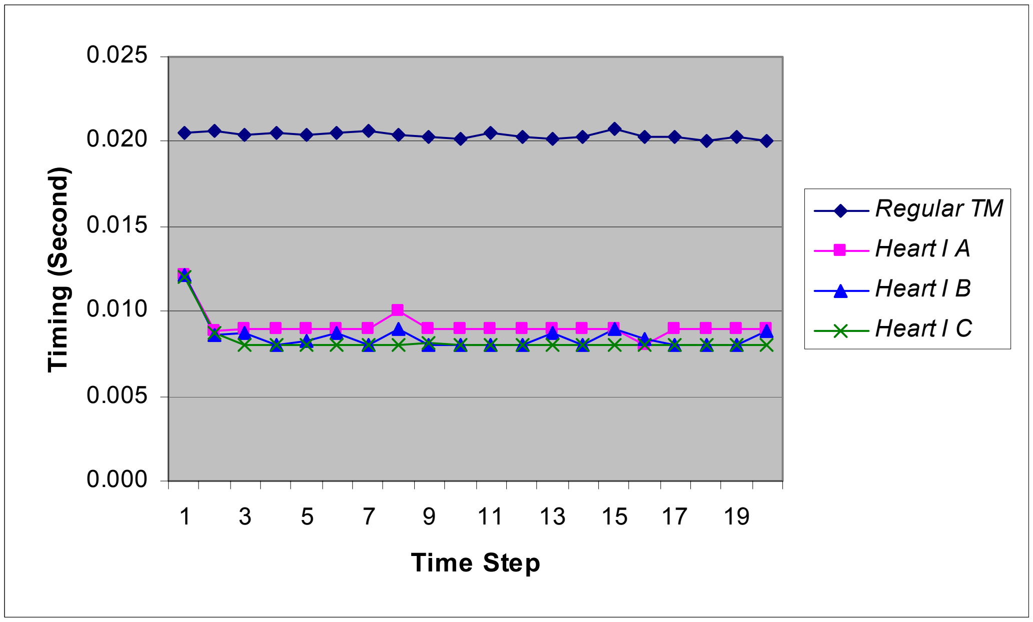


Figure 4.21 Speed comparison between regular texture-mapped rendering and cluster-based rendering of the *HEART I* dataset using 3D texture-mapping

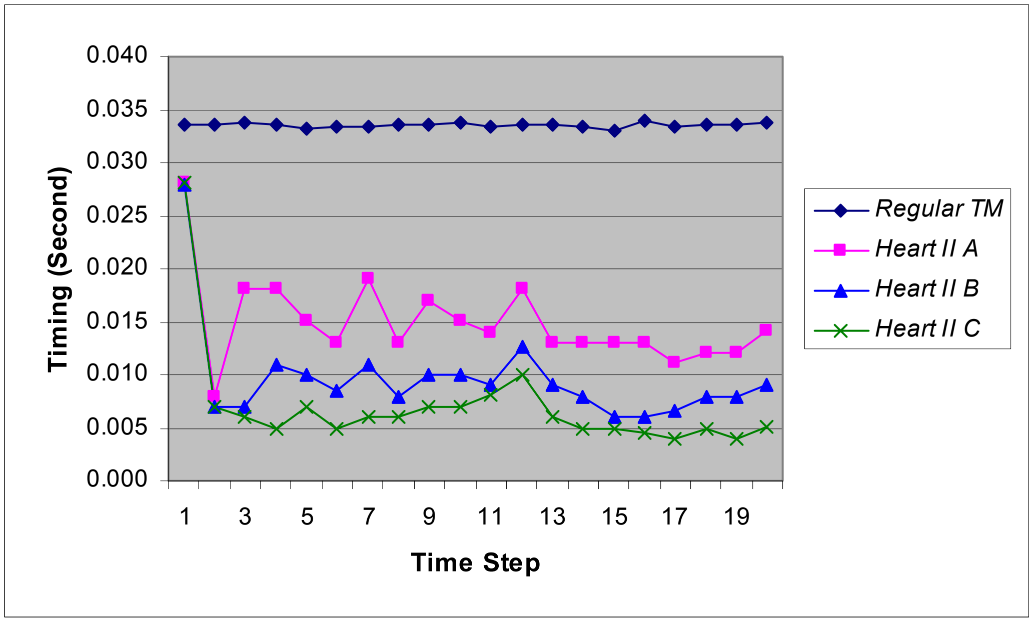


Figure 4.22 Speed comparison between regular texture-mapped rendering and cluster-based rendering of the *HEART II* dataset using 2D texture-mapping

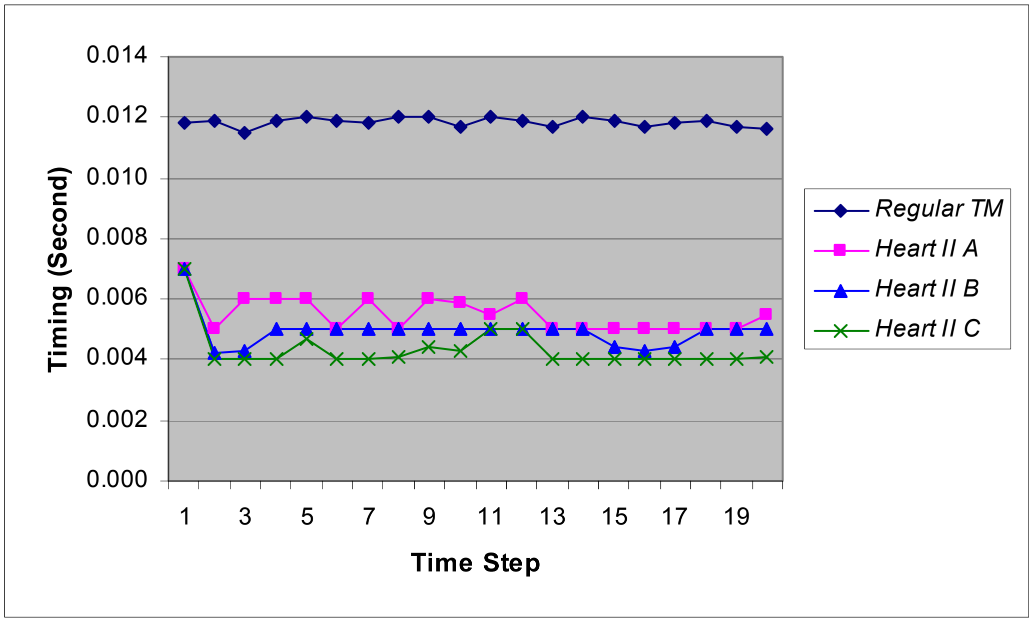


Figure 4.23 Speed comparison between regular texture-mapped rendering and cluster-based rendering of the *HEART II* dataset using 3D texture-mapping

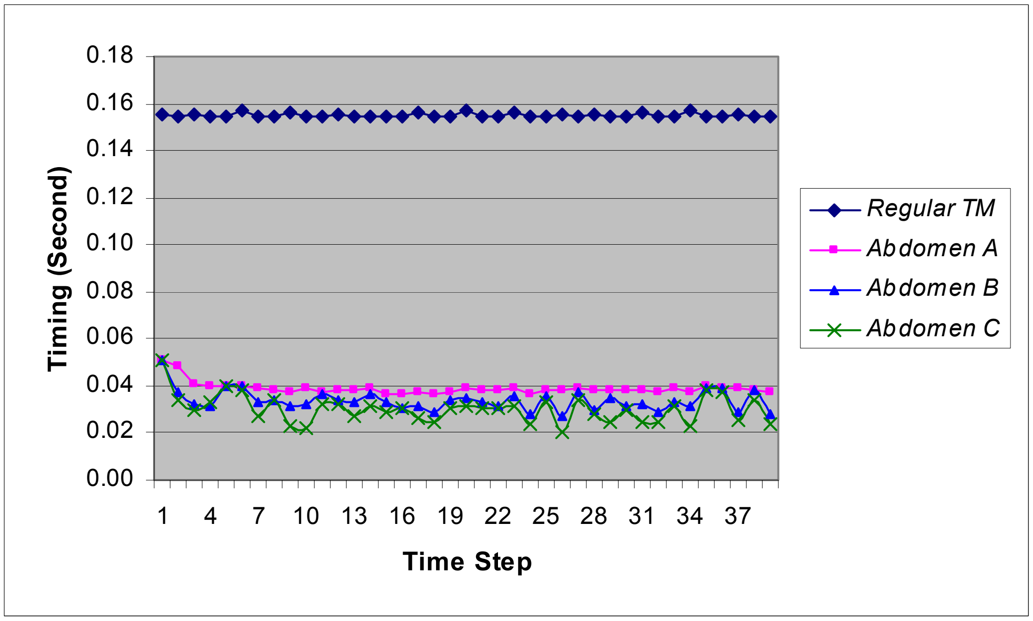


Figure 4.24 Speed comparison between regular texture-mapped rendering and cluster-based rendering of the *ABDOMEN* dataset using 2D texture-mapping

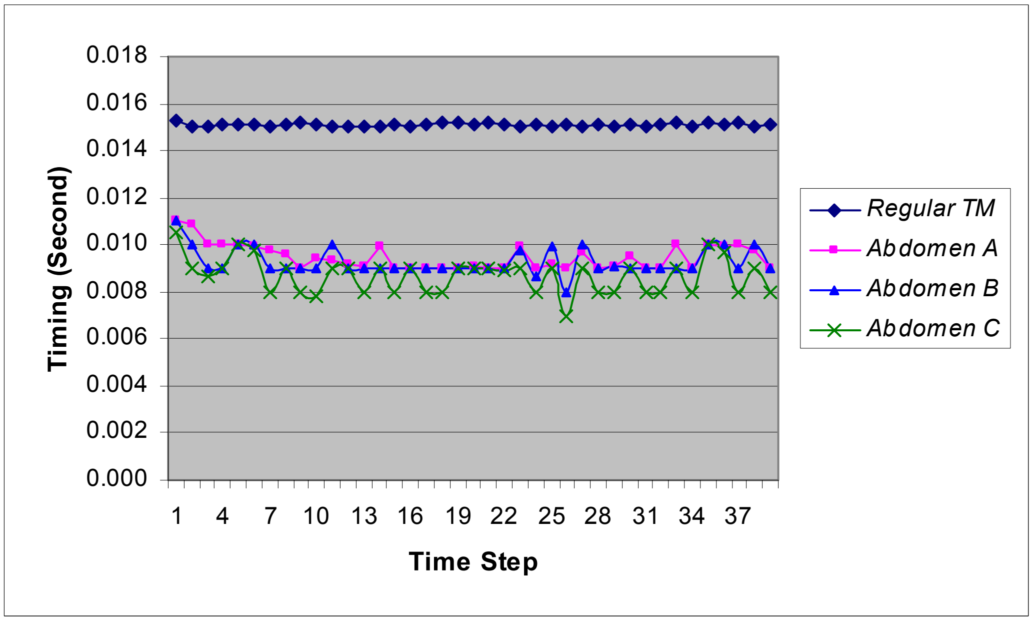


Figure 4.25 Speed comparison between regular texture-mapped rendering and cluster-based rendering of the *ABDOMEN* dataset using 3D texture-mapping

We observed that the rendering time was directly proportional to the I/O throughput of each dataset. This is based on the fact that it takes most of the time for a cluster-based renderer to process new I/O inputs (*KeyBlock*s) in every time step. For the texture-based implementation of the renderers, the reduced I/O throughput improves the rendering performance mainly in three aspects. First, it reduces the file loading time from a hard disk to the system memory. Second, it reduces the texture downloading time from the system memory to the texture memory in the graphics card. The speed of these two processes is hardware dependent. The reduced I/O throughput however, will improve the relative performance anyway. Last, for 2D texture-based rendering, the volume has to be reconstructed for the projection other than the axial direction, so the reduced I/O throughput reduces the volume data that need to be rebuilt so as to accelerate the overall rendering speed.

It is observed that the regular texture-mapping renderer is heavily dependent on the capacity of system memory and cache. In the initial cycle of rendering, the entire set of raw volume data is loaded into the system for the first time. This process could take long time at every time step. Thereafter, raw data are cached and the rendering speed increases. In contrast, the rendering speed of cluster-based rendering is more consistent and there is little difference in the initial and the following rendering cycles. Cluster-based rendering therefore is superior over the traditional methods when there is no demand for repeated rendering at a viewing session.

It is interested to quantitatively study the speedup of the cluster-based renderer over the regular renderer under different encoding error tolerances. The speedup was calculated based on the cycle rendering time of two renderers. Similar to the step timing acquired previously, the cycle timing was based on an average of 10 cycles of rendering time of each dataset. As defined in Chapter 3, two types of factors, i.e., normal speedup factor and *Speedup**, are employed in this chapter. The latter is calculated by excluding the timing of the first time step. It is therefore, more representative to be compared between different datasets since different datasets usually have different number of time steps and rendering of the first volume steps normally takes longer time than that of the subsequent steps.

The speedup results tested based on each dataset are given in Table 4.10 to Table 4.14 and the cycle rendering timings are compared and illustrated in Figure 4.26 to Figure 4.30, where Tex2D/3D stands for 2D/3D texture-based rendering, *Regular TM* is regular texture-mapped rendering and the MVD *dataset names* represents the corresponding cluster-based rendering of each dataset.

Table 4.10 Cycle rendering time (in seconds) and speedup of cluster-based rendering over regular texture-mapped rendering of the *HAND* dataset

		Regular TM	*Hands A*	*Hands B*	*Hands C*
Tex2D	**Total Time**	30.339	4.252	3.479	3.214
	Speedup	–	7.14	8.72	9.44
	Speedup*	–	11.45	18.02	22.34
Tex3D	**Total Time**	1.801	1.156	1.095	1.064
	Speedup	–	1.56	1.64	1.69
	Speedup*	–	1.71	1.86	1.93

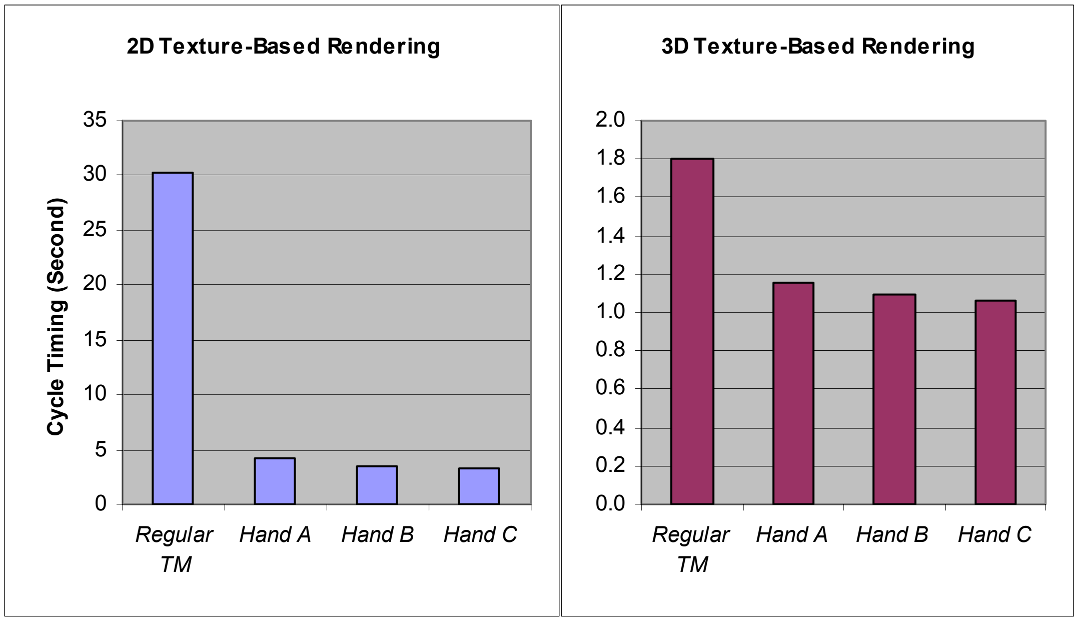


Figure 4.26 Comparison of the cycle rendering time between cluster-based rendering and regular texture-mapped rendering of the *HAND* dataset

Table 4.11 Cycle rendering time (in seconds) and speedup of cluster-based rendering over regular texture-mapped rendering of the *BREAST* dataset

		Regular TM	***Breast A***	***Breast B***	***Breast C***
Tex2D	**Total Time**	1.614	0.301	0.300	0.249
	Speedup	–	5.35	5.38	6.49
	Speedup*	–	6.56	6.60	8.93
Tex3D	**Total Time**	0.111	0.090	0.088	0.084
	Speedup	–	1.22	1.27	1.31
	Speedup*	–	1.28	1.34	1.40

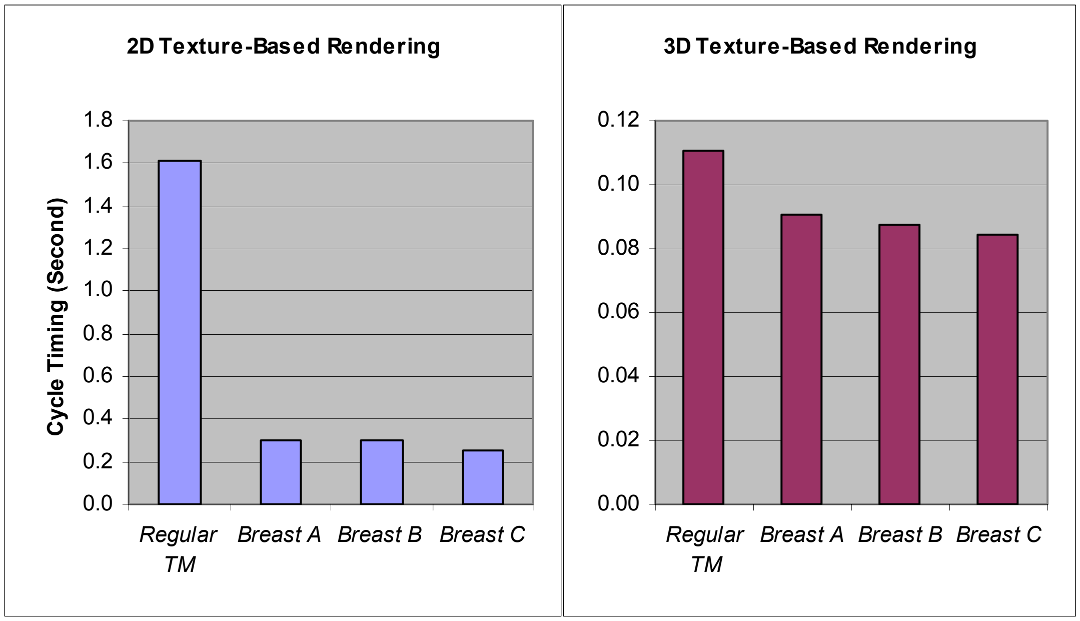


Figure 4.27 Comparison of the cycle rendering time between cluster-based rendering and regular texture-mapped rendering of the *BREAST* dataset

