University of Alberta

A System for Real-Time Rendering of Compressed Time-Varying Volume Data

by

Biao She

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Spring 2011
Edmonton, Alberta

Examining Committee

Pierre Boulanger, Computing Science

Michelle Noga, Radiology & Diagnostic Imaging

Nilanjan Ray, Computing Science

To my family

For all your unconditional support.

Abstract

Real-time rendering of static volumetric data is generally known to be a memory and computationally intensive process. With the advance of graphic hardware, especially GPU, it is now possible to do this using desktop computers. However, with the evolution of real-time CT and MRI technologies, volumetric rendering is an even bigger challenge. The first one is how to reduce the data transmission between the main memory and the graphic memory. The second one is how to efficiently take advantage of the time redundancy which exists in the time-varying volumetric data. Most previous researches either focus on one problem or the other. In this thesis, we implemented a system which efficiently deals with both of the challenges.

We proposed an optimized compression scheme that explores the time redundancy as well as space redundancy of time-varying volumetric data. The compressed data is then transmitted to graphic memory and directly rendered by GPU, so the data transfer between main memory and graphic memory is significantly reduced. With our implemented system, we successfully reduce more than half of the time of transferring the whole data directly. We also compare our proposed compression scheme with the one without exploiting time redundancy. The optimized compression scheme shows a reduce compression distortion over time. With usability, portability and extensibility in mind, the implemented system is also quite flexible.

Acknowledgements

First of all, I would like to thank my supervisor Dr. Pierre Boulanger for his patient guidance and support throughout my thesis research. Secondly, I would like to thank Dr. Michelle Noga for providing the volumetric medical data and valuable feedbacks. Also, I would like to thank Dr. Nilanjan Ray for his time on reading and commenting this work. Special thanks to Rui Shen and Matthew Hamilton for their help on VTK/ITK/CMAKE. Special thanks to Dr. Jens Schneider for providing the benchmark system. Special thanks to Xida Chen for useful feedback on various topics.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Coronary heart disease is the single largest killer of people in Canada. Every 7 minutes, someone in Canada dies from heart disease or stroke. The average annual mortality rate for congestive heart failure in Canada is 10%, with a 50% five-year survival rate [57]. A study, which examined hospital discharges in the year 2000, found that a total of 1.38 million hospital days were associated with congestive heart failure. Approximately 15% of patients died while in hospital, and the average hospital stay was slightly less than 13 days. These statistics highlight the strategic importance of being able to diagnose this disease and to reduce the cost of health care with early diagnosis using 3-D imaging technologies. The ability to image the heart in real-time, to visualize it, and to plan for treatments is key to reducing the cost and to improving treatments. All the heart disease treatments involve the establishment of critical competency assessment and require planning, rehearsal and predictive virtual tools. One advantage of high-resolution visualization is the possibility of virtual surgical training and planning. One study by Gallagher et al. [16] showed a six-fold improvement in avoiding vital structures after training with a surgical simulator. For accurate training, advanced visualization of the heart is key, given that it is a dynamic organ. Integration of technologies opens new possibilities to improve competency and move away from the hands-on experience with live patients - the so called Halstedian "see one, do one, teach one" method that has so long underpinned many surgical training approaches.

The processing speed of today's Central Processing Units (CPU) is not sufficient to achieve interactive visualization of real-time volumetric data. Fortunately,

the recent development of high-speed Graphics Processing Units (GPU), capable of processing data at a rate of 1 Tera-flops per card, is changing the landscape of today's computing power. Often a surface-based approach is used to render these data sets, but with the development of new graphics hardware, it is now possible to visualize, in real-time, volumetric data using texture-mapping or ray-casting techniques which are visually more accurate and do not require segmentation. This is a significant improvement because intra-anatomical structures cannot be visualized using surface techniques and many important details are lost. Volumetric visualization techniques are computationally much more expensive, making high-quality visualization of real-time dynamic cardiac data a challenge for current 3D rendering algorithms. Using a combination of tightly coupled scalar computing elements CPUs with a Graphics Processing Unit (GPU), the goal of this thesis is to develop GPU based algorithm capable of displaying large temporal volumetric data sets. As demonstrated in [15], the GPU contains multiple graphics processors that work in parallel, allowing the rendering of volumetric data set with high image quality in real-time. Using new volume texture binding techniques, one can dynamically bind volume data to the 3D rendering engine without degrading performance. The main problem with dynamic data is the high-bandwidth communication between the central memory and the GPU. The use of high-performance computers, such as the Altix 4700, connected by a high-speed data bus (60 Gb/s), such as NUMA II, solves these problems only partially because the connection between CPU and GPU is still limited by a relatively low-speed PCI-e bus. In order to solve this problem, we present a novel algorithm to decompress the volumetric data within GPU in real-time, hence reducing the traffic on PCI-e, to match the demands for real-time display and collaboration. This includes the development of techniques similar to video MPEG encoding, but for temporal volumetric data, as in [25].

1.1 Time-varying Volumetric Data

Volumetric data is typically a set of material property samples at different locations in 3D space. The basic element of volumetric data can be generalized as (x, y, z, w),

where (x, y, z) denotes the location in space and w denotes some material properties at this location. The value w can be a scalar (CT, MRI), vectorial (DTI), or multi-valued properties obtained by sensors, simulation, or modeling techniques. One good example of volumetric data is data obtained from Magnetic Resonance Imaging (MRI) or Computed Tomography (CT). In this specific case the (x, y, z) is the location of the sample in space and w is a scalar representing X-ray absorbtion in Hounsfield units in the case of CT or spin decay time for MRI. A more detailed introduction to volumetric data analysis can be found in [32]. In theory, the sampling location (x, y, z) may be taken without any restrictions in a 3D volume; however, in general, medical volumetric data is often sampled space at regularly spaced intervals along xyz axes creating blocks of data called voxels.

As mentioned previously the size of volumetric data is large, which makes visualization of volumetric data computationally challenging. To get a better sense of its size, let us assume that we have a volumetric data where w is a scalar value between 0 to 255, and sampled at 1024^3. The corresponding representation of the volumetric data in computer main memory is an array of 1024^3 elements of 8-bit unsigned characters, which occupies 1GB of memory space. In many applications this is not a very high resolution volumetric data and it already exceeds the video memory capacity of most GPUs which is responsible for rendering this data in real-time.

Time-varying volumetric data are even bigger in size. It is essentially a series of stationary volumetric data at different time steps. Theoretically, the size of time-varying volumetric data has no upper bound as it could contain an infinite amount of time steps if we digitize a moving organ for a long-period of time. Although it is limited by the data acquisition hardware, the time interval between two continuous time steps of a typical time-varying volumetric data should be very short. Otherwise, some important information of the moving organ will be lost. The data similarity between different time steps is called *time redundancy*.

The visualization of time-varying volumetric data is very difficult due to its unlimited large data size. Visualizing this data helps end users to perceive the change and understands the behavior of an organ like the heart in time and space, which

stationary volumetric data can never provide. Numerous systems have focused on visualizing stationary volumetric data in the past. Unlike these systems, the visualization of time-varying data set should have its own techniques which take advantage of the unique features of time-varying volumetric data.

1.2 Volumetric Data Rendering

Volumetric data rendering has been studied by researchers for decades. Generally, it has several advantages over traditional polygon rendering techniques such as: viewpoint independent and insensitivity to scene complexity. Its disadvantage is that it needs large storage space and its rendering require to solve complex equations which are very computationally intensive. Volume rendering allow both interior and exterior information to be displayed by using complex density color look-up functions and transparency.

The term real-time rendering means that the rendering speed (typically 10 to 30 Hz) should be sufficient to allow interactive feedback to end users allowing then to zoom, change viewpoint, and move around the data set [3]. Many sophisticated speed-up techniques have been proposed and tested in the past. Splating, shear-wrap, texture mapping and ray casting are the most notable techniques and will be reviewed in the following chapter. In essence, many real-time volumetric rendering algorithms are striving to solve two issues. One is to deal with the heavy computational load of the rendering algorithm and the other one is how to deal with the memory limitation imposed by hardware.

The first issue results from the need to solve the volumetric rendering integral. The emission-absorption model is the most widely used optical model in volume rendering. The integral is the following [27]:

$$I(D) = I_0 e^{-\int_{s_0}^{D} k(t)\,dt} + \int_{s_0}^{D} q(s) e^{-\int_{s}^{D} k(t)\,dt}\,ds$$

In order to be able to solve this integral numerically, a Riemann sum approximation is used to transform the integral to this discrete approximation:

$$I(D) \approx I_0 \prod_{i=0}^{n} e^{-k(s_0+i*\Delta t)\Delta t} + \sum_{i=1}^{n} q(s_0 + i*\Delta t) \prod_{j=i+1}^{n} e^{-k(s_0+j*\Delta t)\Delta t},$$

The real-time solution of this integral on a commodity CPU is truly a challenging task. The situation is even more complex if global illumination is used in the rendering. The advancement of GPU computational power significantly change the situation. A state-of-art NVIDIA Quadro FX5800 can compute at a rate of 933.12 Giga Floating Point Operations Per Second (FLOPS), which is more than enough for most volume rendering algorithm. At present, it is possible to interactively render a gigabyte-scale volumetric data with reasonable fidelity on commodity graphics hardware using a GPU [51] [25].

The second issue is caused by the huge size of time-varying volumetric data. Most real-time rendering algorithms require the volumetric data to be stored before hand on the GPU video memory as textures. When the size of volumetric data exceeds the capacity of the video memory, data exchange between video memory and other storage devices such as main memory and hard drive, is unavoidable reducing significantly the rendering speed as the data transmission between central memory and the GPU memory is usually not fast enough. The computational units in the GPU needs to wait for the data transmission to be done and then compute the rendering results. Therefore, data transmission time gradually becomes the main limit for real-time volume rendering. This is even more true in the case when visualizing time-varying volumetric data set as each time steps is in the order of 1 GB. For every individual time step, a complete copy of the volumetric data has to be transferred into graphic memory before rendering. The most advanced interface which connects the GPU to main memory for a conventional computer is the PCI-e 2.0 bus, which has a maximum transmission rate of 2.0 Gb/s for ideal conditions. In practice, this ideal rate can never be reached. We have tested a state-of-art NVIDIA Quadro FX5800 GPU and in most cases it took 0.07 to 0.12 seconds to transfer 128 MB of data to the video memory, making it impossible to satisfy our constraints of a minimum of 10Hz update. Our real-time constraints mean that the data transmission time between two different time steps should be less than 50 ms. In other words, the amount of data that is allowed to be transmitted between two time steps should be less than 185 MB on a PCI-e 2.0 bus. If this constraint is not respected, no matter how fast the GPU is, the rendering speed will not real-time.

To tackle the data transmission delay problem, a simple way is to add more video memory into the GPU. If all the time-steps could fit into the GPU memory, no data will be transmitted during rendering stage, hence no transmission latency. The increasing capacity and lowering price of memory justify this unsophisticated approach in some sense. However, the size of volumetric data can possibly be infinite. Data compression is a more complex yet feasible way to deal with this issue as it has been used successfully in many video and audio applications over the Internet. A good compression algorithm can greatly reduce the amount of data which needs to be sent to the GPU. Many researchers [51] [17] [25] adopted data compression in their attempt to visualize in real-time large volumetric data set. Octree-structures, multi-resolution representation, wavelet-compression, run-length encoding and vector quantization are the techniques applied in previous works. The trade-off for this compression strategy is the time needed to perform decompression. Any algorithm design should consider the cost of decompression in the rendering loop. If not a compression approach would not be useful if the decompression time takes longer than memory transfer time.

1.3 Proposed System

While the demands of visualizing time-varying volumetric data is growing, most research in volumetric data rendering still focus on algorithms which deal with stationary volumetric data. In most cases, time-varying volumetric data visualization is just a by-product of their research, by treating each time step individually as stationary volumetric data and visualizing them with their algorithms one by one. It is the easiest way to make the transition from stationary volumetric data rendering to time-varying data rendering; but it is definitely not the most efficient way. In most work, only space redundancy is explored in stationary volumetric data rendering algorithms. Most work ignores time redundancy, hence the proposed system, which deals with time-varying volumetric data, is more efficient than conventional rendering techniques. Our proposed system not only explore the time redundancy between two continuous stationary volumetric data, but also borrow an efficient

algorithm which take advantage of the space redundancy as well.

Also, as we analyzed, the bottleneck of large time-varying volumetric data is how to reduce the data transmission among different storage devices. Our proposed system efficiently deals with the bottleneck by porting decompression of the GPU. The system has four modules: encoder, decoder, render and interaction handler. The encoder module reads the original data from the hard drive and produces compressed data using temporal and space redundancy. We then save the compressed data back onto hard drive. The format of the saved file is designed to contain only minimal information which is needed for rendering. The decoder module reads the compressed file from hard drive and transmits it from main memory to the GPU memory. The GPU then decompresses the original data in GPU memory and then renders the volume from the user specified viewpoint. This mechanism minimizes the amount of data which is required to transmit from main memory to GPU memory for visualization. Texture-based volume rendering method is the volume rendering algorithm we used in the rendering module. The interaction handler module supports multiple useful interactions including: side-by-side stereo display, volume clipping, and mouse triggered rotation and movement.