Table 4.12 Cycle rendering time (in seconds) and speedup of cluster-based rendering over regular texture-mapped rendering of the *HEART I* dataset

		Regular TM	*Heart I A*	*Heart I B*	*Heart I C*
Tex2D	**Total Time**	1.165	0.322	0.247	0.209
	Speedup	–	3.62	4.71	5.56
	Speedup*	–	4.08	5.63	6.95
Tex3D	**Total Time**	0.408	0.183	0.170	0.165
	Speedup	–	2.23	2.39	2.47
	Speedup*	–	2.27	2.45	2.53

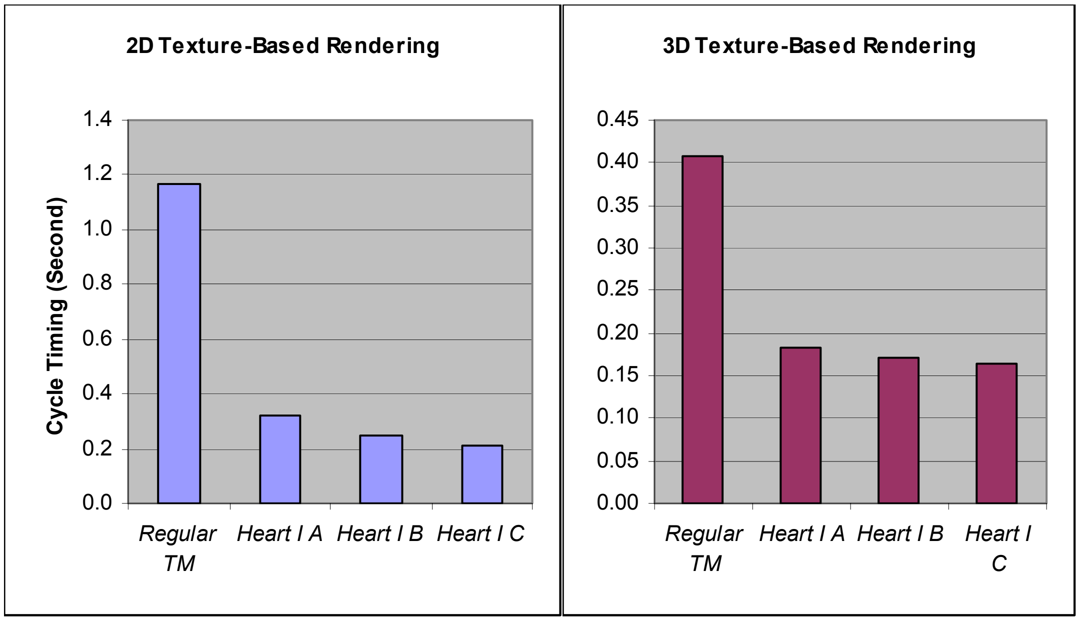


Figure 4.28 Comparison of the cycle rendering time between cluster-based rendering and regular texture-mapped rendering of the *HEART I* dataset

Table 4.13 Cycle rendering time (in seconds) and speedup of cluster-based rendering over regular texture-mapped rendering of the *HEART II* dataset

		Regular TM	*Heart II A*	*Heart II B*	*Heart II C*
Tex2D	**Total Time**	0.670	0.298	0.193	0.141
	Speedup	–	2.25	3.48	4.76
	Speedup*	–	2.36	3.87	5.65
Tex3D	**Total Time**	0.237	0.110	0.099	0.087
	Speedup	–	2.15	2.40	2.73
	Speedup*	–	2.19	2.46	2.83

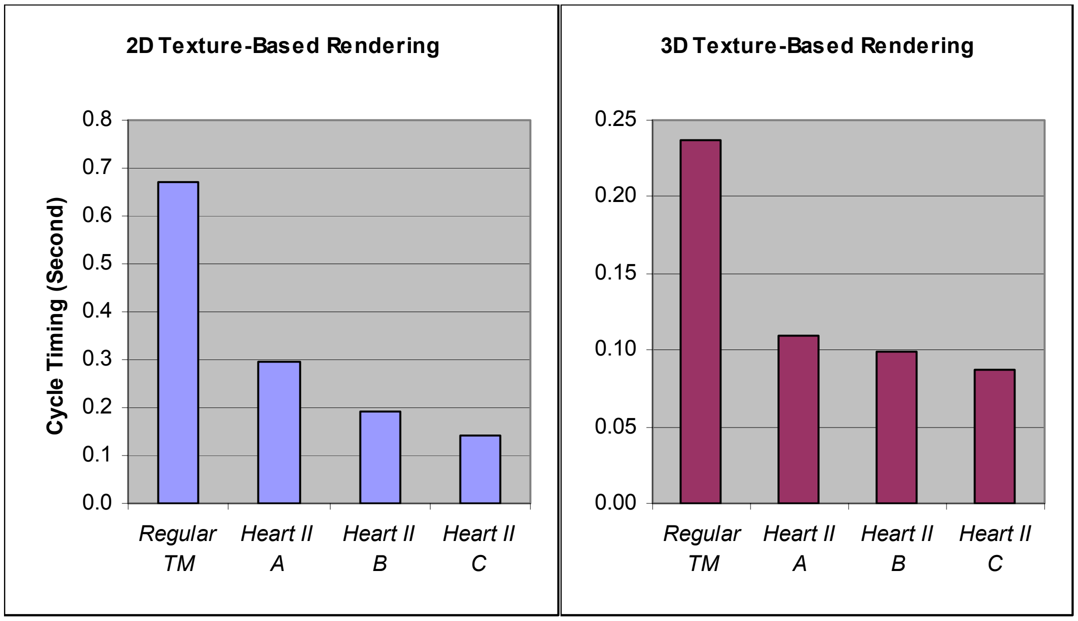


Figure 4.29 Comparison of the cycle rendering time between cluster-based rendering and regular texture-mapped rendering of the *HEART II* dataset

Table 4.14 Cycle rendering time (in seconds) and speedup of cluster-based rendering over regular texture-mapped rendering of the *ABDOMEN* dataset

		Regular TM	*Abdomen A*	*Abdomen B*	*Abdomen C*
Tex2D	**Total Time**	6.044	1.517	1.318	1.177
	Speedup	–	3.98	4.59	5.14
	Speedup*	–	4.02	4.65	5.23
Tex3D	**Total Time**	0.588	0.369	0.362	0.338
	Speedup	–	1.59	1.63	1.74
	Speedup*	–	1.60	1.63	1.75

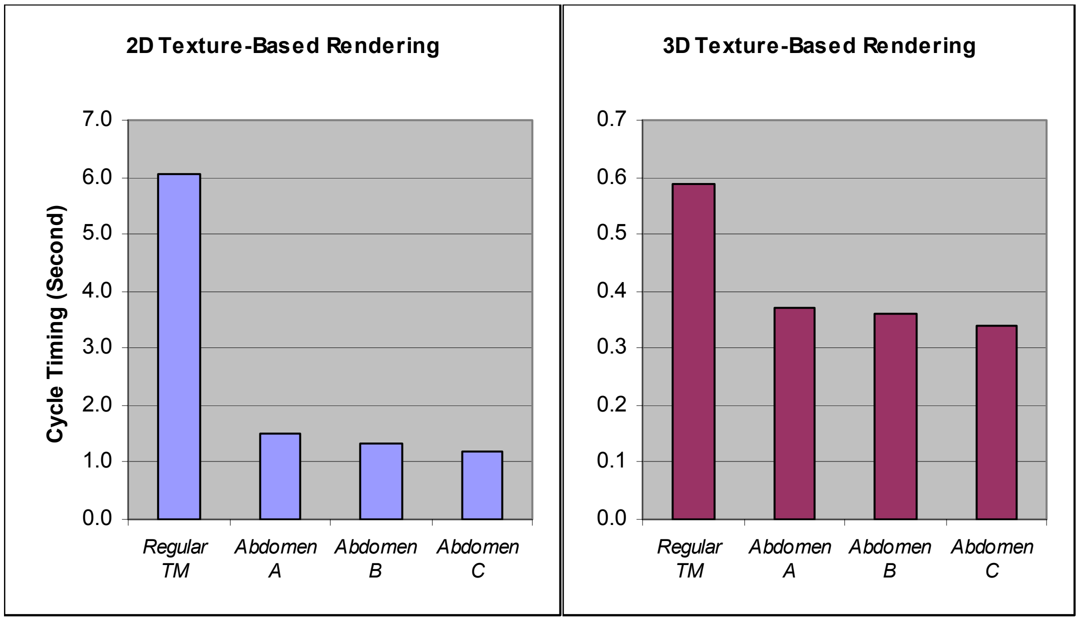


Figure 4.30 Comparison of the cycle rendering time between cluster-based rendering and regular texture-mapped rendering of the *ABDOMEN* dataset

In 2D texture-based rendering, up to 9.4 times of speedup was achieved through the introduction of the proposed cluster-based rendering algorithm, and the speedup could be more than 22 times in terms of *Speedup**. For the datasets *HAND* and *BREAST*, the regular texture-mapping renderer is too slow to provide interactive rendering (above 5 frames per second), which is however, successfully achieved through the cluster-based renderer. For other datasets, the cluster-based renderer realizes real-time rendering (above 25 frames per second) with even higher frame rates. In 3D texture-based rendering, we also achieved up to 2.7 times speedup for the cluster-based renderer. 3D texture-based rendering performs about 10 times faster than 2D texture-based rendering according to the experimental results. It avoids the operations of volume rebuilding as required in 2D texture-based rendering. The speed acceleration therefore, appears less significant. But the speedup is meaningful for high-quality rendering (above 60 frames per second) and large-scale 4D volume rendering.

The proposed algorithm essentially performs a lossy compression of the time-varying volume data. It is necessary to analyze the impact of the compression scheme on the visual quality of rendering. Similar to Chapter 3, the regression testing method from VTK is employed for this purpose. In this experiment, all the images are generated in the size of 500 × 500 24-bit color pixels with red, green and blue channels, while each pixel is assigned with the same value in all three channels and the images thus appear in gray. A threshold tolerance of 5 is used in the calculation of thresholded errors. It is less than 2% of the maximum pixel difference and normally is hard to be noticed by human eyes.

The images generated by the regular texture-mapping method are employed as the valid images, and the image quality of cluster-based rendering is evaluated based on the following procedures. For each dataset, the regression testing is applied to each pair of corresponding images at each time step. The absolute error (E_A) and thresholded error (E_T) of this time step are calculated accordingly. After the regression testing is finished for all time steps, the results are averaged and used to represent the error of cluster-based rendering of this dataset. Based on the images generated by the regular texture-mapping method, the regression testing is also applied between the images at successive time steps. The results are averaged and used to indicate the inter-step differences. It provides us an effective reference to evaluate the rendering quality we have achieved. The inter-step errors also serve as a good measurement of the coherence of the dataset.

The error analysis of cluster-based rendering was performed to all the datasets and the results are shown in Table 4.15 to Table 4.19. It should be noted that all the images must be of the same size so that the regression errors can be compared with each other.

Table 4.15 Error analysis of cluster-based rendering of *HAND* dataset

Name	**Absolute Error (E_A)**	**Thresholded Error (E_T)**
Inter-Step	547.113	104.500
Hands A	15.796	0.009
Hands B	17.938	0.059
Hands C	30.066	0.541

Table 4.16 Error analysis of cluster-based rendering of *BREAST* dataset

Name	**Absolute Error (E_A)**	**Thresholded Error (E_T)**
Inter-Step	1409.644	699.788
Breast A	7.088	0.207
Breast B	15.587	0.182
Breast C	36.298	1.405

Table 4.17 Error analysis of cluster-based rendering of *HEART I* dataset

Name	**Absolute Error (E_A)**	**Thresholded Error (E_T)**
Inter-Step	360.980	52.322
Heart I A	39.980	0.321
Heart I B	54.321	1.182
Heart I C	73.284	7.749

Table 4.18 Error analysis of cluster-based rendering of *HEART II* dataset

Name	**Absolute Error (E_A)**	**Thresholded Error (E_T)**
Inter-Step	650.801	105.331
Heart II A	122.655	11.593
Heart II B	141.360	17.224
Heart II C	151.624	17.859

Table 4.19 Error analysis of cluster-based rendering of *ABDOMEN* dataset

Name	Absolute Error (E_A)	Thresholded Error (E_T)
Inter-Step	2761.917	1658.556
Abdomen A	206.086	17.966
Abdomen B	254.729	36.396
Abdomen C	343.661	70.943

Compared to inter-step difference, all the results of error analysis appear small. It shows that the cluster-based algorithm achieves good rendering quality. The cluster *NED* threshold serves as the global error tolerance to effectively control the rendering quality. The regression testing errors and cluster *NED* threshold presented the same trend in rendering quality. For visual inspection, selected images are shown in Figure 4.31 to Figure 4.35 while the regression errors of each pair of images are also given. It shows that the loss of the image quality is visually tolerable for all the testing datasets.

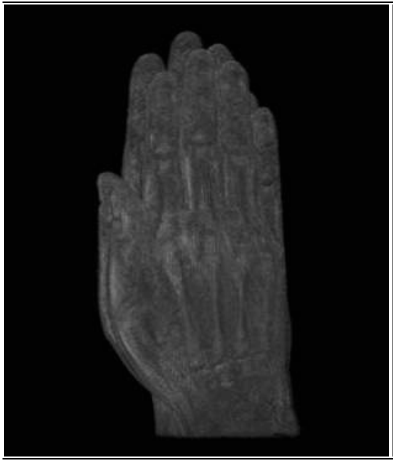
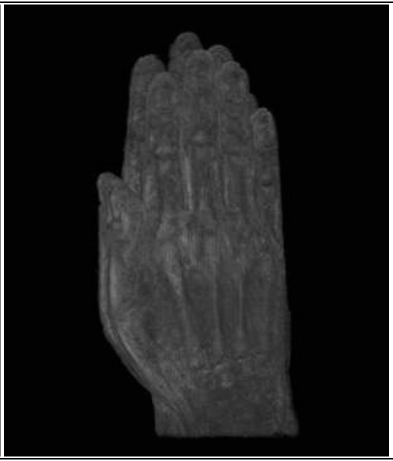
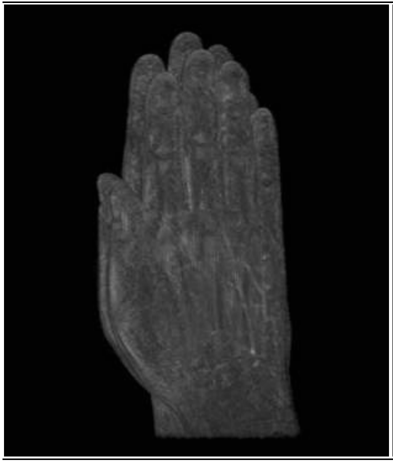
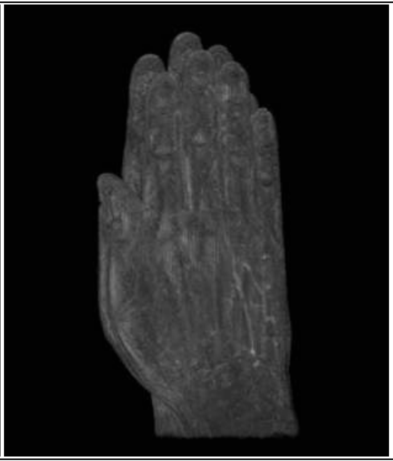
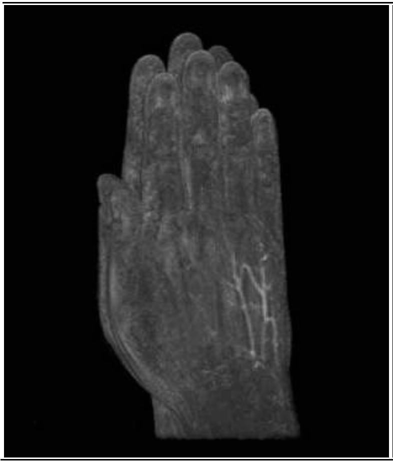
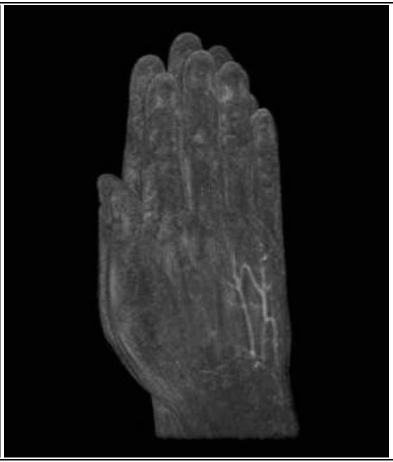
(a) Valid image	**(b) DLLO rendered image**	**(c) Regression Error**
		$\boldsymbol{E_A = 21.0}$ $\boldsymbol{E_T = 0.07}$ **(Step 1)**
		$\boldsymbol{E_A = 20.8}$ $\boldsymbol{E_T = 0.05}$ **(Step 3)**
		$\boldsymbol{E_A = 12.6}$ $\boldsymbol{E_T = 0.02}$ **(Step 5)**

Figure 4.31 Comparison of the image quality between regular texture-mapped rendering and cluster-based rendering of the *HAND* dataset (cluster *NED* Threshold = 0.15)

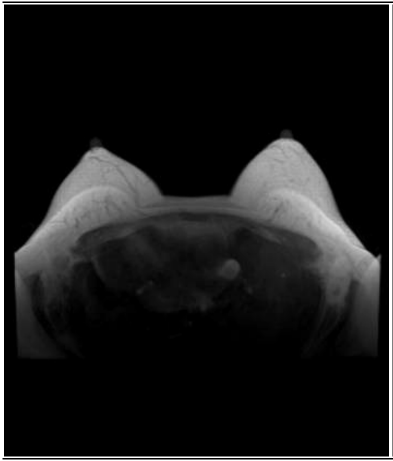
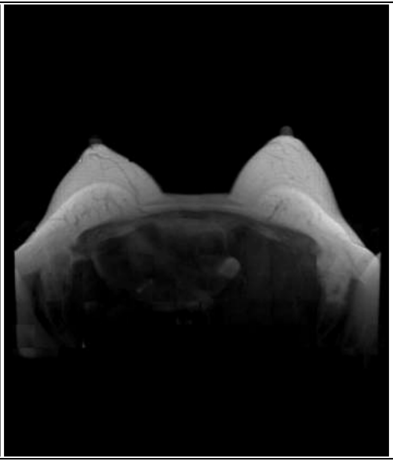
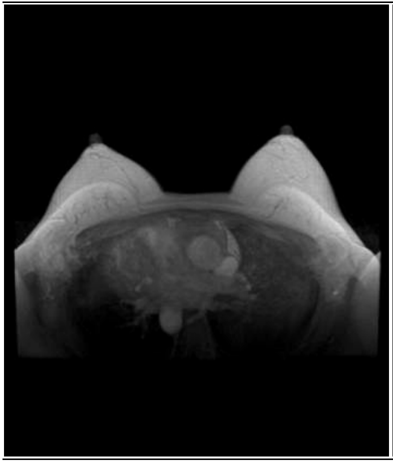
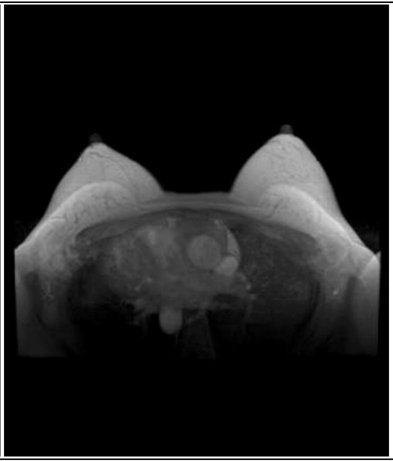
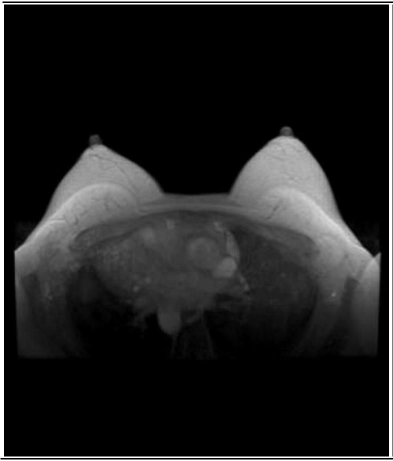
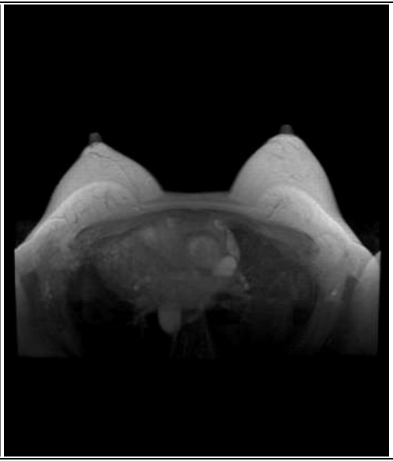
(a) Valid image	**(b) DLLO rendered image**	**(c) Regression Error**
		E_A **= 50.2** E_T **= 1.24** **(Step 1)**
		E_A **= 33.9** E_T **= 0.59** **(Step 3)**
		E_A **= 27.4** E_T **= 2.37** **(Step 5)**

Figure 4.32 Comparison of the image quality between regular texture-mapped rendering and cluster-based rendering of the *BREAST* dataset (cluster *NED* Threshold = 0.15)

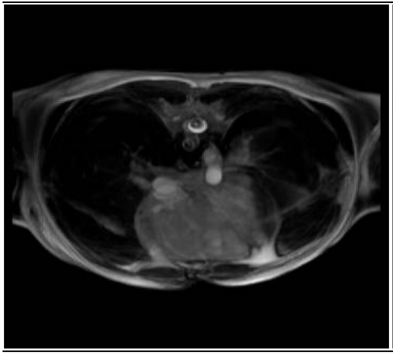
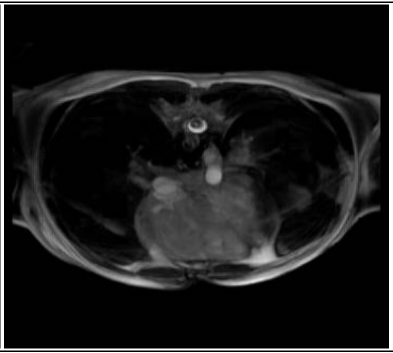
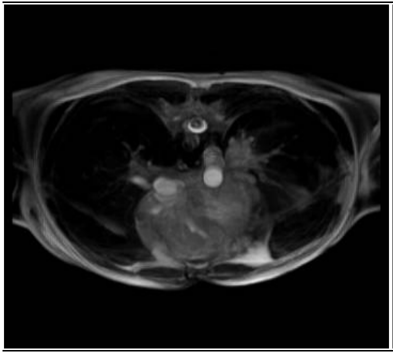
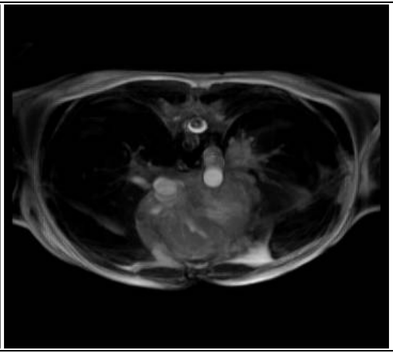
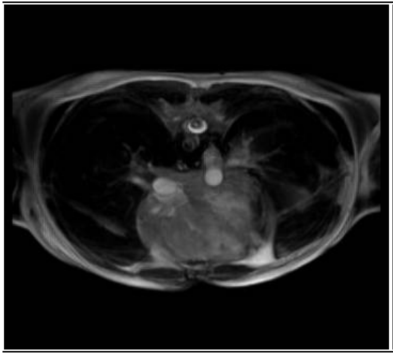
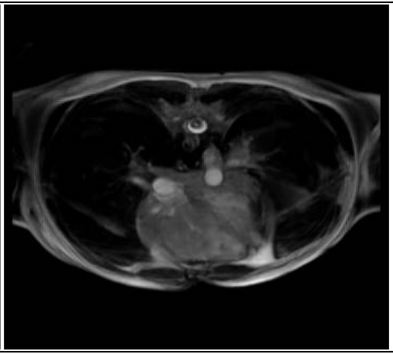
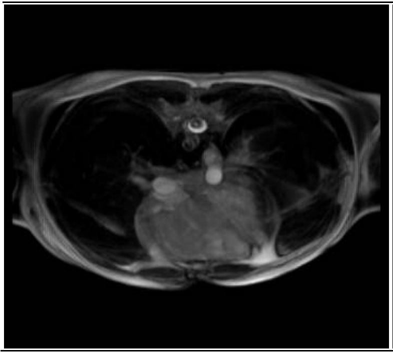
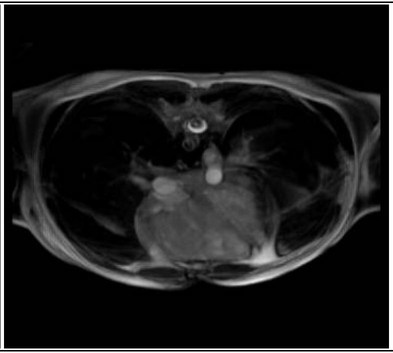
(a) Valid image	**(b) DLLO rendered image**	**(c) Regression Error**
		$\boldsymbol{E_A}$ **= 118** $\boldsymbol{E_T}$ **= 11.5** **(Step 1)**
		$\boldsymbol{E_A}$ **= 46.2** $\boldsymbol{E_T}$ **= 0.30** **(Step 7)**
		$\boldsymbol{E_A}$ **= 48.5** $\boldsymbol{E_T}$ **= 0.18** **(Step 13)**
		$\boldsymbol{E_A}$ **= 41.0** $\boldsymbol{E_T}$ **= 0.04** **(Step 20)**

Figure 4.33 Comparison of the image quality between regular texture-mapped rendering and cluster-based rendering of the *HEART I* dataset (cluster *NED* Threshold = 0.15)

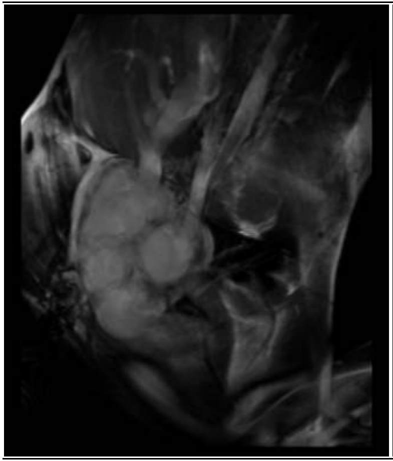
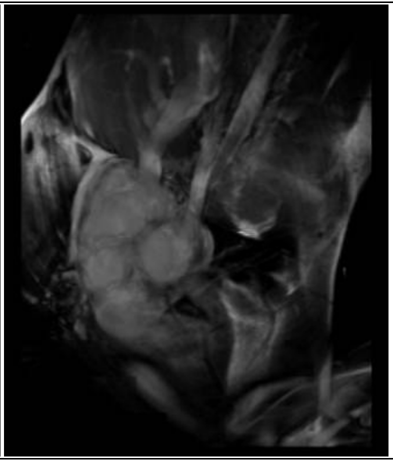
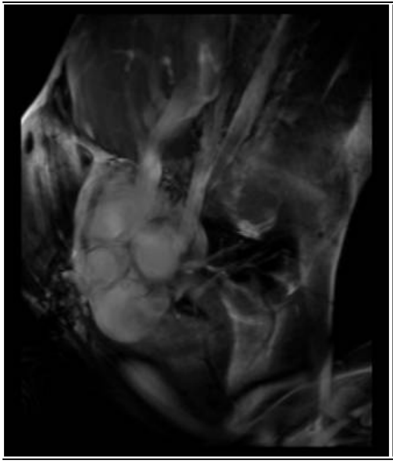
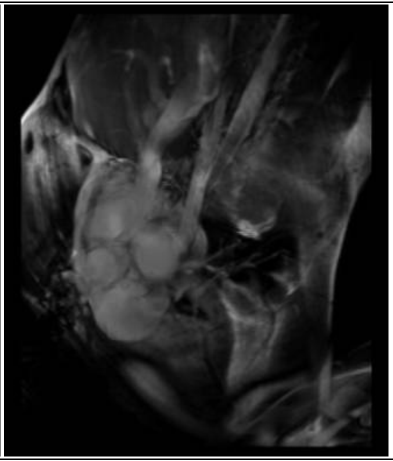
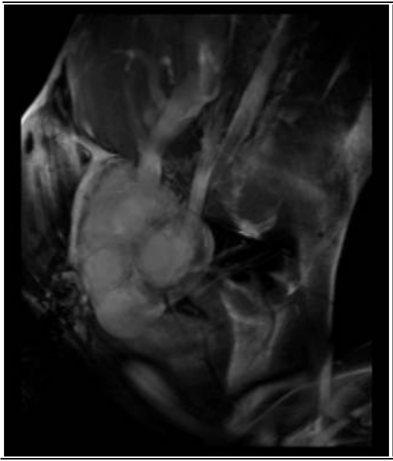
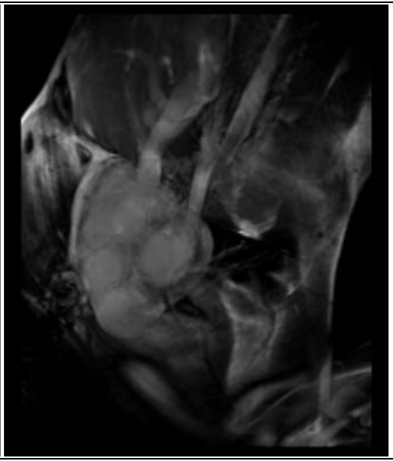
(a) Valid image	**(b) DLLO rendered image**	**(c) Regression Error**
		$\mathbf{E_A = 206}$ $\mathbf{E_T = 36.5}$ **(Step 1)**
		$\mathbf{E_A = 175}$ $\mathbf{E_T = 22.3}$ **(Step 11)**
		$\mathbf{E_A = 120}$ $\mathbf{E_T = 15.0}$ **(Step 20)**

Figure 4.34 Comparison of the image quality between regular texture-mapped rendering and cluster-based rendering of the *HEART II* dataset (cluster *NED* Threshold = 0.15)

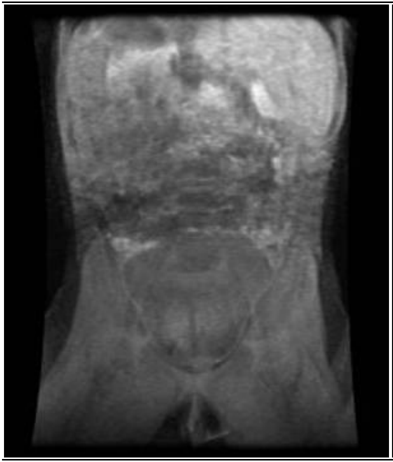
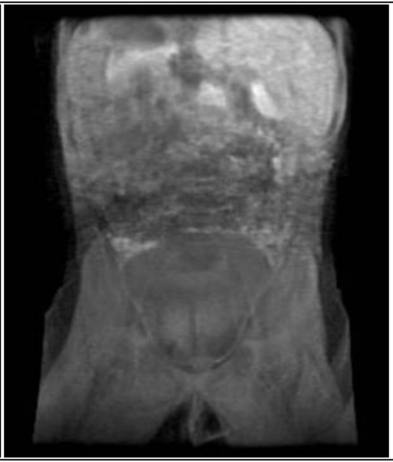
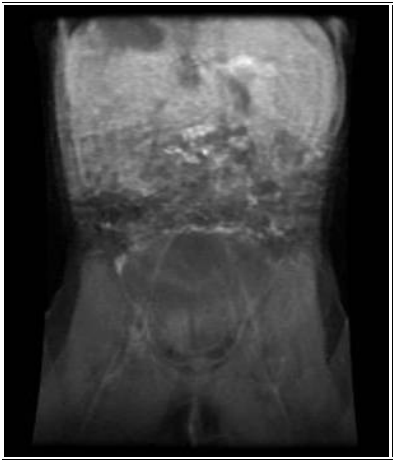
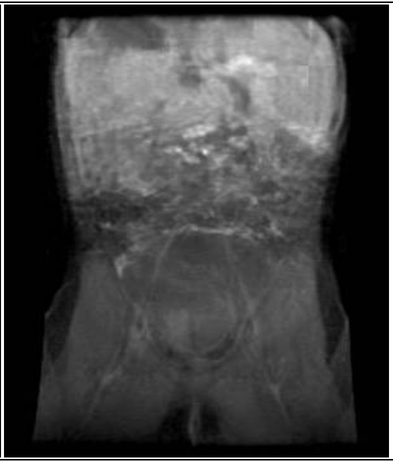
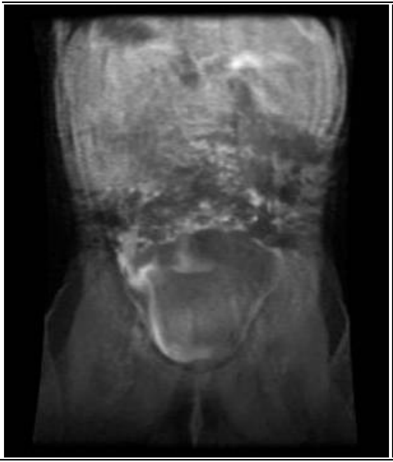
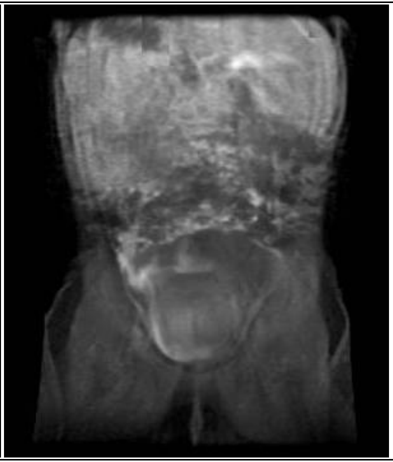
(a) Valid image	(b) DLLO rendered image	(c) Regression Error
		**E_A = 309** **E_T = 60.8** **(Step 1)**
		**E_A = 368** **E_T = 91.3** **(Step 19)**
		**E_A = 399** **E_T = 119** **(Step 39)**

Figure 4.35 Comparison of the image quality between regular texture-mapped rendering and cluster-based rendering of the *ABDOMEN* dataset (cluster *NED* Threshold = 0.20)

4.7 Summary

In this chapter, the cluster-based time-varying volume rendering algorithm is described. It provides a new means for efficient visualization of time-varying volume data. The algorithm takes advantage of the inherent characteristics of time-varying volume data more extensively. A new type of data coherence, namely global coherence, is exploited through the employment of the clustering technique so that rendering is benefited with enhanced performance. Because there is no restriction on the underlying type of renderers, the algorithm also provides sufficient space for further extensions.

Extensive experiments are performed based on the texture-based implementations of this algorithm. It achieves good performance in terms of both speed acceleration and space reduction. Results reported demonstrate its superiority over the regular texture-based algorithms for time-varying volume rendering. The compression ratio obtained could be over 89% and up to 10 times acceleration has been achieved when it is compared to the regular texture-mapped rendering. Based on the analytical results of regression testing, errors introduced due to clustering tolerance are visually small and high rendering fidelity can be achieved.

Chapter 5

Medical Simulation Application in Image-Guided Surgeries

5.1 Introduction

Medical images usually comprise of volumetric data with equal or variable spacing along each axis. The data models are inherently fuzzy and boundaries are transitional zones. Direct volume rendering creates an image from the volume without generating an intermediate geometrical representation. Therefore, it can effectively visualize the fuzzy quality of semi-transparent images. In comparison with surface rendering, there is no explicit need for binary classification about what is and isn't part of a feature. Segmentation which is a key research problem in medical engineering may be avoided. Recently, Hata et al. [2003] used 3D volume rendering to help differentiate congenital cystic adenomatoid malformation from congenital diaphragmatic hernia in preparation for fetus surgery. The differentiation will be very difficult if the clinician relied only on 2D MR slides. Animated visualization of time-series volumetric medical data was used in a study of brain lesions due to the progression of multiple sclerosis [Plesniak et al. 2003]. The animated electro-holographic display provided clinicians an adequate assessment of the spatial-temporal distribution of brain lesions to anatomical structures of the brain. Ra et al. [2002] presented a simulator for spine needle biopsy, in which volume rendering is adopted for effective examination of relative locations of spine and organs and tracking the needle path in terms of both accuracy and reality.

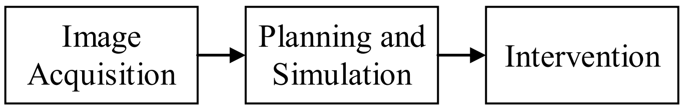


Figure 5.1 Overview of computer-aided image-guided surgery

Spine needle biopsy, as well as interventional radiology, microscopic microsurgery and vertebroplasty are examples of image-guided surgery. An objective of image-guided surgery is to use pre- and intra-operative medical images to help reduce invasiveness of surgery. They are part of the increasing popular minimally invasive surgery. An overview of the processes in the image-guided surgery considered in this thesis is shown in Figure 5.1. As reasoned above, volume rendering is our preferred visualization technique for planning and simulation. Medical images of patients are first acquired using common diagnostic imaging modalities such as CT and MRI. Based on the volumetric images, patient specific anatomical models are reconstructed along with volume rendering of the medical imaging data for planning and simulation. Consequently, the formulated surgical plan is then applied to help guide the procedure during intervention.

In the following sections, I will describe the application of my proposed 4D volume rendering algorithms in simulating interventional radiology, microsurgery and the development of a virtual spine workstation.