In Chapter 2 we will do a review of the pertinent literature, and in Chapter 3 describe the system design and implementation. Experimental results and performance analysis will be presented in Chapter 4. We will then conclude and describe future work.

1.4 Thesis Contributions

Along with the growing needs for real-time rendering of time-varying volumetric data, a system specially designed for this task is both necessary and desirable. The main contributions of this thesis are:

1. A cross-platform interactive system which is able to visualize large-time varying volumetric data is implemented. Several commonly used interactions are supported. A side-by-side stereo mode is also available for immersive displays.

2. We propose and experiment a modified GPU decompression mechanism. This mechanism only requires the compressed data for volume rendering. It reduces more than half of the data preparation time compared to not using this mechanism.

3. Both space and time redundancy of time-varying volumetric data are used, which yields a better compression performance. The proposed compression distortion is less than an existing algorithms for stationary volumetric data.

4. The modular design concept makes it easy to customize or extend our system. The portability is also taken into account when implementing our system. Both NVIDIA and ATI graphic cards are supported in our framework.

Chapter 2

Literature Review

In this chapter, we will start with a brief introduction of GPU (Graphic Processing Unit) architecture. Then, we will review various volume rendering algorithms one can find in the literature. We will also describe techniques which have been used for large volumetric data rendering, such as bricking and volume compression.

2.1 GPU Architecture

GPUs are highly optimized data-parallel streaming processors which was originally dedicated to floating point operations for 3D graphics applications. NVIDIA coined the term GPU in the late 1990s. Since then, there have been an extraordinary evolution of GPU processing power and memory. Today, powerful yet cheap GPUs can process data at double precision and at speed reaching one to two tera-flops. In the history of GPU development there have been four generations according to [15]. Each new generation features a more flexible programable pipeline and a significant performance improvement. In our system implementation a forth generation or newer GPU is required in order to have the flexibility to perform all computation of our algorithms.

Before the invention of GPU, only workstation, clusters or specially designed hardware wre capable of performing real-time volume rendering [48]. Today, all the state-of-art rendering algorithms are implemented on GPUs without exception. To leverage the GPU computational power for real-time volume rendering, we need to review the basic concepts of GPU.

2.1.1 GPU Pipeline

Graphic pipeline is one of the most important ideas in modern computer graphics. OpenGL and Direct3D are two notable graphics pipeline models as an industry standards. While different GPUs may have different implementations of the graphic pipelines, the essential components are universal and standardized. Figure 2.1 shows the common pipeline of a general GPU hardware. The input of this pipeline is an ordered data stream of graphic primitives and graphics commands. The output is a raster image which will be displayed on the screen. The graphics commands are executed by the vertex and the fragment processors in the pipeline. The programmability of GPU comes from these two processors; and they are continuously getting more and more flexible making GPU a more general purpose processor.

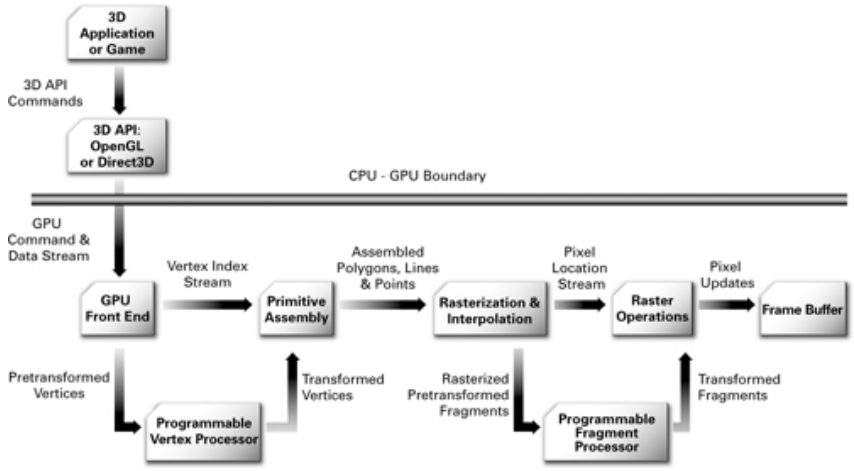


Figure 2.1: The programmable graphics pipeline. Taken from [15]

To control the behavior of the GPU pipeline, several GPU programming languages are available. The most widely used ones are Cg, GLSL, OpenCL, and CUDA. Vertex and fragment programmability are fully supported by these programming languages. Cg and GLSL are quite similar languages and more oriented towards graphics. It is not hard to migrate codes from one to the other. In our system, we choose Cg as the language for our GPU programming. Cg is a high-level shading language very similar to C. It is widely supported by most GPUs on the market. To write an efficient Cg program, GPU Programmers need to balance the

processing load between the vertex and the fragment shader.

In terms of computational power, GPU has exceeded the CPU for a long time. The amount of transistors on a chip is the most common measure of computational power. The top-of-the-line Intel Xeon 7400 has around 330 million transistors, while NVIDIA's GeForce 295 uses over 1400 million transistors [61] [60]. However, the programmability of GPU could not rival a general-purpose CPU, as most of the programming capability of a GPU is for computation that can be defined by Multi-Instruction Multiple-Data (MIMD) architecture. Not all features required for general computing are supported by GPU programming. For instance, object-oriented programming, pointer, file input/output, recursion, and even memory allocation are not supported in Cg language. When programming for GPU, one should always be aware of these limitations. CUDA and OpenCL have changed many of those limitations but when we started the project Cg was the only available language to program GPU and is still the most efficient computationally.

2.1.2 GPU Memory

As we mentioned in the previous section, GPU integrates several millions of transistors on a single chip. To fully harness the computational power of this hardware, a fast memory access speed is necessary. For numerous computationally intensive graphic tasks, fast memory access is vital in order to achieve real-time rendering performance. Take real-time volume rendering as an example, it involves a tremendous number of memory reads and writes. Millions or even billions of sample points need to be interpolated for every frame. At each sample point, a trilinear interpolation is performed. Trilinear interpolation is the primary interpolation method today; because it provides a very smooth rendering of the volumetric data. Eight or even more memory reads must be performed for a single trilinear interpolation. Therefore, at least more than eight million memory reads are required for every frame. A huge memory bandwidth must be available for real-time rendering; otherwise, the memory latency will slow down the rendering speed severely.

The gap between computational power and memory access speed is a common problem for hardware design. The cache memory hierarchy, which has been used

for years in CPU architecture, proves to be a cost-efficient solution to fill the gap and reduce the latency of multiple memory access. GPU architecture also takes advantage of cache hierarchy. Figure 2.2 shows the memory architecture of CPU and GPU. They are very similar to each other. This hierarchial cache architecture is usually composed of two levels of caches and one register. The register has the fastest access speed. Instead of communicating with GPU memory directly, the processors attempt to look up the required data in register. If the data already resides in the register, it is called a cache hit, otherwise, a cache miss. The performance of volume rendering could be greatly affected by the number of cache hits or misses. This is due to the fact that a single cache miss may consume the same time as many cache hits. Memory cache optimizations have been proposed by Kiefer et al. [33] to optimize memory access speed. An average speed up of 3.4 units was observed in their experiment. While cache mechanism alleviates multiple memory access problem, the memory bandwidth inside GPU itself increases significantly. Today, the most advanced NVIDIA Quadro Fx 5800 has a peak memory bandwidth of 102 GB/sec. The fast access speed, combined with cache mechanism, provides a very fast and efficient memory access speed inside GPU. We call the memory access speed inside GPU "local memory bandwidth".

Other than local memory bandwidth issue, another problem with GPU memory is its relatively small capacity. Most GPUs have their own dedicated memory. However, due to the high cost, it is usually smaller in capacity than most main CPU memory. There is no problem if the input data can fit into GPU local memory, however, it is not always the case. Real-time volumetric data of a beating heart from a real-time CT could easily exceed the capacity of any existing GPU. Data replacement strategies, which we will discuss later in this chapter, must be applied in order to deal with such data. When the data replacement happens between main memory and GPU memory, the PCI-e 1.0 and 2.0 bus are used for data exchange. The latest 32-lane PCI-e 2.0 technology allows up to 16 GB/s memory access. This is a theoretical speed as in reality, it is much lower.

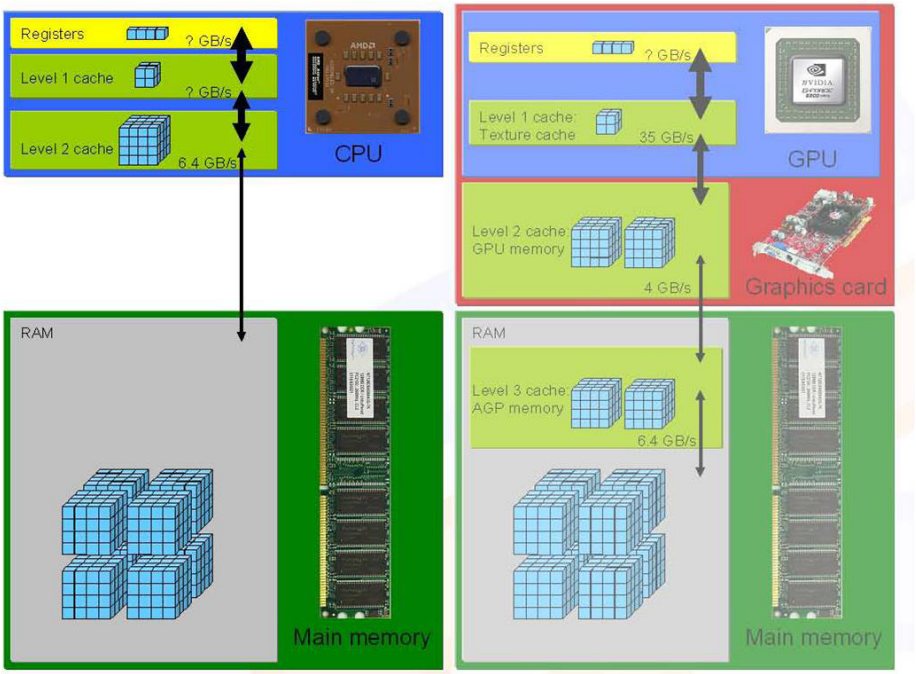


Figure 2.2: Memory hierarchy of GPU and CPU

2.2 Real-Time Volume Rendering

Volume rendering has been extensively studied for many years. Generally, there are five different optical rendering models for volume rendering, which are *Absorption Only [26] [8], Emission Only [27], Emission-Absorption Model [9] [2], Single Scattering [59] [12], and Shadowing [45] [28] and Multiple Scattering*. Single Scattering and multiple scattering calculations are expensive in computer time, hence not suitable for real time rendering yet. The most widely used is the Emission-Absorption Model. Both light emission and absorbtion in the volume are taken into account in this models. The physical model can be described by the following integral:

$$I(D) = I_0 e^{-\int_{s_0}^{D} k(t)\,dt} + \int_{s_0}^{D} q(s) e^{-\int_{s}^{D} k(t)\,dt}\,ds$$

In this equation, I_0 represents the background radiance at the start position $s = s_0$, while $I(D)$ is the final radiance reaching the camera at the end position $s = D$. The first part of the equation represents the background light contribution to the final image after attenuation by the volume. The second part describes the total contribution of the emitted light, which also is attenuated by the participating medium, between

s_0 and D. The integral cannot be solved analytically and a Riemann sum [44] is usually used to approximate the integral. Hence, the approximation of the integral is:

$$I(D) \approx I_0 \prod_{i=0}^{n} e^{-k(s_0+i*\Delta t)\Delta t} + \sum_{i=0}^{n} q(s_0 + i * \Delta t) \prod_{j=i+1}^{n} e^{-k(s_0+j*\Delta t)\Delta t},$$

where $\Delta t = (D - s_0)/n$. Solving the equation of volume rendering is both a computational and memory intensive process. It is even getting more challenging with the continuous trend towards higher and higher resolution data sets. Although many efficient software optimization techniques exist, it is difficult for general-purpose CPU, to deliver a real-time performance. Almost all existing real-time volume rendering algorithms one can find in the literature are either performed in parallel using multiple processors, GPU, or cluster.

2.2.1 Real-Time Volume Rendering on Clusters

In the early days of real-time volume rendering, the computing resources required to achieve interactive rendering speed were far beyond those available for a single CPU. To achieve real-time rendering speed, supercomputers and massive clusters became the intuitive choice to circumvent the intensive computation.