5.2 Interventional Radiology Procedures

5.2.1 Background

Interventional Radiology (IR) is a subspecialty of medical imaging, in which minimally invasive procedures are performed using image guidance, usually X-ray fluoroscopy or CT. Some of these procedures are done for purely diagnostic purposes (e.g., angiograms), while others are done for treatment purposes (e.g., angioplasties). Images are used to direct these procedures, which are usually done with needles or other tiny instruments like small tubes (called catheters). The images provide road maps that allow the interventional radiologist to guide these instruments through the body to the areas of interest.

5.2.2 Catheterization Simulator

Catheterization is one of the main applications for interventional radiology. Such procedure usually involves manipulating plastic catheters and guide-wires through blood vessels to the site of the lesion and then treating the lesions by means of devices or drugs delivered through the catheters. An effective simulation system should provide interventional neuroradiologists with tools to examine patient-specific anatomy through 3D visualization and to interact with the vascular images in real time. We have developed an interventional neuroradiology pretreatment planning system to provide the above capabilities and the setup of the system is shown in Figure 5.2 [Chui et al. 2002a].

The system allows physicians to manipulate and interface interventional devices such as catheters, guidewires, stents and coils within 2D and hybrid surface and volume rendered 3D patient vascular images in real time. It provides a pretreatment planning environment closely resembling the angiography suite and allows clinicians to interact with patient specific vasculature in virtual space using actual interventional devices.

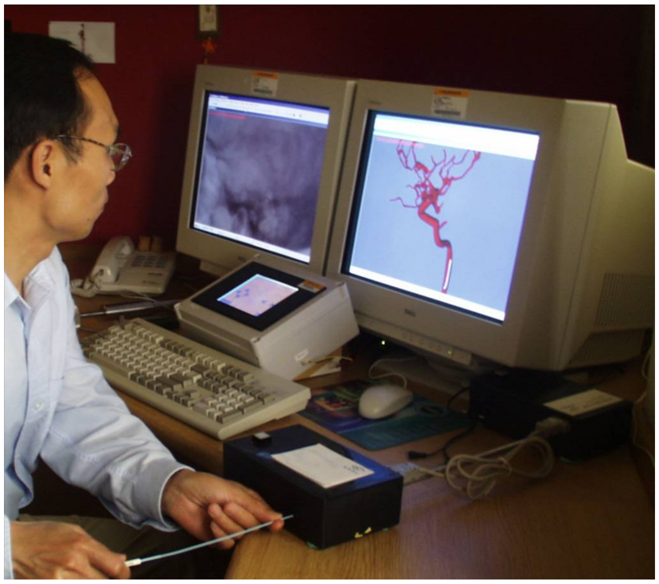


Figure 5.2 Physical setup of the simulation system

Volume rendering as a key component of the simulation system is to produce angiography[12] and 3D/4D views of human brain and its vasculature system. The LLO-based volume renderer is part of our effort in developing PC-based interactive medical simulator [Wang et al. 2002b].

[12] Examination of the blood vessels using X-rays following the injection of a radiopaque substance to look for abnormalities

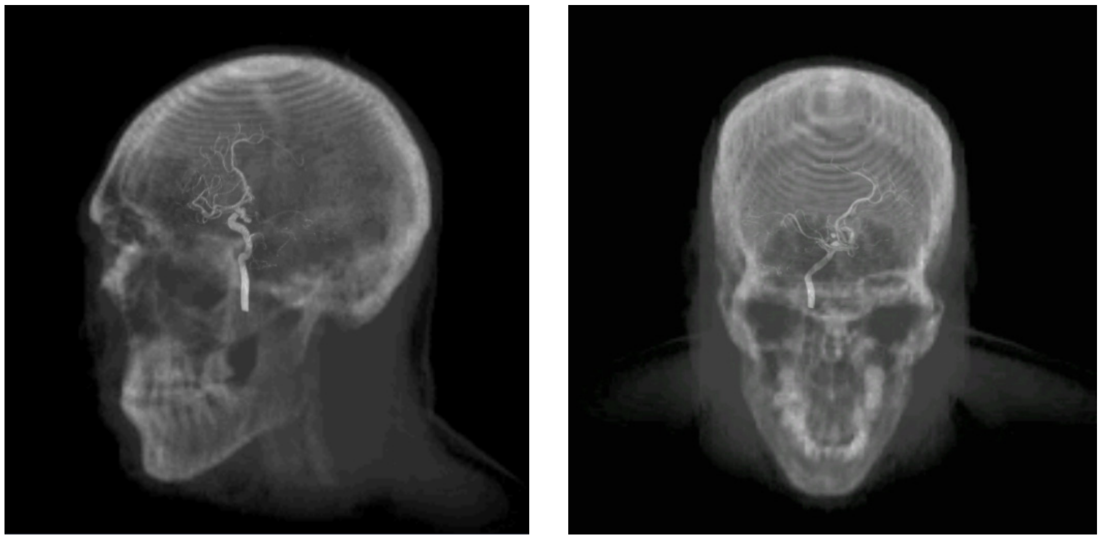


Figure 5.3 Multimodality rendering

Figure 5.3 demonstrates the LLO-based ray-caster used for multimodality rendering. The two datasets used are VHD male and a patient's cerebral angiography. The former has resolution of 256×256 and involves a total of 85 slices with 5 mm inter-slice gap, and is acquired by multi-slice CT scanner. The latter is a reconstructed rotational X-ray angiography (XRA) data. The CT images and reconstructed XRA are registered and visualized on a PC.

5.3 Microsurgical Simulation System

5.3.1 Background

Microsurgical techniques are being increasingly applied in many surgical disciplines because of its superiorities over conventional surgeries in terms of small incision, fast recovery and special capabilities in complex and delicate surgeries such as cataract removal. However, rare opportunities are available for trainees to learn and exercise these skills before practice in clinical situations. With the rapid advancement of virtual reality technology, a microsurgical simulation system is becoming possible by using the state-of-the-art virtual/augmented reality techniques for the purpose of training and even pretreatment planning based on patient-specific medical images.

Superior visualization, sufficient force feedback and interactive manipulation of surgical tools are regarded as main elements of creating a realistic virtual microsurgical environment. Visualization is critical as surgeons perform operations and make decisions mostly based on visual cues and the magnified view of operating area is inevitable in most microsurgeries as well [Salisbury and Kenneth 1998].

5.3.2 Craniotomy Simulator

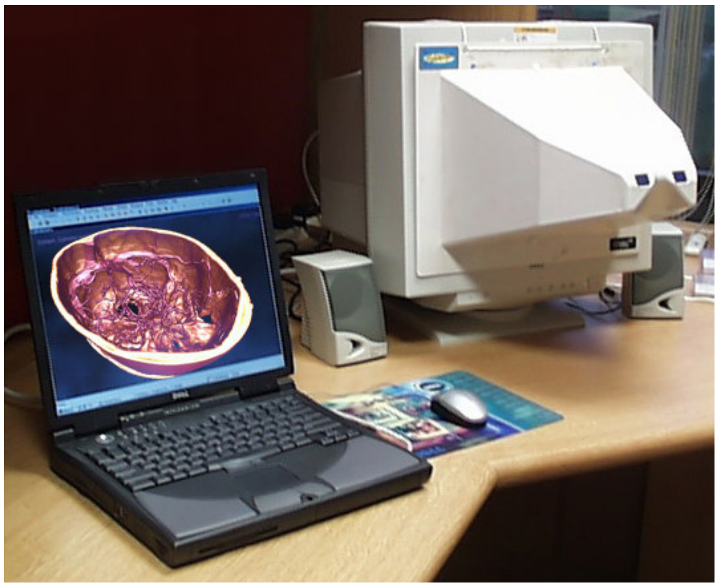


Figure 5.4 Overview of the microsurgical simulation system

An initial prototype of our microsurgical simulator for craniotomy is shown in Figure 5.4 [Wang et al. 2002a]. Craniotomy is a typical example of a microsurgical procedure. It involves a surgical removal of a portion of the skull to operate on a targeted region in the brain. The simulator is designed as a low cost system that can be comfortably placed on the desktop of a neurosurgeon. We aim to design and develop the software executed on a normal notebook computer with dual displays. The user will see the microscopic views with the stereoscope hanging in front of a computer monitor connected to a graphics port of the notebook computer. In the current set-up, we used a mid-range graphic acceleration board (32M RAM GeForce II). Microsurgical operations, such as tissue-cutting with micro-scissors, will be simulated using a force feedback probe. This set-up and the stereoscope resemble the type of microsurgical environment with high-resolution microscope typical in craniotomy. My proposed volume rendering technique is the primary method of computer graphics rendering in this simulation [Wang et al. 2004].

For surgical training, the trainee will first load a set of volume images comprising a clinical scan of a human head. In the planning mode, he/she can selectively render and inspect the various structures within the skull. Both perspective and parallel renderings are available for him/her during the study. The main purpose of this study was to allow the trainee to identify the target area and plan his/her approach. For pre-surgical planning, over 80% of the plan can be achieved with realistic 3D graphics. Only perspective rendering is available to the trainee when he/she is performing the "surgery" using a combination of virtual tools. These tools are modeled as simple geometrical objects.

Instructor, in our plan, could monitor progress of the training via display on the notebook computer - supervisor console. The supervisor console does not have to mirror the display on the stereoscope. The instructor may view a wider version of the rendered image the trainee is using during virtual surgery. The capability of the trainee neurosurgeons to reach the target region and removed the tumor tissue without hurting the surrounding healthy tissue is a good quantitative performance indication. However, we have yet to implement a scoring module to help track the trainee's performance.

Our priority of the simulation system was to duplicate the high power microscopic views during microsurgery. Proficiency of a neurosurgeon depends on how well a user can relay the stereoscopic view with his/her hands. It is unclear to us on the effectiveness of force feedback for this simulation. Hence, our emphasis in microsurgery simulation and particularly, training of microsurgery procedure is on realistic visual effects. In addition to the provision of accurate and high-speed perspective rendering, our rendering engine is capable of handling the emerging 4D medical imaging data.

In the simulation of interactive microsurgical procedures, integrated rendering of virtual surgical instruments with the volume data is necessary for users to provide a realistic scene of virtual surgical environment. For simplicity, surgical instruments such as scalpels, forceps and needles are normally rendered with opaque geometric polygon primitives. I have achieved the embedded volume-geometric rendering by using the cutting-plane technique based on hardware-assisted volume rendering [Wang et al. 2001]. Incorporated with texture-based 3D/4D volume rendering, integrated rendering can be achieved more easily. Opaque geometric surgical instruments must be rendered before texture-mapped volume rendering so that depth buffer test is enabled. Afterwards, the graphics hardware is able to blend the geometric polygons with the texture images correctly and efficiently.

Results obtained here were produced based on an implementation of the microsurgical simulator on a laptop computer with a Pentium III 900MHz processor, 384MB system memory and GeForce II graphics hardware. The LLO/DLLO-based volume renderer is used as the visualization component of the system.

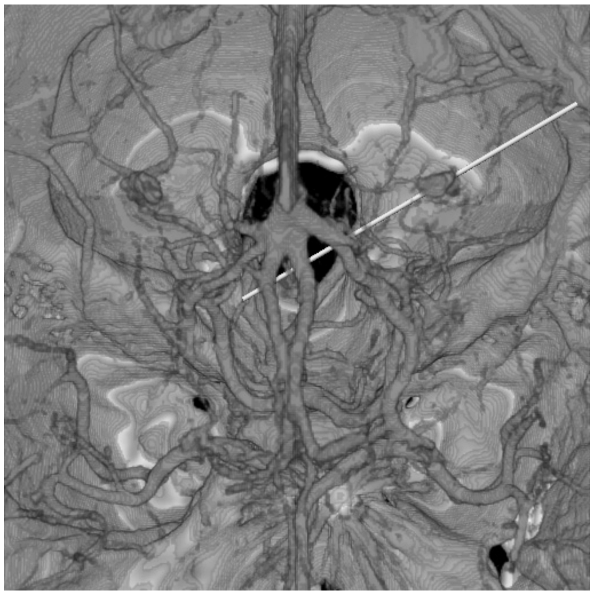


Figure 5.5 Perspective rendering of phantom head interacted with a virtual surgical needle

A CT scan of a phantom head dataset was used in our microsurgical simulator. The dataset comprises 229 slices with an imaging matrix of 512×512 pixels. In Figure 5.5, the phantom head was rendered in perspective projection. A virtual surgical needle was rendered with geometric polygons and embedded in the volume rendered scene, and its interaction with brain vasculature is clearly displayed in the virtual surgery mode. Note that the white matters were hidden for the user to focus on the vessels. Interactive visualization of the integrated surface-volume is achieved during simulation.

There are requirements on visualization that are unique in simulation of microsurgery procedure. For example, rendering has to be fast to respond to the change in viewing angle and movement of surgical devices. The rendering process is also made complex with the inclusion of 4D medical data. From the results of our implementation, the proposed method was clearly a possible solution for this complex visualization process.

One of the significant usages of microsurgery is to connect/repair vessels, termed microvascular anastomoses. For example, when vessels of a hand are involved, the surgeon needs to align the vessels from a cut finger with that of the main hand. A smooth blood flow implies that the vessel has connected well. Multiple scans of the hand while contrast injection is used to capture the blood flow. 4D or time-varying visualization is meaningful for surgeons to observe such procedure. Figure 5.6 shows the maximum intensity projection (MIP) of such a time-varying hand dataset.

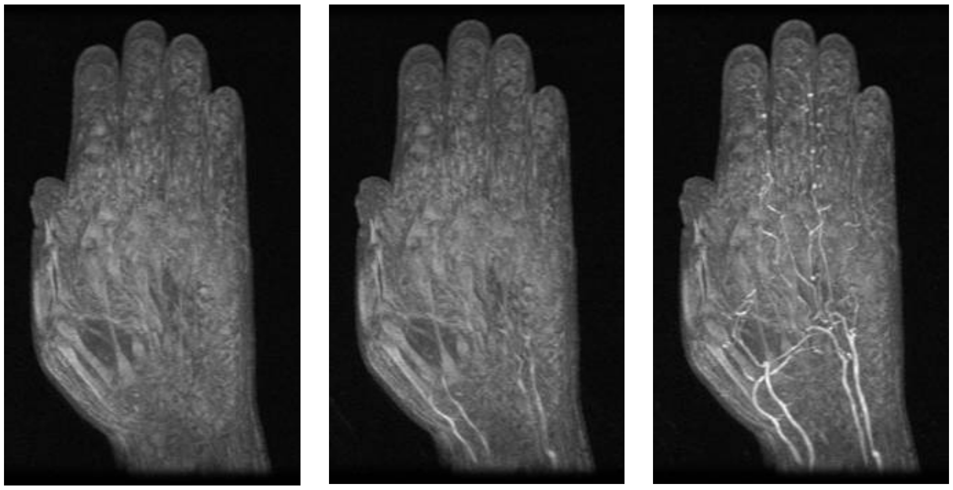


Figure 5.6 Time-varying volume rendering of a hand dataset in MIP

5.4 Virtual Spine Workstation

5.4.1 Background

The objective of Virtual Spine Workstation (VSW) is to develop a surgical workstation to provide realistic and patient-specific simulation of image-guided spine surgical procedures. The workstation aims at medical education and training, information-enhanced pre-operative planning for image-guided spine procedures and scientific analyses. The workstation focuses on the integration of different medical imaging technologies with computational technology and spine biomechanics to create a complete suite for assisting spine surgery [Teoh 2005; Wang et al. 2005b; Chui et al. 2002b].

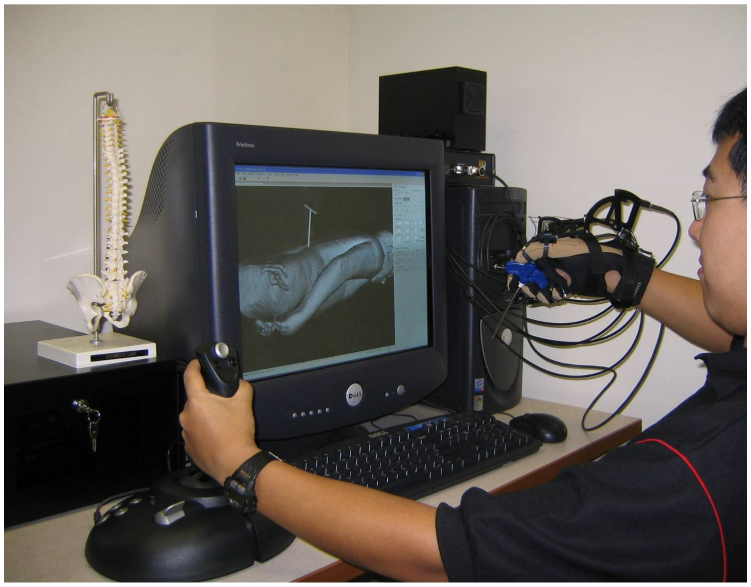


Figure 5.7 Overview of human-computer interaction in Virtual Spine Workstation

As illustrated in Figure 5.7, a user wearing a force feedback glove (CyberGrasp, Immersion, USA) with a real surgical needle probably provides the interaction closest to the actual procedure. With patient-specific volume visualization, users are able to manipulate and view the spine models at their desired view points as well as interactively subject the models to surgical instrumentation commonly used in the operating theatre.

3D/4D volume rendering of the VSW will help in medical courses on the spine, offering a new way of clear anatomical observation without the need for the present intrusive method. It provides realistic anatomical visualization for the spine which will help in better understanding of the spinal structure and help as an efficient training tool. Thus, the surgeons can plan treatment or surgery through good visualization by getting a feel of the actual procedure involved. The VSW, therefore, aids in treatment selection and optimization for surgical planning by interacting imaging and therapy.

The vertebroplasty, an image-guided minimally invasive therapeutic procedure, is identified as the first surgical application with VSW. The technology based and component approaches taken in the solution also enable other spine surgical applications to be adopted with ease.

5.4.2 Vertebroplasty Simulator

Osteoporosis is a major health problem throughout the United States and the developed world. In the United States alone, more than 700,000 vertebral body fractures are diagnosed each year, resulting in more than 100,000 hospital admissions[13]. Vertebroplasty is a relatively new procedure designed to help patients with osteoporosis or other fractures of the spine. Patients lose mineralization of the bones making them prone to compression fractures that can be very painful. It has been found that injecting the affected vertebral bodies with a

[13] Common lower back painful conditions, *http://www.painsolvers.com/CP_back.html*

special preparation of orthopedic cement can greatly relieve this pain. The cement can harden the vertebral body structure and helps to stabilize the bone and prevent further collapse and pain. About 90% of patients experience rapid relief of pain within 24 hours after the procedure [Wilfred et al. 2002]. Under our simulation system, with the guidance of seeable internal vertebrae 3D/4D representations of the patient-specific images (X-rays, CT scan or MRI scan), the surgeon will work on the virtual vertebrae with a virtual needle that can be easily placed through the skin of the patient in the back into the vertebra in the spine. Once the needle is properly positioned, bone cement is slowly injected into the spine.

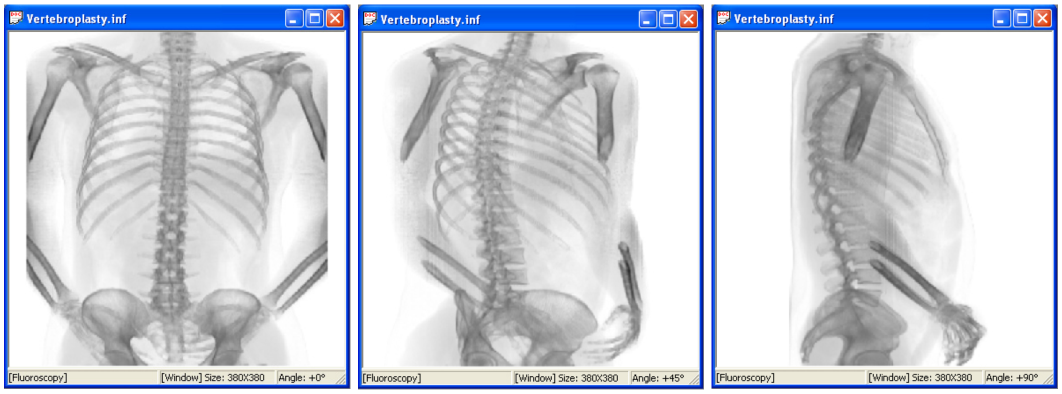
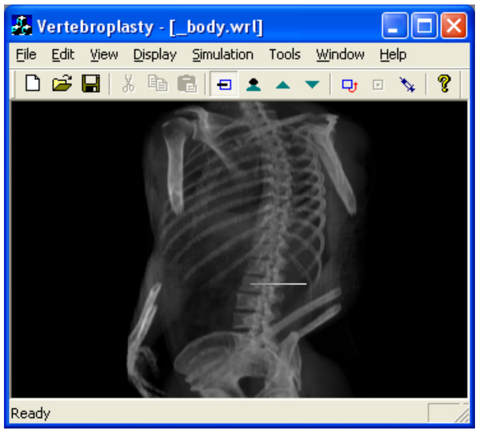


Figure 5.8 Fluoroscopic images

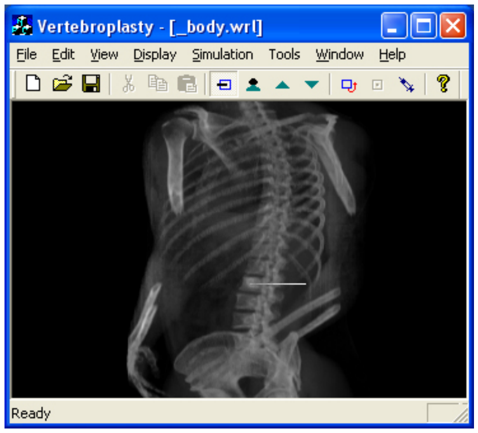
Fluoroscopic[14] guidance is used to pass a needle, by hand, into the affected vertebral body in order to inject bone cement. In the simulation system, there will be no patient available for the real-time fluoroscopy and the images have to be produced based on the patient-specific radiological scans. Figure 5.8 shows the fluoroscopic images generated by using the volume rendering technique.

[14] Fluoroscopy is a variation of X-ray technology in which a continuous X-ray beam is used to assess internal organs or objects in motion. Although the beam is on continuously, the dose is low compared with the amount of radiation from a traditional X-ray.

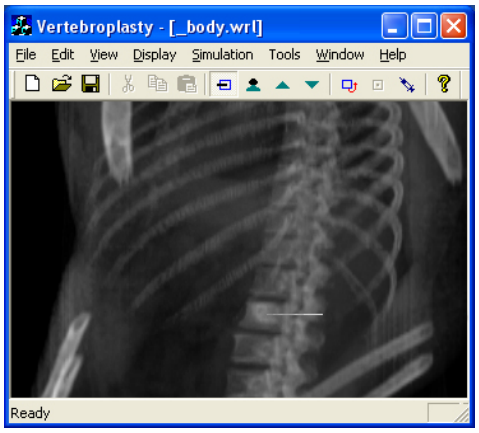
When a needle is inserted to the fractured vertebrae, a bone cement solution is injected at the site of the fracture. A potential-field-based approach was developed to simulate the variations in the vertebrae during such procedure. This process can be very sophisticated in that, based on it, surgeons are able to evaluate whether a vertebroplasty plan is feasible and discover its defects such as the leakage or insufficient injection. As a result, a time-varying volume dataset is produced and the simulated procedure of the cement injection is visualized with a cluster-based 4D volume renderer.


Vertebroplasty - [_body.wrl]
File Edit View Display Simulation Tools Window Help
Ready

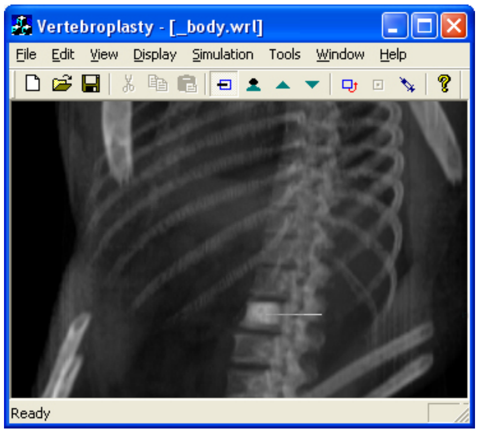
(a) Initial placement of needle into the vertebrae


Vertebroplasty - [_body.wrl]
File Edit View Display Simulation Tools Window Help
Ready

(b) Bone cement is injecting


Vertebroplasty - [_body.wrl]
File Edit View Display Simulation Tools Window Help
Ready

(c) Magnified view of the vertebrae with cement injection


Vertebroplasty - [_body.wrl]
File Edit View Display Simulation Tools Window Help
Ready

(d) The vertebrae after the bone cement injection

Figure 5.9 Time-varying volume rendering of the simulated procedure of the bone cement injection

The simulation is run on a desktop PC with an Intel 2.40GHz Pentium 4 processor and 512MB system memory. The simulation involves 12 sets of 3D volume and each volume has 219 slices with an imaging matrix of 256×256 pixels. The initial data is from the VHD Male Project and the spine in the dataset is normal and with no obvious pathology. It is used here for system validation only. Figure 5.9 illustrates the initial placement of the needle at the L2 vertebrae and the subsequence injection of the bone cement under the CT guidance (in contrast to the fluoroscopy guidance).

5.5 Summary

In this chapter, I reviewed the use of volume rendering in medical diagnosis/examination and simulations. My proposed volume rendering methods have been used in three different PC-based medical simulation systems. My solutions for the multi-dimensional volume rendering played an important role as the visualization component in them. In addition to time-varying volume rendering, results presented also include my work on angiography rendering, fluoroscopic rendering and multimodality rendering. The detailed implementations are not in the scope of this thesis. Interested reader can refer to Appendix B for more details on multimodality volume rendering.

Chapter 6

Discussion

6.1 Introduction

In this thesis, two new time-varying volume rendering algorithms, namely the DLLO-based and the cluster-based algorithms, have been presented. Both of them exhibit outstanding performance in 4D volume rendering as compared to conventional methods. I have compared the new algorithms with other time-varying volume rendering methods found in literatures.

In the following, I will first compare the performance of the new algorithms with several notable time-varying volume rendering algorithms. Then, the two new rendering methods will be further discussed by highlighting their advantages and disadvantages. Although the two algorithms proposed are meant for personal computers with a single processor, the performance of them can be further improved by using parallel computing. The parallelization of the two new algorithms is also discussed in this chapter.

6.2 Comparison of Time-Varying Volume Rendering Algorithms

Research in time-varying volume rendering has multiple objectives. Some are aimed at fast rendering while others are intended for high data compression. The two objectives are not always achievable simultaneously. For example, a compression-targeted method normally employs specialized compression schemes, however, this likely slow down rendering due to the decompression process. The goal of the two new algorithms proposed in this thesis is meant to provide time-varying volume rendering as fast as possible. Therefore, in the following, we are constrained to compare the rendering speed between the new algorithms and others. Although the new algorithms also achieved good results in data compression, it will be only treated as an additional gain in the comparison. In addition, unlike 3D volume rendering, researches in 4D volume rendering are still under way. There is not yet a single approach that is recognized as the one with the best performance. Since it is beyond our means to implement all the algorithms and compare them on the same platform, the comparison will mostly rely on the performance results reported in literatures. As the results are obtained under different conditions, it is not appropriate to compare them directly. Instead, their relative performance gains, e.g., speedup over the conventional methods, will be compared.

Table 6.1 compares the speedup performance of the TSP tree-based volume rendering method [Shen et al. 1999], two-level differential volume rendering method (TLDVRM) [Liao et al. 2003; 2004], texture-assisted rendering method using palette and discrete cosine transform (DCT) techniques [Lum et al. 2001], 4D volume rendering with shear-warp factorization method [Anagnostou et al. 2001] and my two new methods. In all these methods, speedup is calculated in comparison with the corresponding conventional/lossless methods based on the cycle timing of all time steps. Methods without explicit speedup information will have their results estimated based on the performance plots in the respective literatures. Except the TLDVRM, all the methods essentially perform lossy rendering with the degradation of rendering quality being visually tolerable according to the authors. All these methods have been reviewed in Chapter 2, so in the following discussion, their introduction will be omitted.

Table 6.1 Comparison of the speedup performance of different time-varying volume rendering algorithms

4D Rendering Method	Underlying Algorithm	Number of Test Datasets	Speedup	Data Compression
TSP tree	Ray-casting	3	3.4 ~ 5.2	Space overhead
TLDVRM	Ray-casting	3	1.6 ~ 1.8	Space overhead
Texture-assisted	2D texture	3	2.7 ~ 6.7	Yes
4D Shear-warp	Shear-warp	1	2.5 ~ 3.3	Yes
DLLO-based	Ray-casting	5	2.4 ~ 19.5	Yes
Cluster-based	2D/3D Texture	5	2.2 ~ 9.4	Yes

Although the speedup for each method has been implemented independently and on different datasets, the outcome of this comparison does infer that our new methods are at least on par with the existing methods if not better. Nevertheless, it is clear that we repeated the highest instantial speedup.

In the comparison of the TSP tree-based method and the DLLO-based method, although they both use octree, the DLLO-based method achieves higher speedup because the compressed data leads to reduced I/O cost. The DLLO-based method is not a compression-specialized algorithm. The volumes are compressed by discarding the redundant or less important raw data, so there is no extra cost for the decompression process during rendering. On the contrary, the TSP tree-based method uses the original 4D data and an auxiliary TSP tree, which not only consumes more memory but results in more I/O cost and poor cache hit rate.

In the TLDVRM, data coherence is only exploited in the image space. This is inadequate for the performance improvement. However, it is possible to employ the differential volume rendering scheme in the DLLO-based and the cluster-based algorithms so as to avoid updating unchanged pixels over time. The 4D Shear-warp rendering method also exploits the temporal coherence by the decomposition of the 4D data. However, the scheme employed is too straightforward to take advantage of the extensive data coherence as compared to the cluster-based method.

Lum et al.'s texture-assisted rendering method employs a compression-specialized algorithm, discrete cosine transform, for data encoding. The decompression is skillfully realized by using the color palette in graphics hardware. It therefore, achieves good speedup results. However, this method is too much restricted by the texture memory available. The performance decreases significantly when rendering is done out-of-core.

6.3 DLLO-Based and Cluster-Based Time-Varying Volume Rendering Algorithms

Both the DLLO-based and the cluster-based time-varying volume rendering algorithms employ a volume decomposition scheme to take advantage of the 4D data coherence. The former utilizes data coherence between time steps, while the later utilizes data coherence of the whole time sequence using a clustering technique. During rendering, both algorithms are characterized by reusing partial images for speed acceleration. Therefore, the performance of rendering is mostly determined by how efficient the DLLO or MVD data are encoded. The more extensive data coherence is identified and exploited during the encoding, the less I/O cost occurs between the hard disk, system memory and texture memory (if used) and this will also reduce the data elements that need to be rendered. For example, Figure 6.1 and Figure 6.2 compare the I/O throughput of the encoded *HAND* and *ABDOMEN* datasets by the two algorithms where the same *NED* threshold of 0.10 is employed as the temporal error tolerance and the global error tolerance respectively.

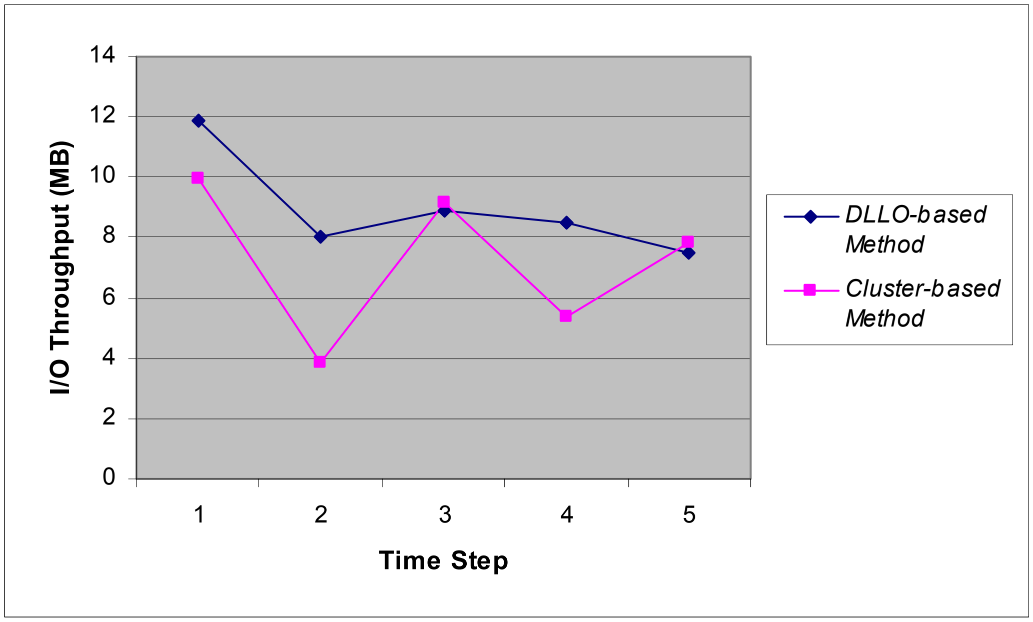


Figure 6.1 Comparison of the I/O throughput of the *HAND* dataset encoded by the DLLO-based method and the cluster-based method where the temporal error tolerance and global error tolerance of 0.10 is used, respectively

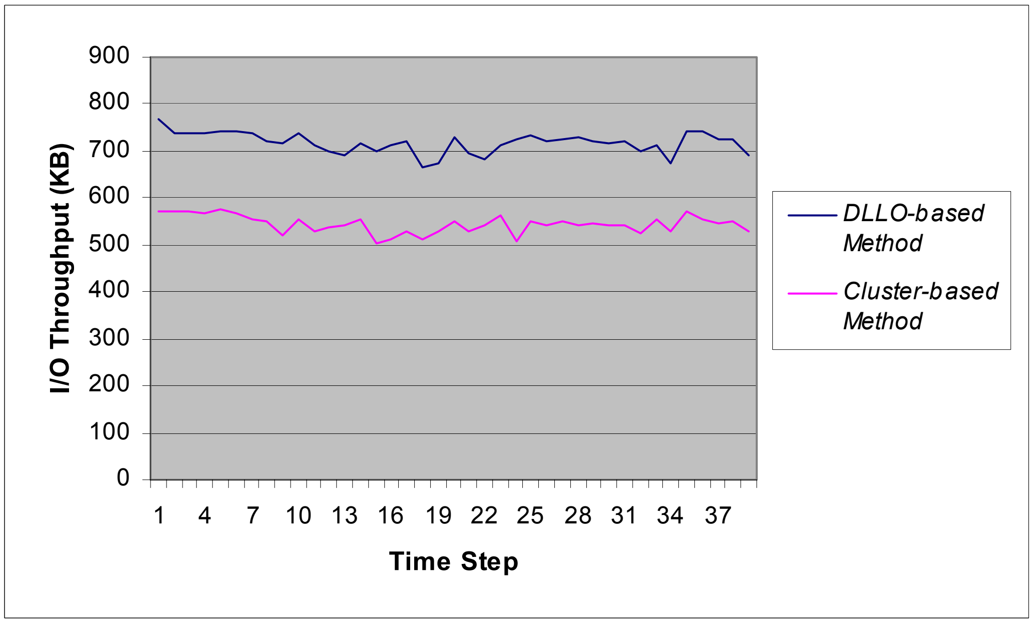


Figure 6.2 Comparison of the I/O throughput of the *ABDOMEN* dataset encoded by the DLLO-based method and the cluster-based method where the temporal error tolerance and global error tolerance of 0.10 is used, respectively

We observed the similar pattern based on all other 4D datasets that the resultant I/O throughput is lower in the cluster-based algorithm than in the DLLO-based algorithm under the same error tolerance. In the DLLO-based algorithm, homogeneous data are only exploited along the temporal direction and this homogeneousness must be continued so as to benefit the encoding process. When rendering the first time step of a 4D dataset using the DLLO-based algorithm, because there is no temporal coherence that can be utilized, the rendering time is almost the same no matter what temporal error tolerance is used. However, in the cluster-based algorithm, the homogeneous data are exploited along not only temporal direction but also spatial direction (i.e., truly in 4D). All the homogeneous data can be organized together to benefit the rendering performance with no requirement of the continued homogeneousness. When higher global error tolerance is employed, the rendering speed of both the first and the subsequent time steps increases as well. Due to the inherent characteristics of the DLLO, volumes can only be decomposed into cubes with the size of power of two, and the space reduction achieved may be compromised for the dataset with skewed sizes in spatial dimensions. On the contrary, the cluster-based algorithm adopts flexible decomposition scheme. Blocks can be divided with arbitrary size in three spatial directions, and it leads to better results in space reduction.

The cluster-based algorithm results in less I/O throughput than the DLLO-based algorithm, and thus performs faster 4D volume rendering when the same underlying renderer is employed. In Table 6.2, I compared the speed of two algorithms. Both are implemented based on 2D texture-mapping. For fair comparison, the minimum octant/block size is set as 16×16×16 and the same error tolerance (*NED*) of 0.10 is used for data coherence exploitation in both algorithms. The cycle rendering timings of two algorithms based on five dynamic MRI datasets are reported. The speedup of the cluster-based algorithm over the DLLO-based algorithm is calculated by the division of their cycle rendering timings. We can observe that the cluster-based algorithm performs faster rendering for all five test datasets and significant speedup can be achieved while rendering most of them. This performance gains are proportional to the savings due to global coherence in comparison with temporal coherence as reported in Chapter 4 (Table 4.9). Global coherence thus exhibits its superiority over other features of 4D data in 4D volume rendering acceleration.