An excellent survey of computer architectures dedicated to real-time volume rendering can be found in [29]. The adoption of *binary classification* technique are the most distinguishable feature of the algorithms discussed in the paper. A binary classification classifies the input volume into two categories 0-voxels and 1-voxels using a threshold. 0-voxels were then discarded directly to reduce the memory consumption and to speed-up the rendering process. After careful examination of a number of machines which optimized the binary classification , Kaufman propose a CUBE(CUbic frame Buffer) architecture [30]. This architecture consisted of many processors and a 3-D cubic frame buffer. The data in frame buffer was manipulated by 3 processors. A unique skewed memory organization and a multiple-write bus were also introduced into this architecture for huge volume data. This CUBE system allowed image generation time of n^3 voxels reduced to $O(n^2 logn)$ from the conventional $O(n^3)$. Real-time rendering speed is achievable through binary classi-

fication. However, this technique also led to false positives or false negatives, which severely affect the image quality.

To improve the image quality while not sacrificing speed, researchers in this field widely use *volumetric compositing*. This technique basically assigned to every voxel a color and an opacity. The final image, which was combined by several images, was then generated by a 2D projection of the semi-transparent volume. Binary classification was no longer required, hence false positive and false negatives disappeared. Drebin et al. [12] wrote a quite influential paper based on volumetric compositing. Many concepts in the current pipeline of volume rendering first appeared in this paper. They managed to produce better images than any previous algorithm through volumetric compositing. However, their algorithm was not designed for real-time rendering by any means. It could take several seconds to generate only one image. Later researchers in this field proposed various hardware frameworks and optimization methods to improve the performance. Both hardware and software solutions adopted parallel computing strategy to reduce the rendering time. Like any other graphic task, volume rendering is highly parallelizable by nature.

Levoy [38] proposed a volume rendering framework based on his image-order volumetric compositing algorithm [37]. Their system was implemented on the PIXEL-PLANES 5 hardware. The workstation consisted of 32 independently programmable 20-MFLOPS graphics processors(GP's), 16 independently programmable renderers with 1/4 million pixel processors, a $512*512$ pixel color frame buffer, and a 640 Mb/sec ring network. Empty space skipping and occlusion culling, which have been discussed in [37], was used to further accelerate the rendering speed. They were able to achieved real-time or near real-time performance.

Schroder et al. [55] explored the data parallel property of ray tracing based volume rendering. For a single ray through the volume, they created an idiom "line drawing" to enumerate the ray tracing steps. Only the rays between the frontmost face and the far face are taken into account in their algorithm. Lock step applied when the ray entering the volume, and toroidal shift preformed after the ray come out. As a result, the rotation angle selected reduced by 10% the runtime of

the algorithm. They have implemented their algorithm on both Princeton Engine, which consists of 64 to 2048 16-bit DSP processors arranged in a ring, and the Connection Machine CM-2, which consists of of 4k to 64k bit-slice processors with one floating point chip for every 32 processors. Arbitrary parallel projections at interactive rendering rates for a 128^3 voxels volume data set was achieved on both of them.

Perhaps the most notable real-time volume rendering algorithm on clusters is designed by Lacroute et al. [48]. It has been recognized as the fastest software render algorithm. They managed to render a 256^3 voxels medical data set at a speed of 10fps and a 128^3 voxels data set over 30fps on a Silicon Graphics machine with 16-processors. The most significant contribution of this paper is the adoption of shear-wrap algorithm, which became a popular method in volume rendering later on. The basic idea of shear-wrap is to shear the volume such that the rays are perpendicular to the volume slices. Bilinear interpolation applied within the traversed volume slices, hence the image quality was not satisfactory.

The design of parallel architectures for real-time volume rendering naturally led to the invention of a dedicated volume rendering hardware, *Volume Pro*. It is the first single-chip hardware, which is compatible with consumer PCs, for real-time volume rendering. Pfister et al. [19]' development is based on their previous Cube-4 framework. This hardware was specialized for shear-wrap algorithm describe in [48]. Instead of linear interpolation, trilinear interpolation was performed to achieve a better image quality. Other features, such as super-sampling and cropping were also incorporated into the VolumePro's architecture. Compare to cluster based method, the VolumePro architecture offers an affordable alternative for real-time volume rendering. However, not every user is willing to buy the hardware only for volume rendering. Obviously, a more "general-purpose" yet powerful graphic hardware is necessary.

2.2.2 GPU Accelerated Volume Rendering

As the demand for graphic computing grew, a separate hardware which dedicated to graphic related tasks became more and more urgent. Chip manufacturer spent

lots of effort on designing such a hardware to satisfy the growing needs. Graphic Processing Unit (GPU) is the outcome of this effort. Ever since its introduction, it got lots of attentions. At the beginning, the computation abilities of GPU were not as powerful as CPU. However, GPU hardware speed manage to evolve faster than Moore's Law because of its true parallel nature. Now cheap yet powerful graphic cards are available everywhere on the market. The widespread use of GPU turned the demanding real-time volume rendering requirement is now possible to be performed on a commodity computer. GPU-based volume rendering became quite popular among researchers because of this reason. Although GPU is not specifically designed for volume rendering like VolumePro, it proved to be very efficient. Data parallelism and high local bandwidth of GPU contribute to its superior performance. Two different types of real-time GPU volume rendering was developed: texture-based rendering and ray-casting.

Texture-based volume rendering was implement on GPU first. One of the main tasks of GPU is texture mapping. Texture mapping was first introduced by Catmull [9] in his PhD thesis. It is a technique to add fine details to 3D geometric objects by using images or texture. A large number of graphic applications rely on texture mapping today. This situation motivates chip manufacturer to continuously improve the efficiency of GPU texture mapping. The texture fill-rate grew from 480 Mega-texels /s (GeForece 256, 1999) to 46.08 Giga-texels/s (GeForce GTX295, 2009). This speed improvement of two order of magnitude happened within 10 years. Volume rendering algorithms based on texture mapping benefit tremendously from this development.

Two dimensional texture mapping became available on GPU earlier than three-dimensional texture mapping. As a result, real-time 2D texture-based volume rendering on GPU were studied by researchers first. Engel et al. have written an excellent introduction of 2D texture mapping volume rendering algorithms in [27]. The volumetric data were stored in several 2D texture images first. Then the object-aligned slices of the data set got projected onto the image plane. These projections were blended in a back-to-front order to form the final image. It is a very intuitive method and easy to implement. A fast rendering speed at medium quality was de-

veloped because of the bilinear interpolation when in reality trilinear interpolations is really needed. The bilinear interpolations were natively supported by 2D texture mapping hardware. In terms of subjective image quality, the mathematical accuracy, and the memory consumption, this algorithm had an enormous potential to be optimized.

Bilinear interpolation is not ideal for volume rendering as it creates severe artifacts. Rezk-Salama et al. [49] introduced a multi-texture-based volume rendering algorithm. The general idea of their algorithm is to substitute a trilinear interpolation by two bilinear interpolations and one linear interpolation. A better and fast estimation of the scalar at an arbitrary position along the chosen axis in the volume space is achieved through this approach. Therefore, an improved image quality compare to the first plain 2D texture-based rendering algorithm was achieved without losing too much rendering speed. The deficiency of this algorithm is the high storage requirements. Three copies of the volumetric data set in main memory were required to eliminate the switching artifacts when users change the view direction more than 45 degrees. The algorithm could only render very small data set.

Engel et al. [27] have examined the possibility of getting rid of the extra two copies while not affecting the image quality. By on-the-fly reconstruction of volumetric data, which definitely introduced severe performance drop off, it is possible to store only one stack of the original data for rendering. This approach is only beneficial when the GPU texture memory is not enough to store three copies, but enough for one copy. If the entire data is larger than the size of texture memory, more sophisticated techniques, such as paging [52] and out-of-core rendering [13], are needed.

Three-dimensional (3D) texture mapping is a more sophisticated algorithm compare to 2D texture mapping. Cabral et al. [5] examined 3D texture-based volume rendering on an SGI RealityEngine, the first hardware which implemented 3D texture mapping. The volume rendering integral was treated as generalized Radon transforms in their method. In this paper, they present a detailed implementation on how to perform the radon transform on a RealityEngine. Details about the architecture of RealityEngine can be found in [2]. It was designed as a high-end

workstation which has special abilities in image generation and image processing. However the high cost of the RealityEngine or any other 3D texture mapping optimized workstation at that time made the algorithm inaccessible for common usage.

As we have described previously, GPU is quite affordable and powerful. Considering the virtues of 3D texture-based volume rendering, it is quite reasonable that it became the mainstream algorithm after the introduction of 3D texture mapping. The 3D texture in GPU was defined as volumetric texture object, which could be viewed as a solid 3D texture object. The users supplied several sets of 3D texture coordinates to "cut out" the object. 3D texture-based volume rendering on GPU used view-port aligned slices to cut out the texture object. The view-port aligned slices were the polygons formed by the intersections between the 3D texture cube and several arbitrarily oriented planes. They were traditionally computed by CPU. Rezk-Salama et al. [50] proposed a vertex shader program to offload the computation of view-port aligned polygons from CPU to GPU. This could balance the workload between the vertex shader and the fragment shader. The vertex shader program was based on the observation that all the 12 edges of a cube could be recognized as 6 different sets of edges. For each set, at most one coordinate, which represents the intersecting point of the edge to an arbitrary plane, could be found. A mathematic proof of this observation is not given in the paper and will be given in Appendix A. Together with some optimization methods, such as empty space skipping, Rezk-Salama et al. [50] achieved a significant performance improvement compare to most of the previous algorithms, while maintain the same image quality. After the view-port aligned slices computation, and maybe empty space skipping in the vertex program, the next step is to execute the fragment program. The fragment programs have permission to access 3D texture objects. Texture set-up and downloading should be done first before the start of fragment program. One can refer to the **OpenGL Red Book** [11] on how to manipulate 3D texture objects in a GPU. One must be careful when choosing the mag and min filters. These two parameters will affect the interpolation preformed by the GPU texture mapping hardware. The fragment programs mainly take care of the texture fetching and color table (transfer function) look up. Some optimization techniques could also be used in the frag-

ment programs. Occlusion clipping which origin from ray-casting methods [39] is such an example. Li et al. [40] reported an average of 2.8 speedup by using those techniques.

GPU-based ray casting method was developed after texture-based method. The basic idea of volume ray casting is to cast rays from the virtual camera into the volumetric data and to calculate the volume integral along each ray. The volumetric data in here is represented by a 3D texture in the GPU the same way as texture-based algorithms. However, the view-port aligned polygons, which must be calculated in texture-based algorithms, were replaced by entry positions of the rays into the volume. The pseudo-code for the calculation of volume integral of a intersected ray is shown at Algorithm 1. As each ray is independent, a parallel algorithm on GPU which take advantage of the parallelism of the rays is quite intuitive. The first implementation of ray-casting on GPU was published in 2003. Roettger et al. [53] proposed the hardware-accelerated ray caster with early ray termination and empty space skipping on NVIDIA GeForce FX or ATI Redeon 9700. This implementation can be classified as a multi-pass GPU ray-casting technique. Some key features, which are essential in the CPU ray-casting implementation, such as loops and conditional breaks were not fully supported by GPUs at that time. As more fragment shader functionalities became available on GPU, Stegmaier et al. [58] proposed a single-pass GPU ray casting framework. They provided a assembler-level fragment programs in their paper.

Algorithm 1 Volume integral calculation of a single ray, inspired by [27]

```
Calculate the entry position of the current ray
Get the ray direction
while rayisstillinthevolume do
    Interpolate the value of current position
    Compositing of the current value with previously accumulated value
    Advance position along ray direction
end while
```

Compared to GPU-based ray casting volume rendering, GPU-base texture mapping algorithm still has several advantages. The rendering speed is faster and the implementation is more portable among different hardware versions. As stated in

[56], the perspective projections can be more easily implemented with texture-based volume rendering, since only a proper scaling factor needs to be assigned to each slice based on several viewing parameters while in ray casting the direction of each ray needs to be determined individually. Hence, we only adopted texture-based volume rendering algorithm into our system. Although, if the end users absolutely require GPU-based rendering algorithm, it is not hard to integrate it into the render module of our system.

2.3 Large Volumetric Data Rendering

The size of volumetric data is continuously increasing because of more advanced data acquisition methods. The virtual address limit of 32-bit PC computers could be easily surpassed by the data. Take Siemens Axiom scanner as an example, it is capable of producing 2D slices with a resolution of 200 microns. One of its typical scanning resulted is a 92 GB volumetric data. This is 23 times the maximum virtual memory address of 32-bit computer. Using normal volumetric rendering algorithm is impossible to visualize such data. Many methods have proposed to tackle this issue.

2.3.1 Bricking

The simplest method to deal with the problem is *bricking*, which is essentially the well-known divide-and-conquer approach in computer science. The original volume is divided into several smaller sub-volumes according to the criteria imposed by individual hardware configurations, to ensure that each sub-volume can be fitted into GPU memory. The final rendering result is composed of several images from the individual volume rendering of all the sub-volumes. Through the whole rendering process, sub-volumes have to be replaced by other remaining sub-volumes to have access to all the volumetric data. The waiting time for block replacement needs to be minimized to make full use of the GPU. Therefore, an efficient block replacement strategy is crucial for efficiency.

Bricking is usually not the answer for real-time large volumetric data rendering

on a single GPU. Even with the latest PCI-e 2.0 technology, the data needed to be transferred can easily starve the bandwidth. The reason is because bricking does not reduce the amount of memory required to represent the original data. On the contrary, it even increases the amount of data. Extra layers for each sub-volume is required to eliminate inconsistent interpolation due to bricking.

For multiple processors volume rendering, bricking is an inevitable step in the rendering pipeline. Take the well-known parallel rendering software, *ParaView* [1], as an example, it divides the original data into equal sized sub-volumes at first. This step is essentially a bricking technique. Then, sub-volumes are distributed to several data servers. Each data server has at least one rendering server connect to them. The rendering sever is responsible for rendering the sub-volume in a data server. After parallel rendering of every data server, IceT, which is a parallel rendering library for image-compositing, is used to perform the compositing of every image from the rendering servers. In this case, bricking is essentially a form of data parallelism.

Except for multiple parallel processors, a real-time rendering speed can hardly be achieved by a simple bricking algorithm. The inter-connection speed between CPU and GPU is not fast enough to satisfy the needs for real-time volumetric rendering. Further improvements which could reduce the memory consumption could improve this situation.

2.3.2 Hierarchical Volume Rendering

As many data intensive applications benefit from hierarchical data representation, large volumetric data rendering algorithms also rely on hierarchical data structure to improve processing. Most algorithms in this subfield process the volumetric data into different resolutions in the spatial or the frequency domains. Octree or similar spatial structures [6] [35] and wavelet transformation are the most popular methods to transform the original data into a multi-resolution representation. Some low resolution data were selected to replace the original one during rendering to reduce the size of volumetric data that have to reside in GPU memory. The rendering quality is not severely affected by the reduced resolution of volumetric data. The reason is that we only replace the data which is far away from the view plane with down-

sampling . Many algorithms use different selection and replacement strategies to perform this task.

The octree spatial hierarchy was first proposed by Chamberiain et al. [6] when they were dealing with rendering of surface (polygonal) geometry. Their algorithm started with an octree subdivision of the data in a preprocessing step. During rendering, they recursively traversed the octree. A projection size of each octree was also computed on the fly. When the size is smaller than a predefined threshold, they considered the current node to be sufficiently accurate to approximate all the nodes below it. More information can be used to determine the selection criteria. LaMar et al. [35] developed a similar technique. Instead of using projected size as the sole selection filter, they selected the hierarchy based on two criteria: distance and filed-of-view. Distance criteria is similar to Chamberiain's algorithm. Field-of-view criteria discarded the nodes which did not intersect the view frustum and selected nodes whose projected angle was less than half the field-of-view angle. A considerable speed up was observed compare to Chamberiain's algorithm. The deficiency of this method, which also appeared in other similar algorithms, is the substantial artifacts at the boundary of different level of details. The inconsistencies of the view-plane at different level of details resulted in visual artifacts. Weiler et al. [42] proposed an extension to smooth the discontinuity artifacts based on the observation. A common problem of the above algorithms is the inability at reducing the volumetric data size in main memory, which limits themselves from rendering data set larger than the main memory size.

Octree and octree-like structures explored the spatial hierarchy of volumetric data set. It is also worthwhile to examine the frequency domain analysis of volumetric data set. Wavelet transformation is one of the most famous methods in the frequency domain. It has been thoroughly studied by researchers in recent years. An overview of wavelet transform and compression can be found in Stollnitz et al. [14].

The general steps of these methods are shown in Figure 2.3 and Figure 2.4. For encoding stage, the original data is transformed to frequency domain through wavelet transformation. The complete volumetric data is represented by wavelet

coefficients after the transformation. Some entropy reduction algorithms, such as run-length, arithmetic or Huffman encoding, are followed to reduce the size of the original data set. The decoding stage is illustrated in Figure 2.4. It consists of decoding, invert transformation, and conditional reconstruction steps. Conditional reconstruction selects the data representation which is sufficiently accurate for rendering. Depending on user needs, a quantization may or may not be used in the pipeline. A compression without quantization is usually called a lossless compression. Some applications, such as text compression, require lossless compression because no information loss is allowed. Most lossless compression algorithms are based on variable-length encoding, which maps most frequently used symbols to smaller number of bits than the original. The decoding process of most lossless coding requires previous information to make the next decoding decision. It also means the decoding process is a quite serial. This makes it difficult to implement on GPU. We will discuss about the issue in the next chapter. In the field of volume rendering and many other graphic related fields, lossless compression is, however, not very popular. There are basically two explanations. One is the relatively small compression ratio of lossless compression; the other is that constrained data inaccuracy might not be observed by users. Hence, as long as the compression of volumetric data does not introduce obvious artifacts after rendering, the compression algorithm is acceptable. Many researchers have developed numerous wavelet based lossless and lossy compression algorithms.

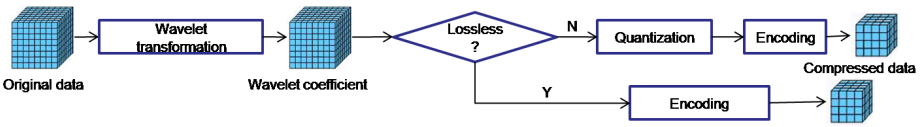


Figure 2.3: Wavelet based hierarchy encoder

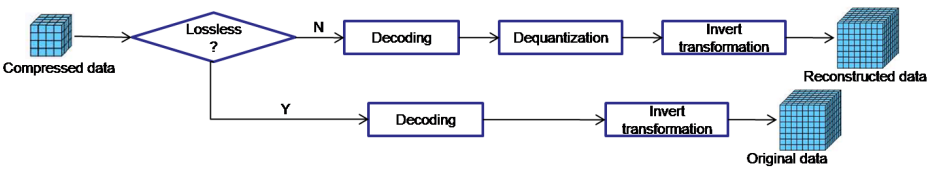


Figure 2.4: Wavelet based hierarchy decoder

Ihm et al. [21] first generalized the traditional 2D Haar wavelet transformation to 3D. Haar wavelet is the simplest and computationally the cheapest wavelet, resulting in a fast reconstruction speed. On the fly reconstruction was actually reported in their paper. Guthe et al. analyzed different wavelets and encoding schemes for 3D volume compression in [18]. They discovered that higher order wavelet is more efficient than Haar wavelet. A very interesting extension for encoding time-varying volumetric data was also proposed in their paper. The basic idea is borrowed from motion compensation in video encoding. In this context, we will also develop our own time-varying data compression scheme based on the same idea. More details will be provided in the next chapter. As stated in their paper, the problem for this method is that the whole volumetric data must be decompressed in main memory to allow access for parts of the data. They solved the problem in a later paper [51]. By integrating their previous method and the algorithm proposed by Nguyen et al. [46], they got a nearly interactive rendering speed on a NVIDIA GeForce Ti4600 GPU. Some frameworks of visualizing large time-varying data sets for wavelet-based volume rendering over PC clusters or workstations also have been proposed by researchers. Shen et al. [10] presented a rendering framework using the wavelet-based time-space partitioning (WTSP) tree for large-scale time-varying data visualization on PC clusters. Currently, we limit our system on a single PC. A detail review of large volumetric data rendering on clusters or workstation is beyond the scope of this thesis.

2.3.3 Vector Quantization

Besides hierarchical volume compression, another important algorithm for large volumetric data compression is called *vector quantization*. Vector quantization is a lossy data compression algorithm where the basic idea is to map a set of n-dimensional vectors to one fixed-length index which reference to a n-dimensional code-book. It is a fixed-to-fixed compression algorithm. The compression algorithm which selected in our system is vector quantization.

N-dimensional vector quantization is not very intuitive for most people. For illustrative purpose, we choose two dimensional vector quantization as an example.

Let N be the number of two-dimension source vectors one want to quantize:

$$S = \{(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)\}. \tag{2.1}$$

Let r denotes the number of bits for each index, which also means that the number of elements in a code-book is 2^r. We then define C as the code-book:

$$C = \{(c_{x1}, c_{y1}), (c_{x2}, c_{y2}), ..., (c_{xn}, c_{yn})\}, where\ n = 2^r. \tag{2.2}$$

Let P be the partition of the 2d plane:

$$P = \{p_1, p_2, ..., p_n\} \tag{2.3}$$

If $(x_i, y_i) \in p_j$, we map (x_i, y_i) to a r bits index j. The index j will refer to (c_{xj}, c_{yj}) in the codebook C. In the decompression process, (c_{xj}, c_{yj}) is decoded to approximate the original (x_i, y_i). In order to get a optimal code-book, an iteration is needed to minimized compression distortion. The average distortion is usually selected to measure the distortion, which is given by:

$$D_{avg} = \frac{1}{Nr} \sum_{m=1}^{N} \parallel \vec{x_m} - \vec{c_m} \parallel^2 \tag{2.4}$$

where $\vec{x_m}$ is the original vector and $\vec{c_m}$ is the corresponding approximation of $\vec{x_m}$ in code-book C. Linde, Buzo and Gray [62] first developed an iterative algorithm to minimize the average distortion. It is a premature algorithm though. The execution speed is extremely slow and it sometimes results in empty cell problem. The empty cell problem will greatly increase the distortion and execution time. The splitting strategy used contributes to the problems significantly. Principal component analysis (PCA), which is commonly used in data clustering, is a better alternative. The splitting strategy which we use in our system is essentially the technique used in [41] [25]. Not only it speeds up the compression process, but also avoids the empty cell problem. We will have more discussion about it in the next chapter. For a more detailed introduction or survey, please refer to [4] and [54]. Vector quantization method works extremely well for data with many similar sub-blocks. A high compression ratio, as well as a small distortion, could be achieved at the same time. The high spatial and time redundancy is one characteristic of time-varying volumetric

data, which make vector quantization an excellent choice for large time-varying volumetric data compression.

For some highly sparse (more than 99%) volumetric data sets, Binotto et al. [7] designed a lossless compression approach based on vector quantization. The highly sparse character of volumetric data sets in their experiment contained a large number of identical blocks. These identical blocks were then coded in a code book; and a 8 bits index was used to store the position of the block in the code book. In the reconstruction stage, the index was stored as a 3D texture in GPU and accessed by a fragment program to look up the original value in the code book. The limitation of this lossless compression is also obvious. Most of the volumetric data is not highly spare. Thus, most of them can not be compressed by this algorithm.

Schneider et al. [25] propose a more general, however, lossy compression algorithm based on vector quantization. They tested it on various volume data set. Without introducing any obvious artifact, around 20:1 compression ratio was obtained. It is comparable to wavelet based compression algorithms in terms of compression rates and fidelity. The advantage of this algorithm is the decompression happened on GPU, while wavelet based compression had to decompress the original data in main memory first. In this way, data transfer between the CPU and the GPU can be minimized. This is desirable for large or time-varying volumetric data rendering. Larger data could introduce data access delay to rendering algorithm. In our system, we choose Schneider et al.'s [25] algorithm as the compression algorithm for our system. Some modifications are made to improve the performance for time-varying volumetric data compression. Details about modifications and results will be described in the next chapter.

2.4 Summary

Large time-varying volumetric data rendering became more and more popular because a requirement for higher volumetric data resolution. The advances of GPU allow a number of efficient and high-quality rendering algorithms at low cost. However, the gap between large data size and limited computation resources is expand-

ing not reducing as new medical imaging modalities get introduced.

In this chapter, we have reviewed several related papers on the issue. According to the literature, a typical large volumetric data rendering algorithm usually adopted a compression step before rendering. Hierarchy compression is the most important and efficient technique for such task. Most previous frameworks used wavelet compression to reduce the volumetric data. However, the decompression of wavelet-based algorithm can only be done on the CPU. The main reason is that the decompression algorithm usually depends on information from previous decompressed data, hence it is a serial process and is very hard if not impossible to implement on GPU. The same theory applies to other decoding algorithms, such as huffman decoding and run-length decoding. To minimize the data transfer between GPU and CPU, we examined vector quantization for time-varying data set rendering. The decompression of vector quantization is simple and parallel enough to be implemented on GPU because of its fix-to-fix encoding mechanism. The compression result is also comparable to hierarchical wavelet compression in terms of compression ratio and image quality according to Schneider et al. [25]. Because of those reasons, we decided to choose vector quantization as the main compression algorithm for our large time-varying volumetric data rendering system. The rendering algorithm for our system could either be 3D texture-based volume rendering or GPU ray-casting volume rendering. These two algorithms are the most widely used volume rendering algorithms up to now. In the following chapters, we will discuss the design and implementation of our system.

Chapter 3

System Design and Implementation

We have discussed the importance and challenges of visualizing large time-varying volumetric data in real-time. In the light of pervious works in stationary volumetric data rendering, we proposed to implement a system designed for time-varying volumetric data rendering. For a specified image quality and rendering speed, the bandwidth problem need to be dealt with using real-time compression techniques.