Table 6.2 Cycle timing (in seconds) of DLLO-based rendering and cluster-based rendering of five dynamic MRI datasets and speedup results of cluster-based rendering over DLLO-based rendering

Dataset	*HAND*	*BREAST*	*HEART I*	*HEART II*	*ABDOMEN*
DLLO-Based Rendering	3.062	0.263	0.268	0.132	1.369
Cluster-Based Rendering	2.243	0.214	0.237	0.130	1.010
Speedup	1.37	1.23	1.13	1.01	1.36

However, in the current implementation of the cluster-based algorithm, MVD encoding takes more time than DLLO conversion. Therefore, the use of two algorithms should be determined according to the needs of applications. The DLLO-based method will perform well when there is a demand of instant visualization of a 4D dataset, whereas the cluster-based method should be used when there is adequate time for the construction of an MVD file or the resultant MVD file is for distribution so that users can enjoy faster rendering and smaller file size.

Although these two new algorithms are targeted at medical applications, they are also applicable to other applications. In computational simulations, geographical or oceanographic studies and biological studies, they also produce the time-varying volume data. These data have the similar characteristics of the 4D medical data. It is believed that the two new algorithms can also perform well in these applications.

6.4 Time-Varying Volume Rendering Parallelization

In parallel computing, a task can be split up and executed simultaneously on multiple processors to obtain faster results. When a 4D volume is large, parallel computing may enhance the capability of the rendering algorithms to provide interactive graphical display. Furthermore, recent advancement in computer graphics increases the demand for high quality rendering. With parallelism, it is possible that the proposed algorithms are capable to provide real-time rendering of medium size dataset in an even higher frame rate.

Parallel computing can be achieved through either multiple computers or a computer made up of multiple processors. Approaches to parallel computers include distributed computing, multiprocessor computing, cluster computing, grid computing, massively parallel computing etc. In distributed computing, multiple computers are organized to collaboratively run a single computational task. They communicate via some form of telecommunication networks and the cost of such communication is high compared to that between processors in the same computer. Cluster computing and grid computing are also characterized by using many computers in a network. The computers involved may be tightly coupled or loosely coupled. Multiprocessor computing is traditionally known as the use of multiple concurrent processes in one computer. It allows the computer to complete operations more quickly and to handle larger and more complex procedures through better utilization of resources.

The 4D volume rendering algorithms described in this thesis is intended for a personal computer or a microcomputer so as to keep the cost of resultant medical simulation systems low. With the advancement in hardware design, a personal computer system with two or more Central Processing Units (CPUs) becomes common and its price is no longer as high as before. Nowadays, multiprocessor systems are available commercially for end users and mainstream operating systems like Windows® and Linux already have built-in support for multiprocessing. Therefore, the proposed parallel 4D volume rendering algorithms are designed for execution on a personal computer system with multiple processors.

A multiprocessor computer can be classified as an MIMD[15] system. When the parallelization of the 4D rendering algorithms is implemented on a shared memory[16] multiprocessor computer, rendering tasks are distributed to processes working cooperatively. Processes are typically independent, carry considerable state information, have separate address spaces, and interact through system-provided inter-process communication (IPC) mechanisms. However, the context switching between processes is typically much more expensive than that of threads. Multiple threads, on the other hand, typically share the state information of a

[15] MIMD (multiple instruction stream multiple data stream) is a type of parallel computing architecture. Contrast with SIMD (single instruction stream multiple data stream), MISD and SISD.

[16] Shared memory refers to a block of Parallel Random Access (PRA) memory that can be accessed by several different CPUs in a multiprocessor computer system.

single process, share memory and other resources directly. A multi-threaded program can operate faster on computer systems that have multiple CPUs. Additionally, the recent Hyper Threading Technology[17] (HTT) makes a single physical processor appear as two logical processors and allows multiple threads to run simultaneously on the same CPU. Thus, even on a single processor system, the parallelized algorithms of time-varying volume rendering could achieve enhanced performance from a multi-threaded implementation.

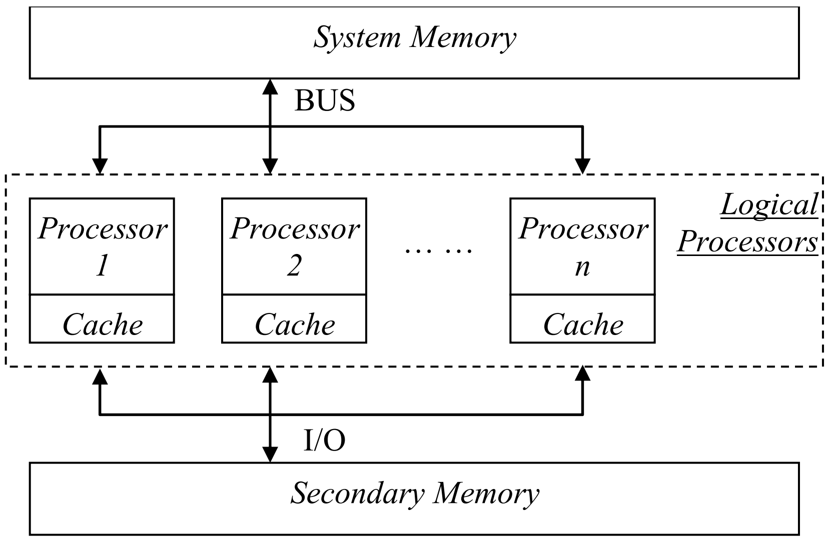


Figure 6.3 Illustration of multi-threading personal computer system architecture

The system architecture of the parallel 4D volume rendering algorithms is illustrated in Figure 6.3. Since parallel rendering is realized based on the multi-threaded programming, threads on different logical processors are able to communicate and interact with one another through the system memory and access the same file resource in the secondary memory

[17] HTT is a technology from Intel® for their implementation of the simultaneous multithreading technology on the Xeon™ and Pentium™ 4 processors. The technology improves processor performance under certain workloads by providing useful work for execution units that would otherwise be idle. An approximate 15-30% performance gain can be expected to be achieved.
http://www.intel.com/technology/itj/2002/volume06issue01/art01_hyper/p01_abstract.htm

(normally a hard disk) directly. The cache attached on each processor is possible to improve the efficiency of data accessing. The problems rising from memory/file accessing (reading and writing) are managed by the algorithm with the intervention of operating system to avoid deadlocks and race conditions.

The two 4D volume rendering algorithms proposed previously are redesigned to make effective use of the parallel hardware, and are described in the following two sub-sections respectively.

6.4.1 Parallelization of DLLO-Based 4D Volume Rendering

The parallelization of DLLO-based time-varying volume rendering consists of partitioning the computation into tasks for different threads/processors and choosing some appropriate synchronization mechanisms. The serial algorithm has three phases: DLLO construction, leaf octant rendering and partial-image composition. We parallelize this algorithm by considering the parallelization of each phase separately.

The DLLO construction is done with a concurrent traversal of the previous DLLO and the current *d*LLO. DLLO is designed to avoid loading the volume data of all time steps. Only octants of the current *d*LLO are read sequentially from the file at current time step and used to update the DLLO. Parallel reading of the same file will cause accessing conflicts and affect the overall performance. Since LLO allows the tree traversal to be fulfilled easily, the task of the DLLO construction is only allocated to one single thread. It plays mostly the same as that of the serial algorithm.

```
ConstructDLLO
{
      while (true)
      {
            // Read a leaf octant of current dLLO from the file
            octant = ReaddLLO();

            // Update the DLLO with octant
            UpdateDLLO(octant);

            // Update the octant-queue
            UpdateOctantQueue(octant);

            // Check whether there is no more octants in the dLLO
            if (octant == NULL)
            {
                  // A pseudo-end-octant is added to the queue
                  AddOctantQueue(PseudoEndOctant);
                  break;
            }
      }
}
```

Figure 6.4 Algorithm of DLLO construction for parallel rendering

However, leaf octant rendering is not required to start until the full DLLO construction is completed. The DLLO construction is not required to start until all the octants of the *d*LLO are loaded into memory either. On the contrary, the DLLO is updated while the reading of every octant of current *d*LLO from the file and at the same time the leaf nodes in updated sub-DLLO can be passed for rendering. Therefore, it is possible that the thread of DLLO construction runs with the threads of leaf octant rendering in parallel. The pseudo-code in Figure 6.4 shows the algorithm used by the thread for DLLO construction.

The parallelization is realized by the employment of a supplementary data structure: octant-queue, where the thread of DLLO construction places the octants needed for (re-)rendering.

The octant-queue is ended with a pseudo-end-octant (`PseudoEndOctant`), which is used to signify that all the leaf octants needed for rendering have already been added to the queue. When an octant rendering thread encounters this octant, it will be refrained from looking for new octants. An empty octant-queue cannot be used for this purpose because the queue could be empty from time to time when all the existing octants are rendered but new octants have not been read from the file yet. In the above algorithm, the behavior of the function `UpdateOctantQueue`( ) varies with the status of the model-view transformation and the transfer functions. If both of them have not been changed since the last time step, this function will only add the latest read octant (if it is not a pseudo-empty-node) to the queue; otherwise, all the leaf octants in the updated sub-DLLO are added to the octant-queue.

Octant rendering is the most computationally expensive phase of the three. It dominates the cost of the serial algorithm. Multiple threads, therefore, are employed for parallel octant rendering and the pseudo-code of the algorithm is given in Figure 6.5. A lock is used in the function `ReadFromOctantQueue`( ) to ensure that only one thread is accessing the octant-queue every time. Each thread will check the octant-queue repeatedly to obtain a new octant once current rendering is finished. Multiple threads will render octants in a preemptive and independent manner. A thread is terminated when the pseudo-end-octant is reached. The parallel rendering algorithm is then synchronized before the start of the next phase.

```
RenderOctant
{
        while (true)
        {
                // Read an octant from the octant-queue
                octant = ReadFromOctantQueue();

                // Check whether this is a valid octant
                if (octant == NULL)
                {
                        // Wait for new octants added into the queue
                        continue:
                }
                // Check whether all the octants have been rendered
                else if (octant == PseudoEndOctant)
                {
                        // Stop rendering
                        break;
                }
                else
                {
                        // Preventing other thread from rendering
                        // the same octant by removing it from the queue
                        RemoveOctantQueue(octant);

                        // Render the octant into a partial image
                        Render(octant);
                }
        }
}
```

Figure 6.5 Algorithm of parallel octant rendering

Parallelizing the computation of partial image composition can be achieved in two levels, namely full image level and partial image level. In the former level, the final image is divided into multiple uniform regions and each is distributed to one of the threads. Each thread composites only the partial images that contribute to the image area allocated. One disadvantage of this method is that load balancing is likely to be poor. Since projection of the volume does not cover the entire image, some thread may finish early and be in idle while others are busy with some hot-spots. A second disadvantage is that the DLLO has to be traversed multiple times for every thread, which increases the overall computational cost.

Therefore, we adopt the method of parallel composition in partial image level. The DLLO is traversed and partial images are accessed according to the current viewing direction. Each partial image is then divided into uniform regions and each region is composited into the final image with a thread. Early ray termination is considered during such process. Threads are synchronized every time before the start of next partial image composition. This method guarantees balanced load of computation tasks in each thread. Moreover, multiple threads working on the same partial image can benefit from the cache coherence. The DLLO is also traversed only once during the whole composition phase.

Suppose that the total execution time of the serial algorithm is T_s and the time of DLLO construction, octant rendering and partial image composition is T_1, T_2 and T_3 respectively. Then we have $T_s = T_1 + T_2 + T_3$. In the parallel algorithm, since the very first octant is updated to the DLLO (takes time T_ε), octant rendering and DLLO construction are running in parallel. Suppose there are p ($p > 1$) physical processors available, the phase of octant rendering can be completed in $T_2/(p - 1)$ time. After that, all the processors can be used for partial image composition, so this phase costs time T_3/p. Obviously, the total time cost in the parallel algorithm, $T_p = T_\varepsilon + T_2/(p - 1) + T_3/p$, could be far less than the time used by the serial algorithm ($T_p < T_s$) when p is large. Furthermore, when the benefit of the hyper threading technology is considered, there are $2p$ logical processors. In practice, the parallel algorithm can achieve even better performance since the total running time $T_p^{'}$ will be $T_p \geq T_p^{'} > T_\varepsilon + T_2/(2p - 1) + T_3/2p$.

6.4.2 Parallelization of Cluster-Based 4D Volume Rendering

The cluster-based time-varying volume rendering algorithm is designed to encode the 4D data with a relatively complicated method, and to decode the processed data with a simple method so as to ensure that it is computationally efficient. We divide the procedure of MVD rendering into three phases, namely *KeyBlock* reading, *KeyBlock* rendering and *KeyImage* composition. Compared to the DLLO-based algorithm, the decoding processes involved in the cluster-based algorithm are not as complicated. It avoids operations such as DLLO construction and tree traversal; on the contrary, it is capable to efficiently reconstruct the volume of each time step through a lookup table (Volume-KeyBlock table) that is just a 3D array. In the following, we analyze the parallelism of this algorithm by considering each individual phase.

Due to the nature of the hard disk, parallel reading of an MVD file will compromise the performance of the algorithm with the introduction of extra seek time, rotational latency and risk of conflicts. Hence, the task of *KeyBlock* reading is assigned to a single thread.

A *KeyBlock* pool is employed in the parallel algorithm to maintain the *KeyBlock*s used in rendering of either current or future volumes. During the initialization of each time step, the "expired" *KeyBlock*s are released and the *KeyBlock* reading thread is triggered. It sequentially loads the new *KeyBlock*s referred by the current volume into the *KeyBlock* pool

(Figure 6.6). Although the operations involved in this phase are running in serial, the thread is possible to be executed in parallel with the *KeyBlock* rendering phase.

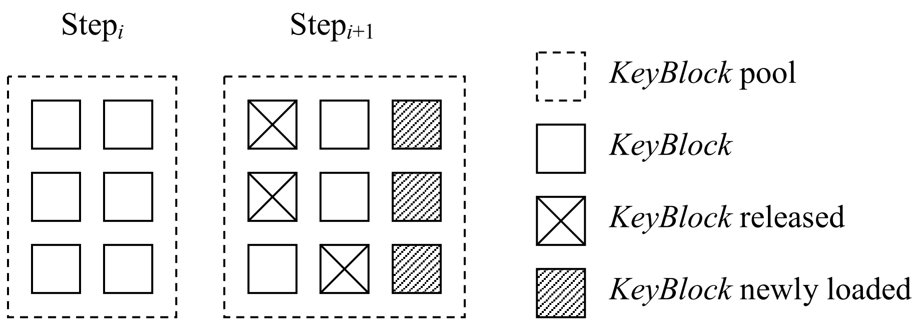


Figure 6.6 Management of the *KeyBlock* pool

Multiple threads are created for *KeyBlock* rendering. Each thread repeatedly requests from the *KeyBlock* pool and renders the allocated *KeyBlock* into the associated partial image buffer. Depending on whether the model-view transformation and transfer functions have been changed since the last time step, the set of returned *KeyBlock* varies. If the above conditions are changed, both existing and newly loaded *KeyBlock*s are returned to the thread; otherwise, only the newly loaded *KeyBlock*s are returned. The *KeyBlock*s passed to a thread will be marked and will not be rendered by another thread. The mechanism guarantees that parallel threads are running independent of one another without the risk of conflicts. All threads are only synchronized when all the *KeyBlock*s have been rendered.

There are two options for the parallel image composition. First, the final image is divided into uniform regions and each thread is responsible for one. Alternatively, we divide a *KeyImage* into uniform regions and each region is composited into the final image with a thread. Similar to the previous parallel DLLO-based rendering algorithm, the latter has advantages over the former for load balancing and cache coherence. This discussion is therefore not repeated here.

Threads of *KeyImage* composition are synchronized every time when all have finished execution. The cost of synchronization is low as the loads of threads are balanced. After the *KeyImage*s of all blocks in the current volume are processed, the final image is displayed to users.

When texture-mapping is employed as the underlying renderer, the parallel rendering algorithm will not include the phase of composition. The main tasks in the phase of *KeyBlock* rendering will also be changed. Threads of this phase are responsible for downloading *KeyBlock*s to the texture memory of the graphics hardware based on the Volume-KeyBlock table. The cluster-based 4D volume rendering algorithm is designed to have small data throughput by exploiting the global coherence of data. However, in the parallel algorithm, it is still possible to reach the texture download limit of the graphics hardware. The bottleneck of the parallel texture-mapped algorithm, therefore, could be the download bandwidth of the texture memory, although it increases quickly with the advancement of graphics processing unit (GPU) techniques. Anyway, the proposed parallel algorithm maximizes its performance by utilizing fully the capability of the graphics hardware.

Suppose that, in the serial algorithm, the time taken by *KeyBlock* reading, rendering and composition phases is T_1, T_2 and T_3 respectively. For the parallel algorithm that runs on a computer with p ($p > 1$) physical processors, the time taken by phase two can be reduced to

$T_2/(p-1)$. The simultaneous execution of phases one and two will require a time of $max(T_1, T_2/(p-1) + T_\varepsilon)$, where T_ε is the time for loading one *KeyBlock*. In the best case, the time taken by phase three can also be reduced to T_3/p with the parallel algorithm. The parallel algorithm is thus more efficient than the serial algorithm on a multiprocessor computer.

Chapter 7

Conclusion

7.1 Summary

With the development of medical radiology technology, the contemporary medical scanners not only can image the internal organs or structures of a human body in more and more details, but are also able to capture the dynamic activity of a human body during a period. Visualization of the new time-varying volume data is meaningful for radiologists and doctors for better diagnosis and treatment but it also poses a new challenge to the computer graphics technology. Compared to the steady state data, the size of the 4D data increases remarkably due to the introduction of time domain and it makes the traditional 3D renderers cannot perform well any more. New rendering methods specialized in 4D therefore, are demanded to be able to visualize the time-varying volume data more efficiently but not dependent of high cost hardware.

This thesis describes two new time-varying volume rendering algorithms. Both of them are characterized by decomposing the 4D volume data to take advantage of the data coherence. They are well designed to exploit the 4D features of the time-varying data while placing little restriction on the underlying 3D renderers. Thus, either of them has good extensibility and can be easily varied to fit for different applications. The two algorithms have been successfully used in our medical simulation systems to provide interactive and real-time 4D volume rendering on personal computers.

The first method achieves time-varying volume rendering through extending a spatial data structure with the consideration of time domain. The new data structure called dynamic linear level octree takes advantage of both spatial and temporal coherence of the time-varying data. An efficient algorithm was developed to dynamically keep the octree updated over the change of time steps. Representation of the 4D data in DLLO leads to the reduced space requirement and fast rendering. Because the DLLO conversion time is fairly short, this method is suitable to provide online preprocessing and instant rendering.

Since a time-varying volume dataset contains much more information than a static one, it will usually be viewed repeatedly and probably from multiple directions for enhanced comprehension. Therefore, it is worth taking more time in data preprocessing for better rendering performance. The second method is thus proposed based on this idea. The clustering technique is employed to investigate the 4D data more extensively where the global coherence is identified as a better characteristic of the 4D data that is exploited for performance improvement. The employment of a flexible volume decomposition scheme also results in further reduced space requirement. A sophisticated encoding algorithm leads to a task-relieved renderer as well as distinguished performance. Compared with the DLLO-based method, the rendering component of this method uses lookup tables instead of complicated data structures and their related operations such as tree traversal. This simplifies the rendering process and benefits the overall performance.