3.1 System Overview

Our proposed system is designed to render large time-varying volumetric data in real-time. In order to do so, we had to take flexibility and compatibility into account. The flexibility allows user to customize our system with their own algorithm easily and the compatibility makes our shader programs be able to run on most GPUs, either NVIDIA or ATI.

Figure 3.1 shows the overview of our system. We generalize the process of visualizing volumetric data into four different modules. Each module is decoupled with each other, so that users have the flexibility to adopt their own algorithms into any module. The four different modules are: encoder, decoder, render, interaction handler. They are represented by white rectangles in Figure 3.1. The system's dataflow follows the order of encoder, decoder and render. The interaction hander module provides parameters from input devices to the rendering module to generate interactive feedback and manipulations.

The original volumetric data is usually obtained from CT or MRI scan. The

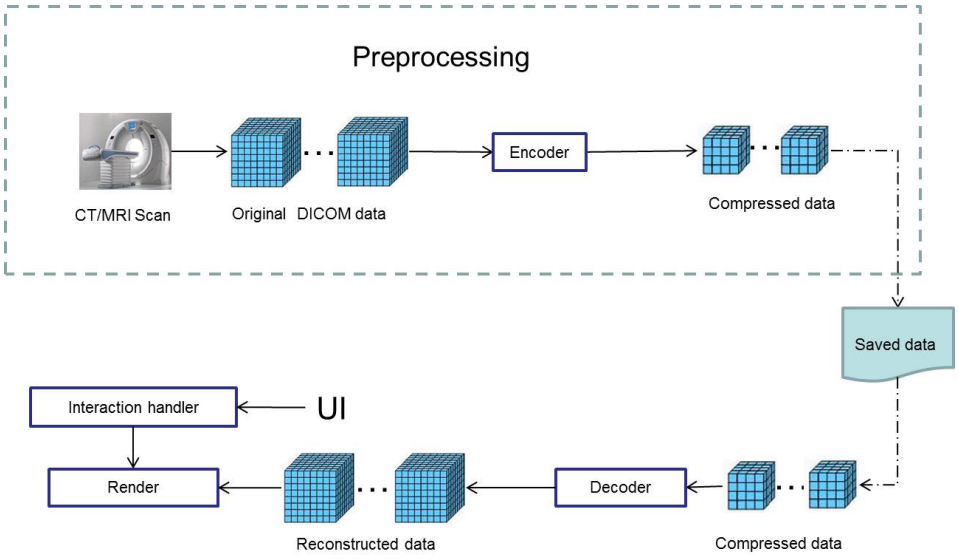


Figure 3.1: The overview of our proposed system.

scanner generates a large set of raw data exported to DICOM format and saves them into hard disk or other massive storage devices. The encoder module then reads these DICOM files and processes them using the compression and encoding algorithms. At this point, the compressed data resides in main memory. Before proceeding to next module, our implementation saves the compressed data into hard disk in a dedicated format described in next section. It can of course be used as the input of decoder module directly; but it is recommended to save the result first. The compression process in our system takes a long time due to the complexity of compression algorithm. It is redundant and time-consuming to regenerate the compressed data each time. The compressed file is on average 20 times smaller than its DICOM version.

After preprocessing, the file reader reads the compressed data from hard disk into main memory. The decoder module then upload the compressed data into GPU video memory for one time step. The decoding process is performed in the GPU after uploading. This is one of the most distinct features of our system. The combination of GPU decompression and volumetric rendering is not usual as most decompression algorithms are inherently serial, which makes an efficient decompression algorithm hard if not possible to be implemented on GPU. However, vector quantization, which is the algorithm we adopted in our system, is almost perfect

for GPU decompression because of its parallel nature. The decompression process is easy to be parallelized, which makes it extremely fast to be calculated by GPU. The decoder module reconstructs the original data from the compressed file as 3D textures in the GPU video memory. This mechanism requires that the GPU supports render to frame buffer object which can be directly read as 3D texture. It requires GL_ARB_framebuffer_object OpenGL extension. Please refer to [47] about the details. After the decoder module reconstructs the original data in GPU local memory as a 3D texture, the render module uses it for 3D texture-based volume rendering. The rendering algorithm slices and samples the texture memory to composite the final image displayed on screen. The interaction module allows users to interact with the system in real-time. In our implementation, users can switch between stereo mode and normal rendering mode. The stereo mode requires a polarized glasses and two polarized projectors. Users can also clip, move and rotate the volume data being displayed. Our implementation allows clipping along all three axis of the volume coordinates.

3.2 Encoder Module

The main function of the encoder module is to reduce the size of the original data in order to reduce the traffic between the CPU and the GPU. This is very important for our proposed system since we need to decompress the encoded data into GPU local memory. The decompression algorithm can be executed on the GPU very efficiently. The efficient decompression process saves us some precious time in real-time rendering compare to the normal process of decompressing data on the CPU and transferring it to GPU memory as it avoids the large transfer of each volume using the PCI-e bus.

Before an in-depth discussion on our encoder module, let's review the type of data our system deals with. The data is a large time-varying volumetric data, more specifically a sequence of DICOM data files, one for each time steps. The information in a DICOM file usually includes data resolution, time stamp, sample rate, coordinates, and other necessary parameters to characterize a DICOM frame. For

a full list of information available in the DICOM format, please refer to [23]. Not all the information which is available in DICOM file is needed for our rendering algorithm.

Original data from CT or MRI are usually not ready for vector quantization directly. There are a few preprocessing steps beforehand that need to be performed. We call these steps *Encoder Preprocessing*. The main components of the encoder preprocessing module are illustrated in Figure 3.2. In Figure 3.2, we define S_t as the original data sample of the volume at time t. Depending on the resolution of the sample data, the encoder module first decides if a re-sampling of the data is necessary. Let us define D_x, D_y, D_z as the resolution of the original data. If $D_x, D_y or D_z$ is not a power of two, one can re-sample the volume to its closest power of two. For instance our algorithm will re-sample a 60*120*79 data into 64*128*128. We define $D_x\prime, D_y\prime, D_z\prime$ as the resolution of the re-sampled data, $S_t\prime$. Re-sampling does not change the physical size of the original data. Trilinear interpolation is used to interpolate the value of the intermediate point from the original sample points. The reason for re-sampling is because our compression and rendering algorithm requires volume data with a size which is a multiple of the power of two in every dimension.

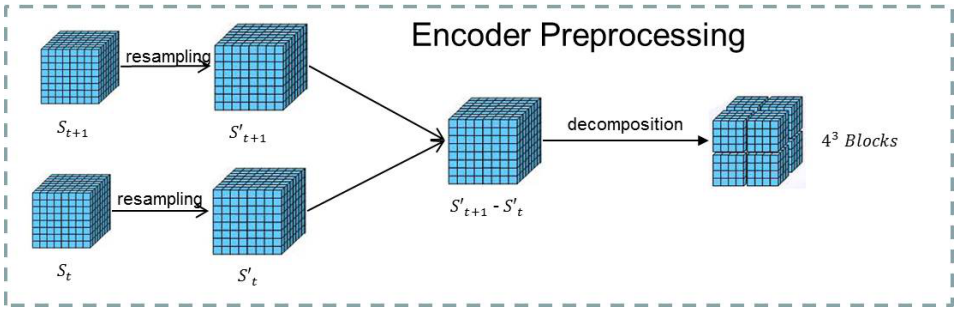


Figure 3.2: The preprocessing steps in the encoder module.

After re-sampling, all the data is set to the desired dimension. The next step is to exploit the time redundancy of the re-sampled data. In our case, we subtract the re-sampled data of two consecutive samples, which is $S_{t+1}\prime - S_t\prime$. The time interval between the two consecutive samples are usually very short and the differences between them are usually small. Most of the difference values are actually 0, which give us lots of opportunities to compress the signal. Vector quantization performs

better on these sparse blocks. In our experiment, a higher signal-to-noise ratio (SNR) [1] was observed when we quantized the sparse blocks instead of original data.

Other than time redundancy, space redundancy is also used before final compression. Hierarchical decomposition is a very efficient way to analyze the relationships of neighborhood sample points. Our system used the same decomposition method as in Schneider's paper [25]. $S_{t+1}\prime - S_t\prime$ is first divided into small blocks of size 4^3 as shown in figure 3.2. Next, these blocks are decomposed into a multiresolution representation. There are three steps to do the decomposition. Figure 3.3 illustrates the whole decomposition process. The first step is to divide the 4^3 block into 8 disjoint 2^3 blocks. Then the mean value of each 2^3 block are computed and stored in a new 2^3 block. We call this *detail level 0*. Essentially, detail level 0 is a down-sampled version of the 4^3 block. All the differences between the samples in 4^3 block and the corresponding mean value are stored in an array of size 64. We use *difference level 0* to represent the differences between two consecutive volume. The mapping from 4^3 block to the 64 array is given by $idx = x + 4 * y + 16 * z$, where idx is the corresponding index in the array for location (x, y, z) in 4^3 block. The last step of decomposition is basically the same process as in the second step. But it is applied to the detail level 0 block. The results are one value which is essentially the mean value of the entire block and 8 values which are the subtraction of each sample in detail 0 level block relative to the mean value. The overall mean value is also a down-sampled version of the original 4^3 block. The 8 differences from the last step will replace the values in detail level 0 to save memory space.

At this point, all the preprocessing of the quantization are finished. Let us take a look at how the blocks are represented in main memory. From top to bottom, the difference 1 level is represented by an array of size 64; the detail 0 level is represented by an array of size 8 and the detail 1 level is represented by a single variable in main memory. There are $D_x\prime/4 * D_y\prime/4 * D_z\prime/4$ array or variables for each level. In most case, the element in the array is 8 bits. To give a rough

[1] The performance of VQ are typically given in terms of the signal-to-noise ratio (SNR). The higher the SNR the better the performance

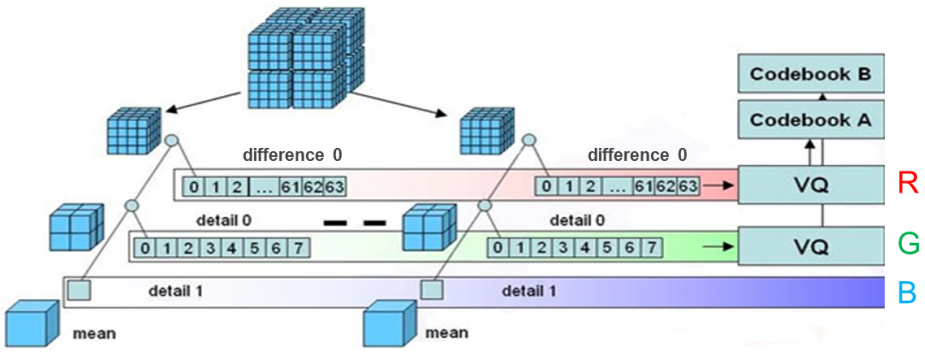


Figure 3.3: Data decomposition and vector quantization in the encoder module.

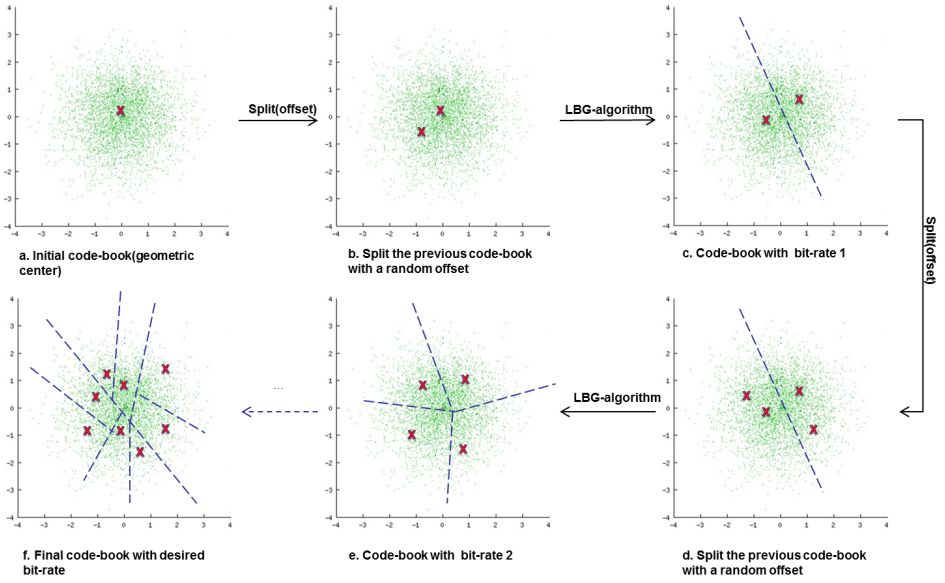
sense of memory size, lets take an example where the original data has a resolution of $128 * 512 * 512$, which is $32MB$. After decomposition, the memory size is $(128*512*512+128*512*512/8+128*512*512/64)*8 = 306184192bits = 36.5Mb$. this correspond to an increase of memory size of $(36.5-32)/32 = 14\%$ from the original data. Each vector will be represent by a 8-bits value at the end of quantization, which means the length of code-book A and B is 256. The original 8 or 64 dimensional vectors are either in the code-book or approximated by it.

To quantize these blocks, we use a modified vector quantization algorithm. It is a more advanced algorithm than the conventional vector quantization algorithms. Figure 3.4 illustrates the difference between the normal LBG-algorithm and the modified algorithm. The advantages of the modified version come from the way it gets the initial code-book. Normal algorithm, also called LBG-algorithm, recursively splits data set to get the code-book. Figure 3.4a shows the result of normal LBG-algorithm. It selects the geometric center of the entire input vectors as a first entry in the code-book, then split another entry by randomly setting an offset to the first entry. The two entries are then passed to the LBG-algorithm and processed. It stops until convergence is reached. The same splitting then apply to the 2 entries to generate 4 entries. The whole algorithm ends until the desired bit-size is achieved, where the bit-size determine the size of the code-book. A bit-size of 8 means that a 256 length code-book is used. The normal procedure involves the process of finding nearest neighbors for n dimensional vector. It is very time consuming and the recursive splitting technique requires repeating the process several times. Hence,

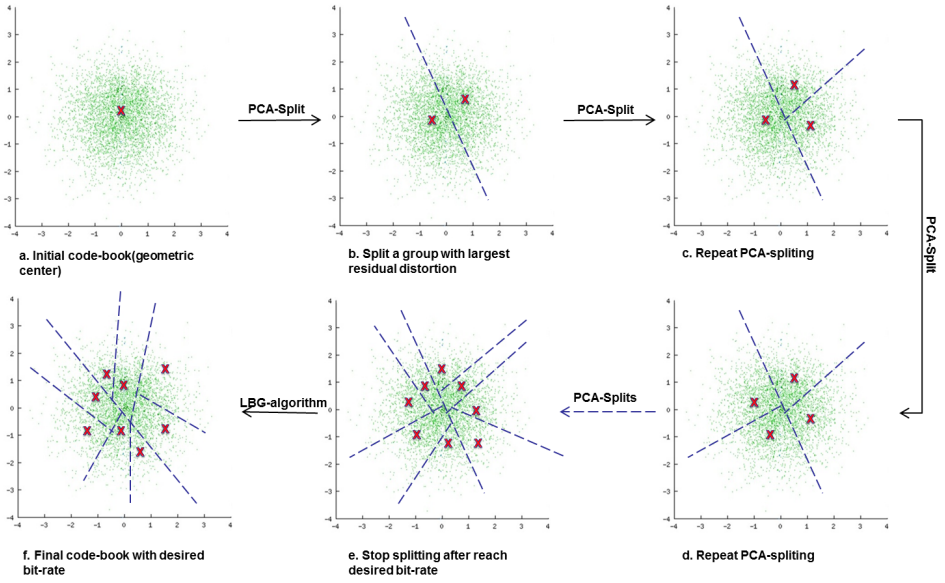
the LBG-algorithm is extremely slow in terms of speed. Pauly et al. [41] proposed a splitting technique when they addressed hierarchical clustering problems. The modified algorithm uses the same technique to split the original data as illustrated in Figure 3.4b. Like the normal splitting, it also starts with a single entry in the code book. Instead of getting another entry by random offset, it divides the entry into two entries roughly at the same distortions. The next splitting continues with the entry with the largest distortion. Only the chosen entry splits into two smaller entries and each small entry has a smaller distortion. Only one more entry is generated after each split. The splitting process will stop until the desired number of entries are reached. The generated initial code-book is further refines by normal LBG-algorithm to get the final code-book. This splitting technology avoids significant number of time-consuming nearest neighbors search operation. Moreover, it also solve the empty cell problem which is common in the normal LBG-algorithm. Empty cell is not desired as it significantly increases the compression distortion. A more detailed description of the modified vector quantization algorithm used can be found in Schneider's paper [25].

After vector quantization, we now have two code-books and a group of RGB color triplets. As shown in Figure 3.2, we denote the two code-books as code-book A and code-book B. Code-book A is the code-book for difference at level 0, which contains 64-dimensional arrays. Originally, there are $D_x\prime/4 * D_y\prime/4 * D_z\prime/4$ arrays in total for the difference level 0. Those vectors are approximated by 256 different vectors after vector quantization. The *R* element in the RGB color triplets records the index in the code book of corresponding vector. The index is a 8-bits variable. The code-book B is the code-book for detail level 0. The same vector quantization applies to detail level 0 and results with 256 8-dimensional arrays. The index of corresponding array is stored in the G element in the RGB color triplets. The last element in the RGB color triplets stores the mean value of the corresponding 4^3 block. There are $D_x\prime/4 * D_y\prime/4 * D_z\prime/4$ such RGB color triplets in total. These triplets, as well as the code-books A and B, are then saved to the hard drive for latter use.

To save the results of the preprocessing step, we need to define an appropriate



(a) LBG vector quantization



(b) Vector quantization using principal component analysis (PCA)

Figure 3.4: Procedures of two different vector quantization algorithms

file format. The format must contain enough information for the visualization algorithm. In our system, we use a file header to store the information which is required later. As our visualization algorithm only needs information about the dimension of the volumetric data, the file format which we use is defined in Figure 3.5. The first component in the header is a 8-bits flag to indicate the endianness of the file. The second component gives the size of an integer in bytes. Different computers may have different integer size. By storing the size of integer into the file header, the file is more machine-independent. Mode is a variable to indicate if the file is compressed from the difference or is the original data. We define 1 as compressed from the difference of two time steps and 0 as the original data. We will discuss more about the reason for this variable later. The size of volumetric data on each dimension is also very important. We save $D_x\prime/4 * D_y\prime/4 * D_z\prime/4$ into sizeX, sizeY and sizeZ respectively. The scaleDiff0 and scaleDetl0 are the largest absolute values in difference level 0 and detail level 0. The code book contains code book A and code book B, which consume $256 * (64 + 8)$ bytes. The RGB color triplets, as its name indicates, stores all the $D_x\prime/4 * D_y\prime/4 * D_z\prime/4$ triplets. We use l3vq as the file extension for the compressed file, where "l3" stands for 3 levels (difference 0, detail 0 and detail 1 level) and "vq" stands for vector quantization. As stated previously, even though the modified vector quantization speeds up the quantization process significantly, it still takes a long time to calculate.

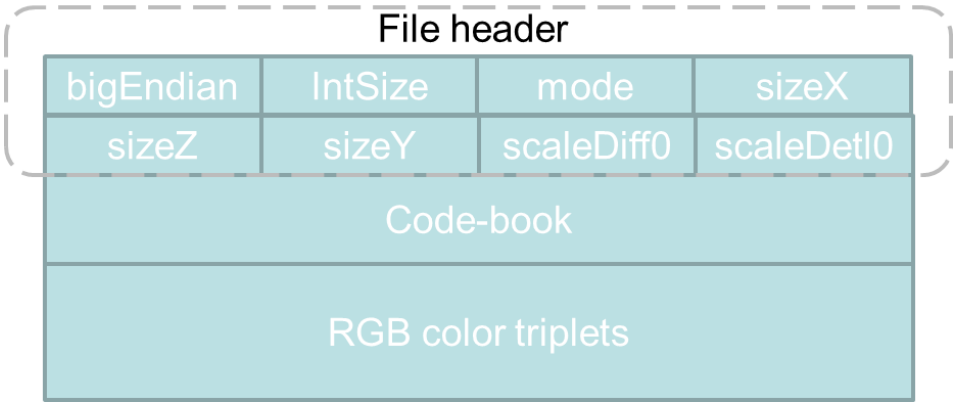


Figure 3.5: The definition of l3vq file format.

3.3 Decoder Module

The encoder preprocessing ends after the resulting files are saved on the hard drive. The first thing the decoder module needs to do is to read those saved files from the hard drive to main memory. In our implementation the file reader parses the file stream according to the file format definition. The code books and the RGB color triplets are parsed and saved in main memory.

As mentioned previously, the bottleneck of real-time rendering of large time-varying volumetric data is the data exchange between the main memory and the GPU memory. Our solution is to move the decoding process to the GPU and then minimize the transfer to the GPU by using the compression scheme. The next step for the decoder module is to transfer the code-books and RGB triplets into GPU memory. This step is harder than it looks. Because the GPU memory model is not as flexible as the main memory model. There are certain data structures which GPU memory model does not support. For a more detailed introduction of GPU memory model, please refer to Lefohn et al's course [36]. The most widely used memory in GPU are texture memory. This memory is fast and optimized for related texture operations. We use texture memory in GPU to store the code-books and the triplets from main memory for a time-step. The decoder module generates 3D and 2D textures in the GPU first. The code books and RGB color triplets in main memory are then uploaded to these generated textures. Code book A and code book B are stored in two different 2D textures C_a and C_b. RGB color triplets are stored in a 3D texture I. The 2D textures C_a and C_b have $256 * 64$ and $256 * 8$ elements, respectively. To access the element in 2D textures, 2-dimensional coordinates (s, t) are required. The s coordinate of C_a and C_b are given by the R and G components in RGB color triplets, which are now stored in the 3D texture I. Figure 3.6 shows the overview of different textures for the decoder module.

Before we introduce how to implement the GPU decoding scheme, let us take a look at the equation used by the GPU in order to decode the data. In its simplest form, the equation is:

$$w_{(x,y,z)} = I_{(x\prime,y\prime,z\prime)}.b + C_{b(s,w)} + C_{a(s,w)}. \tag{3.1}$$

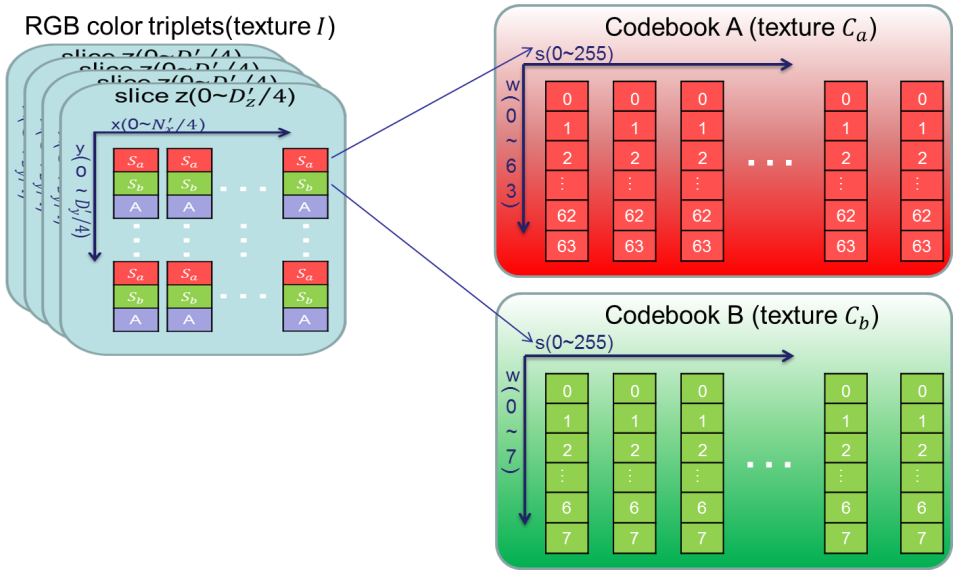


Figure 3.6: The textures in GPU for decoder module.

In this equation, $w_{(x,y,z)}$ is the scalar value at location (x, y, z); $I_{(x\prime,y\prime,z\prime)}$.b is the B component of the the RGB color triplets, which is also the mean value of the corresponding 4^3 block. Each (x, y, z) must belong to one of these blocks when we decomposed the volumetric data in the preprocessing step. The parameters $C_{b(s,w)}$ and $C_{a(s,w)}$ are the values which are returned by the code book B and code book A, respectively. To solve the equation, we need to somehow map all the coordinates on the right side of equation 3.1 to a function of (x, y, z), because (x, y, z) are the only coordinates we know during decoding. The main obstacle to implement a working decoding module is how to relate the different coordinates for different textures. The (x, y, z) coordinates are the actual coordinates in the re-sampled data. The coordinates $(x\prime, y\prime, z\prime)$ is for the texture I and needs to be calculated from (x, y, z). The s coordinates can be directly calculated from the returned values of texture I. The w coordinates of texture C_a and C_b are related to (x, y, z) as well.