Performance tests show that both proposed methods are capable of providing interactive and real-time 4D volume rendering. The experiments studied the performance gains of the proposed algorithms in terms of space reduction and rendering acceleration. Although the two methods are not meant for data compression, they achieved very good compression ratios (up to 80%), and this in turn benefits their performance in rendering speed. Both of them could achieve significant acceleration as compared to conventional algorithms with reasonable error tolerance. The degradation of rendering quality is quantitatively and visually tolerable.

The possibility of parallelization of the two time-varying volume rendering algorithms is also discussed based on a multiprocessor personal computer by the employment of multi-threaded programming techniques towards delivering low cost simulation systems. Although parallel rendering is not the focus of the proposed algorithms, their capability in parallelism further extends their potential in performance improvement and the scope of applications.

Finally, I demonstrated the application of the proposed algorithms in simulating interventional radiology, microsurgery and the development of a virtual spine workstation. The time-varying volume rendering methods played an important role as the visualization component in these simulation systems.

The performance of the proposed time-varying volume rendering algorithms is dependent on the data coherence. 4D volume datasets containing large transparent regions or large homogeneous regions at successive time steps result in fast rendering speed. Both the DLLO and cluster based algorithms exhibit slower rendering if a 4D dataset is less coherent. However, the medical datasets tend to have high data coherence. According to the experimental results, even for the dataset with lowest data coherence, both algorithms achieved 2 to 5 times speedup which is quite substantial. Another limitation of the proposed algorithms is that their renderings are not lossless. However, the error tolerances employed in the algorithms effectively control the rendering quality according to the applications' need. Error analysis also shows that the degradation of the rendering quality is essentially small.

7.2 Future Work

There are several aspects of the proposed algorithms worth further investigation. It is interesting to investigate an automatic method for the selection of the optimal parameters and error tolerances in data conversion for the best rendering performance while the degradation of the rendering quality is within a user-specified tolerance. For the DLLO-based method, the error metrics can be improved by considering the classification information and transfer functions so that coherent regions can be identified more accurately. For the cluster-based method, through a more extensive performance comparison of different clustering algorithms, the best one can be employed to improve performance of the MVD encoding process in terms of both efficiency and accuracy. The implementation of the proposed parallelized algorithms in medical simulators on multiprocessor systems can further enhance the performance gains and their potential application scope.

References

ANAGNOSTOU, K., ATHERTON, T.J. AND WATERFALL, A.E. 2000, 4D Volume Rendering with the Shear Warp Factorisation, *Proceedings of the IEEE Symposium on Volume Visualization*, 129 – 137.

ANAGNOSTOU, K., ATHERTON, T.J. AND WATERFALL, A.E. 2001, 4D Volume Rendering with the Shear Warp Factorisation: Extensions and Quantitative Results, *Proceedings of Fifth International Conference on Information Visualisation*, 435 – 443.

ANDERSON, J., CHUI, C.K., CAI, Y.Y., WANG, Y.P., LI, Z.R., ENG, M., MA, X., NOWINSKI, W.L., SOLAIYAPPAN, M., MURPHY, K., GAILLOUD, P. AND VENBRUX, A. 2002, Virtual Reality Training in Interventional Radiology: the Johns Hopkins and Kent Ridge Digital Laboratory Experience, *Seminars in Interventional Radiology*, 19, 2, 179 – 185.

AVILA, R., SOBIERAJSKI, L.M. AND KAUFMAN, A.E. 1992, Towards a Comprehensive Volume Visualization System, *Proceedings of the Conference on Visualization*, 13 – 20.

BEESON, B., BARNES, D.G. AND BOURKE, P.D. 2003, A Distributed Data Implementation of the Perspective Shear-Warp Volume Rendering Algorithm for Visualisation of Large Astronomical Cubes, *Publications of the Astronomical Society of Australia*, 20, 3, 300 – 313.

BEHRENS, U. AND RATERING, R. 1998, Adding Shadows to a Texture-Based Volume Renderer, *Proceedings of Symposium on Volume Visualization*, 39 – 46.

BLINN, J.F. 1982, Light Reflection Functions for Simulation of Clouds and Dusty Surfaces, *Computer Graphics*, 16, 3, 21 – 29.

BOADA, I. AND NAVAZO, I. 2003, 3D Texture-Based Hybrid Visualizations, *Computers & Graphics*, 27, 1, 41 – 49.

CABRAL, B., CAM, N. AND FORAN, J. 1994, Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware, *ACM Symposium on Volume Visualization*, 91 – 98.

CHUI, C.K., YIN, Z.M., SHU, R.B. AND LOE, K.F. 1991, An Efficient Algorithm for Volume Display by Linear Level Octree, *Proceedings of Seminar on Computer Graphics DISCS/NUS*, 46 – 62.

CHUI, C.K., LI, Z., ANDERSON, J.H., MURRPHY, K., VENBRUX, A., MA, X., WANG, Z.L., GAILLOUD, P., CAI, Y., WANG, Y. AND NOWINSKI, W.L. 2002, Training and Planning of Interventional Neuroradiology Procedures - Initial Clinical Validation, *Studies in Health Technology and Informatics*, 85, 96 – 102.

CHUI, C.K., TEO, J., TEOH, S.H., ONG, S.H., WANG, Y., LI, J., WANG, Z.L., ANDERSON, J.H. AND NOWINSKI, W.L. 2002, A Finite Element Spine Model from VHD Male Data, *Proceedings of VHD Conference*.

COHEN, D. AND SHEFER, Z. 1993, *Proximity Clouds – an Acceleration Technique for 3D Grid Traversal*, Technical report FC 93-01, Ben Gurion University of the Negev.

DEMIRIS, A., MAYER, A. AND MEINZER, H.P. 1997, 3-D Visualization in Medicine: An Overview, *Contemporary Perspectives in Three-Dimensional Biomedical Imaging*, C.Roux and J.-L. Coatrieux (Eds.) IOS Press, 79 – 105.

DEV, P. 1999, Imaging and Visualization in Medical Education, *IEEE Computer Graphics and Applications*, 19, 3, 21 – 31.

DOBASHI, Y., VLATKO, C., KANEDA, K., YAMASHITA, H. AND NISHITA, T. 1998, A Fast Volume Rendering Method for Time-Varying 3-D Scalar Field Visualization Using Orthonormal Wavelets, *IEEE Transaction on Magnetics*, 3431 – 3434.

ELVINS, T.T. 1992, a Survey of Algorithms for Volume Visualization, *Computer Graphics*, 26, 3, 194 – 201.

ELLSWORTH, D., CHIANG, L.J. AND SHEN, H.W. 2000, Accelerating Time-Varying Hardware Volume Rendering Using TSP Trees and Color-Based Error Metrics, *Proceedings of the IEEE Symposium on Volume Visualization*, 119 – 128.

FREUND, J. AND SLOAN, K. 1997, Accelerated Volume Rendering Using Homogeneous Region Encoding, *Proceedings of the Conference on Visualization*, 191 – 196.

GELDER, A.V. AND KIM, K. 1996, Direct Volume Rendering With Shading via Three-Dimensional Textures, *Proceedings of Symposium on Volume Visualization*, 23 – 30.

GOLDSMITH, J. AND SALMON, J. 1987, Automatic Creation of Object Hierarchies for Ray Tracing, *IEEE Computer Graphics and Applications*, 7, 5, 14 – 20.

GOLDWASSER, S.M., REYNOLDS, R.A., TALTON, D.A. AND WALSH, E.S. 1989, High Performance Graphics Processors for Medical Imaging Applications, *Parallel Processing for Computer Vision and Display*, P.M. Dew, R.A. Earnshaw, and T.R. Heywood, Ed., Addison-Wesley, 461 – 470.

GUTHE, S. AND STRAßER, W. 2001, Real-Time Decompression and Visualization of Animated Volume Data, *Proceedings of the Conference on Visualization*, 349 – 356.

HANRAHAN, P. 1990, Three-Pass Affine Transforms for Volume Rendering, *Proceedings of SIGGRAPH'90*, 24, 71 – 78.

HATA, N., WADA, T., CHIBA, T., TSUTSUMI, Y., OKADA, Y. AND DOHI, T. 2003, 3D Volume Rendering of Fetal MR Images for the Diagnosis of Congenital Cystic Adenomatoid Malformation, *Academic Radiology*, 10, 3, 309 – 312.

HUA, W., CHUI, C.K., WANG, Y., WANG, Z.L., CHEN, X., PENG, Q. AND NOWINSKI, W.L. 2000, A Semiautomatic Framework for Vasculature Extraction from Volume Image, *Proceedings of International Conference on Biomedical Engineering*, 515 – 516.

KAJIYA, J.T. AND HERZEN, B.V. 1984, Ray Tracing Volume Densities, *Computer Graphics*, 18, 3, 165 – 174.

KAPLAN, M.R. 1985, Space-Tracing: A Constant Time Ray-Tracer, *Uses of Spatial Coherence in Ray-Tracer, SIGGRAPH*.

KAUFMAN, A.E., COHEN, D. AND YAGEL, R. 1993, Volumetric Graphics, *IEEE Computer Graphics*, 26, 7, 51 – 64.

KRUGER, J. AND WESTERMANN, R. 2003, Acceleration Techniques for GPU-Based Volume Rendering, *IEEE Visualization*, 287 – 292.

LACROUTE, P. AND LEVOY, M. 1994, Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation, *Proceedings of SINGGRAPH'94*, 451 – 458.

LACROUTE, P. 1995, Real-Time Volume Rendering on Shared Memory Multiprocessors Using the Shear-Warp Factorization, *Proceedings Parallel Rendering Symposium*, 15 – 22.

LACROUTE, P. 1996, Analysis of a Parallel Volume Rendering System Based on the Shear-Warp Factorization, *IEEE Transaction and Computer Graphics*, 2, 3, 218 – 231.

LAUR, D. AND HANRAHAN, P. 1991, Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering, *Computer Graphics*, 285 – 288.

LEE, C.H. AND PARK, K.H. 1997, Fast Volume Rendering Using Adaptive Block Subdivision, *Proceedings of Computer Graphics and Applications*, 221, 148 – 157.

LEVOY, M. 1988, Display of Surfaces from Volume Data, *IEEE Computer Graphics and Applications*, 8, 3, 29 – 37.

LEVOY, M. 1990, Volume Rendering, A Hybrid Ray Tracer for Rendering Polygon and Volume Data, *IEEE Computer graphics and Applications*, 10, 2, 33 – 40.

LEVOY, M. 1990, Efficient Ray Tracing of Volume Data, *ACM Transactions on Graphics*, 9, 3, 245 – 261.

LIAO, S.K., LIN, C.F., CHUNG, Y.C. AND LAI, J.Z.C. 2003, A Differential Volume Rendering Method with Second-Order-Difference for Time-Varying Volume Data, *International Journal of Visual Languages and Computing*, 14, 3, 233 – 254.

LIAO, S.K., LA, J.Z.C. AND CHUNG, Y.C. 2004, Time-Critical Rendering for Time-Varying Volume Data, *Computers & Graphics*, 28, 2, 279 – 288.

LICHTENBELT, B., CRANE, R. AND NAQVI, S. 1998, *Introduction to Volume Rendering*, Hewlett-Packard Company, Prentice Hall PTR.

LORENSEN, W.E. AND CLINE, H.E. 1987, Marching Cubes: A High Resolution 3D Surface Construction Algorithm, *Computer Graphics*, 21, 4, 163 – 169.

LUM, E.B., MA, K.L. AND CLYNE, J. 2001, Texture Hardware Assisted Rendering of Time-Varying Volume Data, *Proceedings of the Conference on Visualization*, 263 – 270.

MA, K.L., PAINTER, J.S., HANSEN, C.D. AND KROGH, M.F. 1994, Parallel Volume Rendering Using Binary Swap Image Composition, *IEEE Computer Graphics and Applications*, 14, 4, 59-68.

MA, K.L., SMITH, D., SHIH, M.Y. AND SHEN, H.W. 1998, *Efficient Encoding and Rendering of Time-Varying Volume Data*, Technical report, TR-98-22, ICASE, NASA Langsley Research Center.

MA, K.L., LUM, E.B. AND MURAKI, S. 2003, Recent Advances in Hardware-Accelerated Volume Rendering, *Computers & Graphics*, 27, 5, 725 – 734.

MEAGHER, D. 1982, Geometric Modeling Using Octree Encoding, *Computer Graphics Image Processing*, 19, 129 – 147.

MEISSNER, M., HOFFMANN, U. AND STRASSER, W. 1999, Enabling Classification and Shading for 3D Texture Mapping based Volume Rendering using OpenGL and Extensions, *Proceedings of IEEE Visualization*, 207 – 214.

MEISSNER, M., HUANG, J., BARTZ, D., MUELLER, K. AND CRAWFIS, R. 2000, A Practical Evaluation of Popular Volume Rendering Algorithms, *Proceeding Volume Visualization and Graphics Symposium*, 81 – 90.

MEISSNER, M., KANUS, U., WETEKAM, G., HIRCHE, J., EHLERT, A., STRASSER, W., DOGGETT, M., FORTHMANN, P. AND PROKSA, R. 2002, VIZARD II, a Reconfigurable Interactive Volume Rendering System, *Proceedings of the Eurographics Workshop on Graphics Hardware*, 137–146.

MUELLER, K., SHAREEF, N., HUANG, J. AND CRAWFIS, R. 1999, High-quality Splatting on Rectilinear Grids with Efficient Culling of Occluded Voxels, *IEEE Transaction on Visualization and Computer Graphics*, 116 – 134.

NEOPHYTOU, N. AND MUELLER, K. 2002, Space-Time Points: 4D Splatting on Efficient Grids, *Proceedings of the IEEE Symposium on Volume Visualization and Graphics*, 97 – 106.

NEUMANN, U. 1994, Communication Costs for Parallel Volume Rendering Algorithms, *IEEE Computer Graphics and Applications*, 14, 4, 49 – 58.

NIEH, J. AND LEVOY, M. 1992, Volume Rendering on Scalable Shared-Memory MIMD Architecture, *Proceedings of Workshop on Volume Visualization*, 17 – 24.

PALMER, M.E., TAYLOR, S. AND TOTTY, B. 1997, Exploiting Deep Parallel Memory Hierarchies for Ray Casting Volume Rendering, *Proceedings of the Parallel Rendering Symposium*, 15 – 22.

PAWASAUSKAS, J. 1997, Volume Visualization with Ray Casting, *http://www.cs.wpi.edu/~matt/courses/cs563/talks/powwie/p1/ray-cast.htm*, WPI, SC.

PFISTER, H., HARDENBERGH, J., KNITTEL, J., LAUER, H. AND SEILER, L. 1999, The VolumePro Real-time Ray-casting System, *Proceedings of SIGGRAPH'99*, 251 – 260.

PHONG, B.T. 1975, Illumination for Computer Generated Pictures, *Communications of the ACM*, 18, 6, 311 – 317.

PLESNIAK, W., HALLE, M., PIEPER, S.D., WELLS, W., JAKAB, M., MEIER, D.S., BENTON, S.A., GUTTMANN, R.G. AND KIKINIS, R. 2003, Holographic Video Display of Time-Series Volumetric Medical Data, *IEEE Visualization*, 589 – 596.

PORTER, T. AND DUFF, T. 1984, Compositing Digital Images, *Computer Graphics*, 18, 3, 253 – 259.

RA, J.B., KWON, S.M., KIM, J.K., YI, J., KIM, K.H., H. PARK, W., KYUNG, K.U., KWON, D.S. et al. 2002, Spine Needle Biopsy Simulator Using Visual and Force Feedback, *Computer Aided Surgery*, 7, 6, 353 – 363.

RAMAKRISHNAN, R. AND GEHRKE, J. 2000, *Database Management Systems*, 2nd edition, McGraw Hill.

RIEDERER, S.J. 2000, Current Technical Development of Magnetic Resonance Imaging, *IEEE Engineering in Medicine and Biology Magazine*, 19, 5, 34 – 41.

ROBB, R.A. 1995, *Three-Dimensional Biomedical Imaging: Principles and Practice*, VCH Publishers, Inc.

SALISBURY, J. AND KENNETH, JR. 1998, The Heart of Microsurgery, *Mechanical Engineering*, 46 – 51.

SAMET, H. AND WEBBER, R.E. 1988, Hierarchical Data Structures and Algorithms for Computer Graphics I Fundamentals, *IEEE Computer Graphics and Applications*, 8, 3, 48 –68.

SCHROEDER, W., MARTIN, K. AND LORENSEN, B. 1998, *The Visualization Toolkit*, 2nd edition, Prentice Hall PTR.

SHEN, H.W. AND JOHNSON, C.R. 1994, Differential Volume Rendering: a Fast Volume Visualization Technique for Flow Animation, *Proceedings of the Visualization Conference*, 180–187.

SHEN, H.W., CHIANG, L.J. AND MA, K.L. 1999, A Fast Volume Rendering Algorithm for Time-Varying Fields Using a Time-Space Partitioning (TSP) Tree, *Proceedings of the Conference on Visualization*, 371–377.

SOBIERAJSKI, L., COHEN, D., KAUFMAN, A.E., YAGEL, R. AND ACKER, D. 1993, A Fast Display Method for Volumetric Data, *The Visual Computer*, 116 – 124.

SOHN, B.S., BAJAJ, C. AND SIDDAVANAHALLI, V. 2002, Feature Based Volumetric Video Compression for Interactive Playback, *Proceedings of the IEEE Symposium on Volume Visualization and Graphics*, 89 – 96.

SOHN, B.S., BAJAJ, C. AND SIDDAVANAHALLI, V. 2004, Volumetric Video Compression and Interactive Playback, *Computer Vision and Image Understanding*, 96, 3, 435 – 452.

SRAMEK, M. 1994, Fast Surface Rendering from Raster Data by Voxel Traversal Using Chessboard Distance, *Proceedings of the Conference on Visualization*, 188 – 195.

TEOH, S.H. 2005, *Virtual Spine Workstation*, Technical Report, R-265-000-184-305, National University of Singapore.

TORY, M., RÖBER, N., MÖLLER, T., CELLER, A. AND ATKINS, M.S. 2001, 4D Space-Time Techniques: A Medical Imaging Case Study, *Proceedings of IEEE Visualization*, 473 – 476.

TUY, H.K. AND TUY, L.T. 1984, Direct 2-D Display of 3-D Objects, *IEEE computer Graphics and Applications*, 4, 29 – 33.

UPSON, C. AND KEELER, M. 1990, The V-Buffer: Visible Volume Rendering, *Computer Graphics*, 22, 4, 59 – 64.

WANG, Z.L., MA, X., ANG, M.H. JR., CHUI, C.K., ANG, C.H. AND NOWINSKI, W.L. 2001, A Virtual Environment-Based Practical Surgery System, *Asian Conference on Robotics and its Applications*, 69 – 73.

WANG, Z.L., CHUI, C.K., CAI, Y., ANG, C.H. AND NOWINSKI, W.L. 2002, Fast PC-based Visualization Algorithms for Virtual Reality Simulation of Microsurgical Procedures, *Proceedings of International Conference on Biomedical Engineering (ICBME)*.