The relationship of (x, y, z) and $(x\prime, y\prime, z\prime)$ is not difficult to find. Texture I is essentially a lower resolution of the original data. Each dimension of the original data is divided by 4 to get the dimension of texture I. And only the integer part of the quotient is used. So the transformation between (x, y, z) and $(x\prime, y\prime, z\prime)$ can be expressed by:

$$
\begin{array}{l}
x\prime = \lfloor x/4 \rfloor, where\ 0 \leq x\prime \leq (D_x\prime/4 - 1)\ and\ 0 \leq x \leq (D_x\prime - 1) \\
y\prime = \lfloor y/4 \rfloor, where\ 0 \leq y\prime \leq (D_y\prime/4 - 1)\ and\ 0 \leq y \leq (D_y\prime - 1) \\
z\prime = \lfloor z/4 \rfloor, where\ 0 \leq z\prime \leq (D_z\prime/4 - 1)\ and\ 0 \leq z \leq (D_z\prime - 1)
\end{array} \tag{3.2}
$$

The previous equations only deal with integer numbers. In the implementation, we need to use texture mapping coordinate. Normal texture coordinates are not integers. We need to further transform the integer coordinates to float coordinates between 0.0 and 1.0 in the actual implementation.

The relationship between (x, y, z) and the s, w coordinates of C_a and C_b are less intuitive though. Let us examine the case for C_a first. The s coordinate is directly given by the R component of the RGB color triplets as illustrated in Figure 3.6. The w coordinate is an integer between 0 and 63. It is essentially the local index of (x, y, z) in the corresponding 4^3 block. The reminder of (x, y, z) divided by 4 can be used to calculate the local index. So the coordinate w and s can be computed by the following equations:

$$
(s, w) = \begin{cases} s = I_{(x\prime,y\prime,z\prime)}.r \\ w = x\%4 + 4 * (y\%4) + 16 * (z\%4) \end{cases} \tag{3.3}
$$

where $I_{(x\prime,y\prime,z\prime)}.r$ represents the R component of the return value from the texture$I$ sampling. The same equations applies to C_b as well. The only difference is that one needs to find the local index of corresponding 2^3 block instead of 4^3 block. The equation for C_b is showed in Equation 3.4.

$$
(s, w) = \begin{cases} s = I_{(x\prime,y\prime,z\prime)}.g \\ w = \lfloor \dfrac{x\%4}{2} \rfloor + 2 * \lfloor \dfrac{y\%4}{2} \rfloor + 4 * \lfloor \dfrac{z\%4}{2} \rfloor \end{cases} \tag{3.4}
$$

Now we have all the mapping equations. We need one equation to put them all together. The generalized equation should only contains coordinates of (x, y, z). Equations 3.2, 3.3 and 3.4 can be inserted into Equation 3.1 to get Equation 3.5.

$$
\begin{array}{rcl}
w_{(x,y,z)} & = & I_{(\lfloor x/4 \rfloor, \lfloor y/4 \rfloor, \lfloor z/4 \rfloor).b} \\
& & + C_{b(I_{(\lfloor x/4 \rfloor, \lfloor y/4 \rfloor, \lfloor z/4 \rfloor).g, \lfloor \frac{x\%4}{2} \rfloor + 2 * \lfloor \frac{y\%4}{2} \rfloor + 4 * \lfloor \frac{z\%4}{2} \rfloor)} \\
& & + C_{a(I_{(\lfloor x/4 \rfloor, \lfloor y/4 \rfloor, \lfloor z/4 \rfloor).r, x\%4 + 4*(y\%4) + 16*(z\%4))}
\end{array} \tag{3.5}
$$

where (x, y, z) are integer values ranging from $[0, D_x\prime - 1]$, $[0, D_y\prime - 1]$ and $[0, D_z\prime - 1]$ respectively.

Equation 3.5 is the final equation which represents the whole decoding process related to the coordinates (x, y, z). Our decoder module needs a program in GPU to solve the equation efficiently. The GPU programming language which we choose to program the GPU is Cg from NVIDIA. One could have use CUDA instead but our program would have been usable only on NVIDIA hardware. This restriction would make our solution not portable to other GPU manufacturer such as ATI. Unlike CUDA, Cg can compile to GLSL for ATI GPUs. It enables us to deploy our proposed system on ATI machines as well. The decision to use Cg to implement the decoding module gives our system a better portability. To implement Equation 3.5 in GPU with Cg, there are some limitations which we must bear in mind:

1. The shift operations are not supported by Cg. In general programming, it is easy to get the results of $\lfloor x/4 \rfloor$ by shift operation $x >> 2$. Unfortunately, one cannot use the operation in Cg.

2. The bitwise operations are not supported by Cg. One efficient way to calculate $x\%4$ is to use bitwise AND operation. It is essentially the same as $x\&3$.

3. The texture coordinates in Cg are not integer. All our equations are based on an assumption that (x, y, z) are integers. We need to find a way to map the integer to float.

In the following, we will introduce ways to deal with those limitations. At first, let us take a look at the setup of viewing parameters for the decoding program. The projection mode is an orthogonal projection; and the view port is of size $D_x * D_y$. The volumetric data is reconstructed slice by slice. Either front-to-back or back-to-front order is fine for reconstruction. The rendering results are saved in the frame buffer for binding. Figure 3.7 is a pseudo-code to generate the coordinates (x, y, z). Instead of integers, the coordinates are in the $[-1.0, 1.0]$ range. To convert to the new float coordinate, the integer is divided by $N_x - 1$, $N_y - 1$ or $N_z - 1$ and then subtract by 1.0 as in $x/(N_x - 1) - 1.0$.

To use the (x, y, z) coordinates in the GPU, there are still two problems remaining for us to solve. The first problem is to figure out a way to get coordinates

```
for (int iZ=0; iZ<iSlices; iZ++) {
    m_cDecodeBuffer->Slice(iZ);
    glBegin(GL_QUADS);
        glVertex3f(+1.0f,+1.0f,2.0f*float(iZ)/float(iSlices-1)-1.0f);
        glVertex3f(+1.0f,-1.0f,2.0f*float(iZ)/float(iSlices-1)-1.0f);
        glVertex3f(-1.0f,-1.0f,2.0f*float(iZ)/float(iSlices-1)-1.0f);
        glVertex3f(-1.0f,+1.0f,2.0f*float(iZ)/float(iSlices-1)-1.0f);
    glEnd();
}
```

Figure 3.7: The coordinates which are used in the decoder implementation.

$(x\prime, y\prime, z\prime)$ for texture I from (x, y, z) without a shift operation or a direct floor function. The essence of the function $\lfloor x/4 \rfloor$ is to map four continuous integers to the same value. In texture mapping, there is a GL_NEAREST parameter for GL_TEXTURE_MAG_FILTER. When the pixel being textured maps to an area less than or equal to one texel, texture mapping returns the value of the texture element that is nearest (using the Manhattan distance) to the center of the pixel being textured. This way the nearest operation also maps different value to a single value. This gives us a way to overcome the first problem. Say we want to calculate the results of function $\lfloor x/4 \rfloor$ when $x = 4, 5, 6, 7$. If we can find a way to convert these integers to texture coordinates in a specific range and use the GL_NEAREST feature to texture map them, we are able to map all the texture coordinates to the same texture element. Thus, the function $\lfloor x/4 \rfloor$ can be simulated by texture mapping operation. Back to our application domain, one can prove that the following function yields the correct coordinates for texture I:

$$\begin{aligned} x\prime &= (x * 0.5 + 0.5) * \frac{D_x - 1}{D_x - 4} - \frac{3}{2 * D_x - 8} \\ y\prime &= (y * 0.5 + 0.5) * \frac{D_y - 1}{D_y - 4} - \frac{3}{2 * D_y - 8} \\ z\prime &= (z * 0.5 + 0.5) * \frac{D_z - 1}{D_z - 4} - \frac{3}{2 * D_z - 8} \end{aligned} \tag{3.6}$$

where $D_{(x,y,z)} = \{2^n \mid n >= 3\}$. When the coordinates are less than 0.0 or greater than 1.0, we use GL_CLAMP to clamps the texture coordinate into the $[0.0, 1.0]$ range. The clamping function handles the problem associated with texture border.

The second problem is to get the w coordinates for each small block. Address texture, as proposed by Schneider et al. [25], is a good solution for this problem. The basic idea is to use the address texture to hold the w coordinates of one single

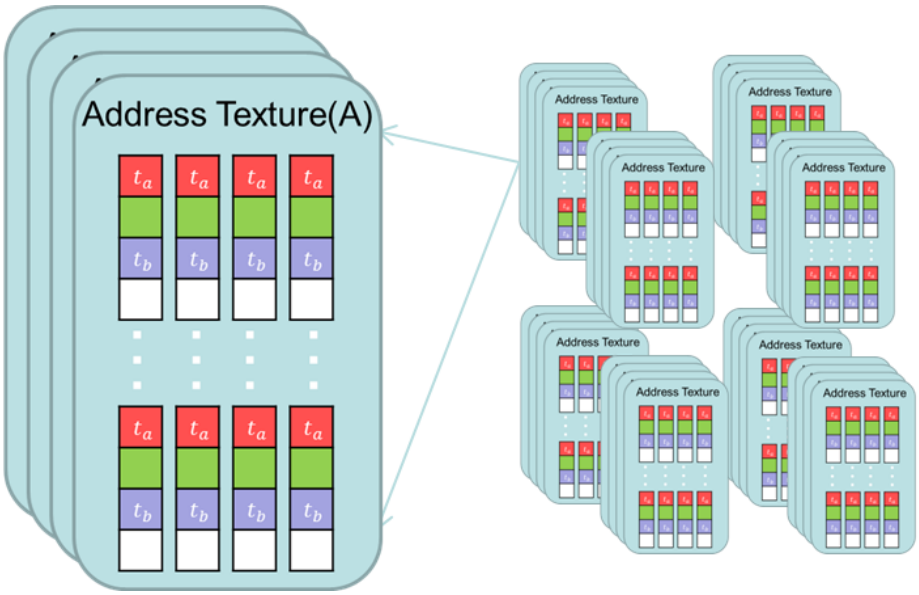


Figure 3.8: The structure and repeat pattern of address texture.

block and reuse it for other blocks. If we examine every block individually, the w coordinates of C_a and C_b for these blocks have the same pattern. So a single address texture which stores the pattern is enough for decoding all blocks. In our application domain, the address texture is a 4^3 block. We denote it texture A. Figure 3.8 illustrates the structure of address texture. Each element in the address texture has four components, RGBA. The R and B components store the w coordinates of the code-books A and B. The G and A component are reserved for s coordinates which come from the R and G component of texture I. To get the pattern for the texture address, let us assume the size of original data is the same as address texture. It is easy to calculate the w coordinates of C_a and C_b from equation 3.5. One can use a progressively larger integer in every dimension to replace (x, y, z). The integer

value should be between 0 and 3. It is easy to get the following results:

$$w_a = \begin{pmatrix} \frac{0}{63} & \frac{1}{63} & \frac{2}{63} & \frac{3}{63} \\ \frac{4}{63} & \frac{5}{63} & \frac{6}{63} & \frac{7}{63} \\ \frac{8}{63} & \frac{9}{63} & \frac{10}{63} & \frac{11}{63} \\ \frac{12}{63} & \frac{13}{63} & \frac{14}{63} & \frac{15}{63} \\ \frac{32}{63} & \frac{33}{63} & \frac{34}{63} & \frac{35}{63} \\ \frac{36}{63} & \frac{37}{63} & \frac{38}{63} & \frac{39}{63} \\ \frac{40}{63} & \frac{41}{63} & \frac{42}{63} & \frac{43}{63} \\ \frac{44}{63} & \frac{45}{63} & \frac{46}{63} & \frac{47}{63} \end{pmatrix} \begin{pmatrix} \frac{16}{63} & \frac{17}{63} & \frac{18}{63} & \frac{19}{63} \\ \frac{20}{63} & \frac{21}{63} & \frac{22}{63} & \frac{23}{63} \\ \frac{24}{63} & \frac{25}{63} & \frac{26}{63} & \frac{27}{63} \\ \frac{28}{63} & \frac{29}{63} & \frac{30}{63} & \frac{31}{63} \\ \frac{48}{63} & \frac{49}{63} & \frac{50}{63} & \frac{51}{63} \\ \frac{52}{63} & \frac{53}{63} & \frac{54}{63} & \frac{55}{63} \\ \frac{56}{63} & \frac{57}{63} & \frac{58}{63} & \frac{59}{63} \\ \frac{60}{63} & \frac{61}{63} & \frac{62}{63} & \frac{63}{63} \end{pmatrix}$$

$$w_b = \begin{pmatrix} \frac{0}{7} & \frac{0}{7} & \frac{1}{7} & \frac{1}{7} \\ \frac{0}{7} & \frac{0}{7} & \frac{1}{7} & \frac{1}{7} \\ \frac{2}{7} & \frac{2}{7} & \frac{3}{7} & \frac{3}{7} \\ \frac{2}{7} & \frac{2}{7} & \frac{3}{7} & \frac{3}{7} \end{pmatrix} \begin{pmatrix} \frac{0}{7} & \frac{0}{7} & \frac{1}{7} & \frac{1}{7} \\ \frac{0}{7} & \frac{0}{7} & \frac{1}{7} & \frac{1}{7} \\ \frac{2}{7} & \frac{2}{7} & \frac{3}{7} & \frac{3}{7} \\ \frac{2}{7} & \frac{2}{7} & \frac{3}{7} & \frac{3}{7} \end{pmatrix} \begin{pmatrix} \frac{4}{7} & \frac{4}{7} & \frac{5}{7} & \frac{5}{7} \\ \frac{4}{7} & \frac{4}{7} & \frac{5}{7} & \frac{5}{7} \\ \frac{6}{7} & \frac{6}{7} & \frac{7}{7} & \frac{7}{7} \\ \frac{6}{7} & \frac{6}{7} & \frac{7}{7} & \frac{7}{7} \end{pmatrix} \begin{pmatrix} \frac{4}{7} & \frac{4}{7} & \frac{5}{7} & \frac{5}{7} \\ \frac{4}{7} & \frac{4}{7} & \frac{5}{7} & \frac{5}{7} \\ \frac{6}{7} & \frac{6}{7} & \frac{7}{7} & \frac{7}{7} \\ \frac{6}{7} & \frac{6}{7} & \frac{7}{7} & \frac{7}{7} \end{pmatrix}.$$