WANG, Z.L., CHUI, C.K., ANG, C.H., LI, Z. AND NOWINSKI, W.L. 2002, Shear-Warp Volume Rendering Algorithm using Linear Level Octree for PC-based Medical Simulation, *Proceedings of International Conference on Medical Imaging Computing and Computer Assisted Intervention (MICCAI)*, LNCS, 2489, 2, 606 – 614.

WANG, Z.L., CHUI, C.K., CAI, Y.Y. AND ANG, C.H. 2004, Multidimensional Volume Visualization for PC-Based Microsurgical Simulation System, *Proceedings of ACM SIGGRAPH International Conference on Virtual Reality Continuum and its Applications in Industry (VRCAI)*, 309 – 316.

WANG, Z.L., CHUI, C.K., CAI, Y.Y., ANG, C.H. AND TEOH, S.H. 2005, Dynamic Linear Level Octree-Based Volume Rendering Methods for Interactive Microsurgical Simulation, to appear *International Journal of Image and Graphics*.

WANG, Z.L., TEO, J.C.M., CHUI, C.K., ONG, S.H., YAN, C.H., WANG, S.C., WONG, H.K. AND TEOH, S.H. 2005, Computational Biomechanical Modeling of the Lumbar Spine Using Marching-Cubes Surface Smoothened Finite Element Voxel Meshing, *Computer Methods and Programs in Biomedicine*, 80, 1, 25 – 35.

WESTOVER, L. 1989, Interactive Volume Rendering, *Proceedings of the Chapel Hill Workshop on Volume Visualization*, 9 – 16.

WESTOVER, L. 1990, Footprint Evaluation for Volume Rendering, *ACM SIGGRAPH Computer Graphics*, 24, 4, 367 – 376.

WESTOVER, L. 1991, *Splatting - A Parallel, Feed-Forward Volume Rendering Algorithm*, PhD thesis, University of North Carolina.

WHANG, K.Y., SONG, J.W., CHANG, J.W., KIM, J.Y., CHO, W.S., PARK, C.M. AND SONG, I.Y. 1995, Octree-R: An Adaptive Octree for Efficient Ray Tracing, *IEEE Transactions on Visualization and Computer Graphics*, 1, 4, 343 –349.

WILFRED, C.G.P., LOUIS, A.G. AND DALLAS D.P. 2002, Percutaneous Vertebroplasty for Severe Osteoporotic Vertebral Body Compression Fractures, *Radiology*, 223, 121 – 126.

WILSON, B., MA, K.L. AND MCCORMICK, P.S. 2002, A Hardware-Assisted Hybrid Rendering Technique for Interactive Volume Visualization, *Proceedings of the IEEE Symposium on Volume Visualization and Graphics*, 123 – 130.

WOODRING, J. AND SHEN, H.W. 2003, Chronovolumes: A Direct Rendering Technique for Visualizing Time-Varying Data, *Proceedings of the Eurographics/IEEE TVCG Workshop on Volume Graphics*, 27 – 34.

WU, Y., BHATIA, V., LAUER, H. AND SEILER, L. 2003, Shear-Image Order Ray Casting Volume Rendering, *Proceedings of the symposium on Interactive 3D graphics*, 152 – 162.

YAGEL, R. 1999, Efficient Techniques for Volume Rendering of Scalar Fields, *Data Visualization Techniques*, Edited by C. Bajaj, Chichester: Wiley.

YAGEL, R. AND MACHIRAJU, R. 1995, Data Parallel Volume Rendering Algorithms, *The Visual Computer*, 11, 6, 319 – 338.

ZHANG, T., RAMAKRISHNAN, R. AND LIVNY, M. 1996, BIRCH: An Efficient Data Clustering Method for Very Large Databases. *Proceedings of ACM International Conference on Management of Data*, 103 – 114.

Appendix A

Space and Time Complexity of Linear Level Octree

A.1 Space Savings of Linear Level Octree

The results shown in Chapter 3 demonstrate the superiority of linear level octree (LLO) in space savings as compared to the raw volume data. In this section, we analyze the space efficiency of LLO over linear octree (LO).

Let m_{LO} be the number of bits required by a black node in LO, then

$$m_{LO} = 3n + (1 + [\log_2 n]) \qquad \textbf{(A.1)}$$

where n is the resolution.

The location code of LLO can be implemented using a variable number of bits. Let m_{LLO} be the number of bits required by a L-th level node in LLO, then

$$m_{LLO(L)} = 3L + (1 + [\log_2 L]) \tag{A.2}$$

The table below compares the space requirement of the LLO nodes and LO nodes.

Table A.1 Comparison of space usage of LLO and LO (*n* = 10)

Level of node (*L*)		1	2	3	4	5	6	7	8	9	10
LLO	**No. of Bits**	4	8	11	15	18	21	24	28	31	34
	No. of Bytes	1	1	2	2	3	3	3	4	4	5
LO	**No. of Bits**	34	34	34	34	34	34	34	34	34	34
	No. of Bytes	5	5	5	5	5	5	5	5	5	5

Let $R_{(LLO/LO)}$ be the reduction ratio in space required by LLO versus LO, M_{LLO} and M_{LO} be the number of bits required by all nodes in LLO and LO respectively, then

$$M_{LLO} = \sum_{L=1}^{n} (K_L \cdot m_{LLO(L)}) = \sum_{L=1}^{n} (K_L \cdot (3L + (1 + [\log_2 L]))) \tag{A.3}$$

$$M_{LO} = m_{LO} \cdot N = (3n + (1 + [\log_2 n])) \cdot N \tag{A.4}$$

where K_L is the number of nodes in L-th level, N is the number of nodes in LO. The space reduction ratio is:

$$\begin{aligned} & R_{(LLO/LO)} \\ & = (M_{LO} - M_{LLO}) / M_{LO} \\ & = \frac{((3n + (1 + [\log_2 n])) \cdot N) - \sum_{L=1}^{n} (K_L \cdot (3L + (1 + [\log_2 L])))}{(3n + (1 + [\log_2 n])) \cdot N} \end{aligned} \quad \textbf{(A.5)}$$

Based on the experimental results reported in [Chui et al. 1991], a reduction in space requirement ranging from 22 – 42% can be achieved. The memory space required using LLO is much less than that of LO for the same object.

A.2 Complexity Analysis of the LLO Generation Algorithm

The time complexity of the algorithm of LLO generation is dependent on the number of nodes. According to the leaf node criteria given in Chapter 3, the size of the smallest octant is 2^l and the number of levels will not exceed ($n – l$ + 1). Therefore, there are $(2^{n-l})^3 = 8^{n-l}$ leaf octants initially for a volume with size 2^n. The basic operation of the algorithm is to merge eight adjacent smaller octants into a bigger octant. Suppose the processing time of each operation is identical, say T_0. Then the time taken for merging eight adjacent octants at level ($n – l$) to one node at ($n – l$ – 1) is:

$$T_l = T_0 \cdot \frac{8^{n-l}}{8} = T_0 \cdot 8^{n-l-1} \quad \textbf{(A.6)}$$

As the merge operation propagates up level by level, the time is reduced by a factor of 8 in each level. Hence the total time of LLO generation can be expressed as follows:

$$\begin{aligned} T &= \sum_{i=l}^{n-1} T_i = T_0 \cdot 8^{n-l-1} \cdot (1 + \frac{1}{8} + \frac{1}{8^2} + \cdots + \frac{1}{8^{n-l-1}}) \\ &= T_0 \cdot \frac{8^{n-l}}{7} \cdot (1 - \frac{1}{8^{n-l}}) \\ &< \frac{T_0}{7} \cdot 8^{n-l} \end{aligned} \qquad \textbf{(A.7)}$$

Therefore, the time complexity of the LLO generation algorithm is $O(8^{n-l})$.

Appendix B

LLO-based Multimodality Volume Rendering

B.1 Introduction

Multimodality volume rendering is an important branch of volume visualization providing additional valuable insights of medical images. With the development of medical image acquisition techniques, many modalities of medical imaging are available and they are good at presenting different tissues or structures of human body. It is desirable to integrate important characteristics of multiple volume datasets acquired from the same anatomy into a single visual representation to get more comprehensive information about the interested structures.

Octrees are efficient in set-theoretic operations. If multiple datasets have been encoded in linear level octree (LLO) representations, they can be easily fused by simple octree "OR" operations. In particular, since the empty regions are already excluded from the encoding of

LLO, many redundant operations required in traditional volume integration such as to merge a sub-volume with an empty space can be effectively avoided. The integration of multiple LLOs, therefore, can be very fast. Additionally, due to the small size of LLOs as compared to the volume data, the employment of LLO in rendering also reduces the memory requirement.

The LLO-based volume rendering algorithm is, therefore, extended to support rendering multimodality volume images. An interactive visualization of multiple modalities can be achieved through fast online LLO integration.

B.2 Method

Before rendering, it is necessary for multiple datasets to go through registration[18] and re-sampling to ensure that the datasets are aligned properly as well as having the same dimensions. Then, they are converted to LLOs.

Multiple modalities could be integrated at the data pre-processing stage, the rendering stage or the composition stage. An integration criterion, referred to as integration function, need to be defined. Since the online integration is used in our solution, an integration function at the rendering stage can be defined as:

$$S_I = \alpha \cdot T_A(S_A) \cdot S_A + \beta \cdot T_B(S_B) \cdot S_B \qquad \textbf{(B.1)}$$

where:

[18] Registration is the process of transforming the different sets of data into one coordinate system.

S_A and S_B are samples from volume datasets A and B respectively at the same location;

S_I is the integrated sample value;

T_A and T_B are the opacity transfer functions of volumes A and B respectively. Their values are dependent on the sample values;

α and β are the integration factors.

A lookup table of the integration factors as shown in Table B.1 is employed for efficient multimodality integration.

Table B.1 Integration factor lookup table

Intensity value	$\boldsymbol{\alpha}$	$\boldsymbol{\beta}$
1	0.6	0.4
2	0.2	0.8
…	…	…

The boundary condition, $\alpha + \beta \leq 1$, is applied. It guarantees the value of the integrated sample will not be overflowed. Similar to the opacity transfer functions, the integration factors are also dependent on the sample values. Samples of different intensities are weighted differently so as to provide flexible multimodality rendering.

As shown in Equation B.1, the integration factors are working together with the opacity transfer functions. Firstly, an opacity transfer function of each volume dataset determines the intensity ranges of samples that are to be seen and how transparent they appear. The integration factors then determines how samples from the each volume can contribute to the final image. This mechanism makes it possible that structures from different modalities can be interactively and selectively visualized together without confusion.

Appendix C

Error Metrics Computation in DLLO

C.1 Introduction

We take advantage of the spatial and temporal coherence of the time-varying volume data through dynamic linear level octree to accelerate the rendering speed and reduce the space and I/O requirement, where octant variance is used to measure the spatial coherence and the normalized Euclidean distance (NED) is used to measure the temporal coherence. The computation of octant variance and NED could be expensive. However, in some cases, it is not necessary to calculate these two error metrics from scratch by accessing all of the voxel values repeatedly. In this appendix, we will derive the formulas for efficient evaluation of the error metrics. These formulas have been given in the text without derivation.

C.2 General Variance Computation

Assume D_1 and D_2 are two sets of points, where D_1 includes n_1 points with mean of μ_1 and variance of $\sigma^2{}_1$ and D_2 includes n_2 points with mean of μ_2 and variance of $\sigma^2{}_2$. We deduce the equations for the computation of the mean μ and variance σ^2 of set $D = D_1 \cup D_2$ as follows.

Obviously, the mean of set *D* can be easily computed as Equation C.1.

$$\mu = \frac{n_1\mu_1 + n_2\mu_2}{n_1 + n_2} \tag{C.1}$$

Based on the definition of variance, the variance of set *D* can be computed as Equation 3.3.

$$\sigma^2 = \frac{1}{n_1 + n_2}\sum_{i=1}^{2}\sum_{x\in D_i}(x-\mu)^2 \tag{C.2}$$

where

$$\begin{aligned}
&\sum_{i=1}^{2}\sum_{x\in D_i}(x-\mu)^2 \\
&=\sum_{i=1}^{2}\sum_{x\in D_i}(x-\mu_i+\mu_i-\mu)^2 \\
&=\sum_{i=1}^{2}\sum_{x\in D_i}(x-\mu_i)^2+\sum_{i=1}^{2}\sum_{x\in D_i}(\mu_i-\mu)^2+2\sum_{i=1}^{2}\sum_{x\in D_i}(x-\mu_i)(\mu_i-\mu)
\end{aligned} \tag{C.3}$$

and

$$\begin{aligned}\sum_{i=1}^{2}\sum_{x\in D_i}(x-\mu_i)(\mu_i-\mu) &= \sum_{i=1}^{2}(\mu_i-\mu)\sum_{x\in D_i}(x-\mu_i)\\ &= \sum_{i=1}^{2}(\mu_i-\mu)\left(\sum_{x\in D_i}x-\sum_{x\in D_i}\mu_i\right)\\ &= \sum_{i=1}^{2}(\mu_i-\mu)\left(n_i\mu_i-n_i\mu_i\right)\\ &= 0\end{aligned} \tag{C.4}$$

Together with Equations 3.3, C.3 and C.4, we get the equation for the computation of the variance of set *D*:

$$\begin{aligned}\sigma^2 &= \frac{1}{n_1+n_2}\left[\sum_{i=1}^{2}\sum_{x\in D_i}(x-\mu_i)^2+\sum_{i=1}^{2}\sum_{x\in D_i}(\mu_i-\mu)^2\right]\\ &= \frac{1}{n_1+n_2}\left[n_1\sigma^2{}_1+n_2\sigma^2{}_2+n_1(\mu_1-\mu)^2+n_2(\mu_2-\mu)^2\right]\end{aligned} \tag{C.5}$$

This method can be extended for the computation of mean and variance of a set united from more than two sets. Assume D_i is one of $N \geq 2$ sets of points, where D_i includes n_i points with mean of μ_i and variance of $\sigma^2{}_i$. The mean and variance of set $D=\bigcup_{i=1}^{N}D_i$ can be computed based on Equations C.6 and C.7 below, respectively.

$$\mu = \frac{\sum_{i=1}^{N}(n_i\mu_i)}{\sum_{i=1}^{N}n_i} \tag{C.6}$$

$$\sigma^2 = \frac{\sum_{i=1}^{N}\left[n_i\sigma^2{}_i+n_i(\mu_i-\mu)^2\right]}{\sum_{i=1}^{N}n_i} \tag{C.7}$$

C.3 Octant Variance Computation

In the algorithm of LLO/DLLO generation, when we try to merge eight octants into a bigger octant at a higher level, the variance of the parent octant needs to be computed and compared with a variance threshold (spatial error tolerance) to determine if the merged octant satisfies the leaf node criterion. Let us assume the mean and variance of the eight child octants as μ_i and $\sigma^2{}_i$, where i is an integer from 1 to 8 for each octant. Since the eight child octants are at the same octree level, they contain the same number of voxels, say n_0. According to Equations C.6 and C.7, we can compute the mean and variance of their parent octant as follows:

$$\mu = \frac{\sum_{i=1}^{8} n_i \mu_0}{8n_0} = \frac{1}{8}\sum_{i=1}^{8}\mu_i \tag{C.8}$$

$$\sigma^2 = \frac{\sum_{i=1}^{8}\left[n_0\sigma^2{}_i + n_0(\mu_i - \mu)^2\right]}{8n_0} = \frac{1}{8}\sum_{i=1}^{8}\left[\sigma^2{}_i + (\mu_i - \mu)^2\right] \tag{C.9}$$

In Equation C.9, we avoid the access of voxel values, and the variance of a parent octant can be computed efficiently based on the mean and variance values of its eight child octants. These two equations are essentially the formulas we used in Chapter 3.

C.4 Computation of the Normalized Euclidean Distance between Octants

In the generation of a DLLO, the normalized Euclidean distance is used to evaluate the similarity between the corresponding leaf octants from two time successive LLOs. Let us assume there are two such leaf octants A and B. The means and variances of the two octants are represented by μ_A and $\sigma^2{}_A$ and μ_B and $\sigma^2{}_B$ respectively, and the two octants contain n voxels each. Based on the definition, the square of the Euclidean distance between them should be computed as:

$$\begin{aligned} D^2 &= \sum_i (a_i - b_i)^2 \\ &= \sum_i a_i^2 + \sum_i b_i^2 - 2\sum_i a_i b_i \end{aligned} \tag{C.10}$$

where, a_i and b_i are the ith voxel values of octants A and B respectively.

We know that the variance of octant A is defined as:

$$\begin{aligned} \sigma_A^2 &= \frac{\sum_i (a_i - \mu_A)^2}{n} \\ &= \frac{\sum_i a_i^2 + \sum_i \mu_A^2 - 2\sum_i a_i \mu_A}{n} \\ &= \frac{\sum_i a_i^2 + n\mu_A^2 - 2n\mu_A^2}{n} \\ &= \frac{\sum_i a_i^2 - n\mu_A^2}{n} \end{aligned} \tag{C.11}$$

Thus, we get

$$\sum_i a_i^2 = n\sigma_A^2 + n\mu_A^2 \tag{C.12}$$

Similarly, we can also derive Equation C.13 from the definition of the variance computation of octant B.

$$\sum_i b_i^2 = n\sigma_B^2 + n\mu_B^2 \tag{C.13}$$

Together with Equations C.12 and C.13, Equation C.10 can be rewritten as:

$$D^2 = n\sigma_A^2 + n\mu_A^2 + n\sigma_B^2 + n\mu_B^2 - 2\sum_i a_i b_i \tag{C.14}$$

If the variance of octant *A* is small ($\sigma_A^2 \le \nu$), we approximate the computation of the square Euclidean distance between octants *A* and *B* by substituting a_i with μ_A in Equation C.14. Thus we get Equation C.15.

$$\begin{aligned} D^2 &= n\sigma_A^2 + n\mu_A^2 + n\sigma_B^2 + n\mu_B^2 - 2\sum_i \mu_A b_i \\ &= n\sigma_A^2 + n\mu_A^2 + n\sigma_B^2 + n\mu_B^2 - 2n\mu_A\mu_B \\ &= n\left[\sigma_A^2 + \sigma_B^2 + (\mu_A - \mu_B)^2\right] \end{aligned} \tag{C.15}$$

If the variance of octant *B* is small ($\sigma_B^2 \le \nu$), we approximate the computation of the square Euclidean distance between octants *A* and *B* by substituting b_i with μ_B in Equation C.14. We also obtain the same Equation C.15.

Thus the normalized Euclidean distance between octant *A* and *B*, where either octant *A* or octant *B* has homogeneous voxel values, can be estimated efficiently with Equation 3.7.

$$NED = \frac{\sqrt{D^2}}{n} = \sqrt{\frac{\sigma_A^2 + \sigma_B^2 + (\mu_A - \mu_B)^2}{n}} \tag{C.16}$$

This is essentially the same formula we used in Chapter 3.