Note that we have converted the coordinates into the legal range value of $[0.0, 1.0]$ in the equation. Now we need a mechanism to use the address texture repeatedly. Fortunately, there is a GL_REPEAT parameter in texture mapping. It creates a repeating pattern by ignoring the integer portion of the texture coordinate. With the help of GL_REPEAT feature, the w coordinates calculations is reduced to a simple texture sampling operation. For every dimension, we need to repeat the texture coordinates $D_x/4$, $D_y/4$ and $D_z/4$ times respectively. It means that the texture coordinates for the texture address should be ranging from 0 to $D_x/4$, $D_y/4$ and $D_z/4$. A multiplication of (x, y, z) with $(D_x/4, D_y/4, D_z/4)$ will do the job. Equation 3.7 describes the transformation:

$$\begin{aligned} x_A &= (x * 0.5 + 0.5) * (D_x/4) \\ y_A &= (y * 0.5 + 0.5) * (D_y/4) \\ z_A &= (z * 0.5 + 0.5) * (D_z/4) \end{aligned} \quad . \tag{3.7}$$

where x_A, y_A and z_A represent the texture coordinates for accessing texture A. Figure 3.8 illustrates the case where the texture address repeats itself with texture coordinates ranging from $[0.0, 2.0)$. Together with the R and G components of texture I, Equation 3.5 describes the decoding process mathematically as performed

on the GPU. And the GPU customized version of Equation 3.5 is:

$$\begin{aligned} w_{(x,y,z)} &= I_{(x\prime,y\prime,z\prime)}.b \\ &\quad +C_{b(I_{(x\prime,y\prime,z\prime)}.g,A_{((x*0.5+0.5)*(D_x/4),(y*0.5+0.5)*(D_y/4),(z*0.5+0.5)*(D_z/4))}.b)} \\ &\quad +C_{a(I_{(x\prime,y\prime,z\prime)}.r,A_{((x*0.5+0.5)*(D_x/4),(y*0.5+0.5)*(D_y/4),(z*0.5+0.5)*(D_z/4))}.r)} \end{aligned} \tag{3.8}$$

where $x\prime = (x * 0.5 + 0.5) * \frac{D_x-1}{D_x-4} - \frac{3}{2*D_x-8}$, $y\prime = (y * 0.5 + 0.5) * \frac{D_y-1}{D_y-4} - \frac{3}{2*D_y-8}$ and $z\prime = (z * 0.5 + 0.5) * \frac{D_z-1}{D_z-4} - \frac{3}{2*D_z-8}$.

With this equation, it is not hard to write its implementation in Cg. Because our decompressed data is the difference between two continuous time-varying volumetric data, there is one more step before rendering the volumetric data. We need to add the decompressed data with previously reconstructed volumetric data to yield the current reconstructed data. The addition of two volumetric data is also easy to implement using Cg.

The vector quantization algorithm is not a lossless compression algorithm. The bit-size of the index in the code-book affects the compression quality significantly. The bigger bit-size the lower the compression distortion is but at the expense of compression. It is possible to control the compression quality by setting up different bit-size. In this thesis, we use a bit-size of 8.

The initial time step of the time-varying volumetric data is transferred directly into GPU as a start point. Then for the subsequent time steps, we reconstruct the difference and add them to the previous time step. One problem associated with this scheme is the accumulated error. After several time steps, the error accumulated to a larger value. As in video compression, we need to use the concept of an I frame and P frame mechanism which is to reduce the accumulated errors. The I frames do not require other frames to be decoded. In our application domain, we define the original data as the the I frames. The P frames need previous data for decoding. We define the reconstructed volume data as the P frames. After several time steps, we transfer one copy of the original time step to GPU memory as an I frame to reset the accumulated errors.

3.4 Render Module

The render module is responsible for the real-time rendering of volumetric data. Numerous rendering algorithms have been proposed by researchers in the past decades. Generally speaking, they can be divided into two main categories: direct volume rendering and indirect volume rendering. Indirect volume rendering is also referred to as surface rendering. Surface rendering requires a set of iso-surfaces to approximate the surface with polygons. The fidelity of the rendering result is not very satisfactory in some cases. Direct volume rendering is more flexible and is an accurate way as it does not suffer from the artifacts created by the segmentation process. Kaufman et al. [31] give an excellent tutorial on this topic. Further more, direct volume rendering algorithms have three different categories: object-order, image-order, and domain methods. The concepts of object-order and image-order are quite common in computer graphics. Image-order describes the action of iterating over the pixels in the image to be produced; and object-order characterizes the algorithms which iterate over the object elements in the scene to be rendered. Object-order algorithms are generally more efficient than image-order algorithms because the objects usually have fewer elements than image pixels.

We use 3D texture-based volume rendering in our system. It is an object-order volume rendering algorithm because it uses geometry primitives to iterate over the object. The whole rendering process is usually composed of the following steps:

1. Bind the volumetric data as 3D textures. The decoder module in our system did this step for the render module already.

2. Setup intractable rendering parameters, such parameters include: camera position, view direction, view port, number of slices and rendering mode and so on.

3. Calculate a series of slices which intersect with objects in the scene. All the slices should be perpendicular to the viewing direction and have the same distance between two neighbor slices.

4. Render each slice using a 3D texture mapping operation and blend them to

one final image.

5. Respond to any change of interaction parameters and repeat step 2 to step 5.

The first and second steps set-up all parameters which are required for the rendering algorithm. The most time-consuming process in the rendering algorithm starts from step 3. The calculation of cut-off slices are usually performed by CPU. Rezk-Salama et al. [50] proposed a way to move the calculation to the GPU using vertex shader. Vertex shader is designed to transform the 3D position of each vertex to a 2D coordinate on the image plane. It can calculate the properties such as position, color and texture coordinates. The cut-off slice generation is essentially a transformation of vertex. It is not surprising to have a vertex shader deals with the calculation. The implementation is not as intuitive as it sounds though. The intersection between the object and the slicing plane results in a polygon with 3 to 6 vertices. The vertex shader, however, can only modify existing vertices. No inserting or removing of any vertex can be done by the vertex shader. As a result, an algorithm which receives 6 vertices and outputs 6 vertices for every intersection polygon must be designed. To represent polygons with less than 6 vertices, some duplicate vertices must be generated in the algorithm. However, in order to feed fragment shader with correct vertices, the algorithm must get rid of these duplicate or invalid vertices when rendering. The fragment shader also requires the vertices form a closed polygon. The vertices must be sorted before they can be transferred to fragment shader. To achieve all these requirements, let us take a look at the intersection algorithm from a mathematics perspective.

In essence, the problem is to find the intersection between a slicing plane and the border of volumetric data. The border of volumetric data usually is a bounding box which is a cube wrapping the whole object. The bounding box has 12 straight edges. The edge $E_{i\to j}$ between two vertices V_i and V_j can be represented as:

$$\overrightarrow{x} = \overrightarrow{V}_i + \lambda * \overrightarrow{e}_{i\to j} \tag{3.9}$$

where λ in the range of $[0, 1]$ and $\overrightarrow{e}_{i\to j}$ is the subtraction of $\overrightarrow{V}_j$ and $\overrightarrow{V}_i$, also $\overrightarrow{V}_j - \overrightarrow{V}_i$. The clipping plane can be described in Hessian normal form as:

$$\overrightarrow{n}_p \cdot \overrightarrow{x} = d \tag{3.10}$$

where $\overrightarrow{n}_p$ is the normal vector of the plane and d is the distance to the origin. The 3D texture based rendering algorithm usually uses view-port aligned slice planes. In this case, $\overrightarrow{n}_p$ is also the viewing direction. Using Equation 3.9 and Equation 3.10, one can calculate λ:

$$\lambda = \begin{cases} \frac{d - \overrightarrow{n}_p \cdot \overrightarrow{V}_i}{\overrightarrow{n}_p \cdot \overrightarrow{e}_{i \to j}}, \ \overrightarrow{n}_p \cdot \overrightarrow{e}_{i \to j} \neq 0 \\ -1.0, \ otherwise \end{cases} \tag{3.11}$$

When λ is in the range $[0, 1]$, the intersection point x is a valid intersection. An invalid intersection denotes the intersection point is not on the edge of bounding box. When the denominator is zero, this means that the edge $\overrightarrow{e}_{i \to j}$ is on the clipping plane. We use -1.0 to denote this case.

Rezk-Salama et al. [50]'s implementation to calculate λ has a limitation, the nearest vertex must correspond to the origin point of texture coordinate system. Obviously, this is not always true. Rui [56] gives a more flexible coordinate translation equation. It basically translates the coordinate in object space to texture space. We adopted this more flexible algorithm in our system. The basic idea of forming a valid polygon from intersections of these two algorithms is the same. By checking the intersection point in an specific order, the vertex shader can output a valid polygon without any further steps. The edges of the cube can be grouped to 6 groups as showing in figure 3.9. We distinguish different groups by different colors. Vertex 0 is the closest one near the view point(camera). And Vertex 7 is the most far away one. The three edges connecting vertex 0 to vertex 7, which are $E_{0 \to 1}$, $E_{1 \to 4}$ and $E_{4 \to 7}$, has only one intersection point with any clipping plane. In the picture, these edges are marked as red color. The same situation apply to blue edges group ($E_{0 \to 3}$, $E_{3 \to 6}$ and $E_{6 \to 7}$) and green edges group ($E_{0 \to 2}$, $E_{2 \to 5}$ and $E_{5 \to 7}$). Each of the remaining edges ($E_{1 \to 5}$, $E_{2 \to 6}$ and $E_{3 \to 4}$) has one or zero intersection point with clipping plane. The correct order to form a valid polygon is red, yellow, green, cyan, blue and purple. If the yellow, cyan and purple edges do not have an intersection, a duplicated red, green or blue intersection will be generated in this case. For example, a clipping plane has 4 intersections. 3 out of 4 must be intersections with red, green and blue edges respectively. Suppose the last intersection is on the yellow edge. Then the cyan and purple edges have no valid intersection with this

clipping plane. The algorithm duplicates the intersections of green and blue edges for cyan and purple respectively. The 4 intersections are therefore represented by 6 intersections with 2 duplicated intersections. The output vertices still form a valid quadrangle. In the implementation, the duplications are achieved by recalculation. If the yellow edge does not have an intersection, all the red edges must be feedback to the algorithm to duplicate the red intersection.

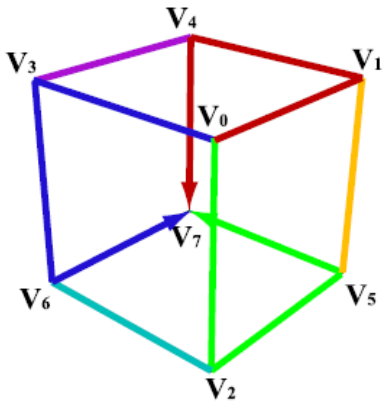


Figure 3.9: The traversal of edges for the intersection algorithm. Taken from [56]

These calculations are carried out in the object space. To translate the coordinates to texture space, the equation given by Shen et al. [56] is the following:

$$P\prime \cdot \vec{k} = \begin{cases} \frac{V_i \cdot \vec{k} - min(B_{\vec{k}})}{max(B_{\vec{k}}) - min(B_{\vec{k}})} & \vec{k} \cdot \overrightarrow{e}_{i \to j} = 0 \\ \lambda & \vec{k} \cdot \overrightarrow{e}_{i \to j} > 0 \\ 1 - \lambda & \vec{k} \cdot \overrightarrow{e}_{i \to j} < 0 \end{cases} \tag{3.12}$$

where $P\prime$ is the corresponding coordinate in texture space; $\vec{k}$ is x, y or z unit vector in texture space; $B_{\vec{k}}$ is the length of bounding box B in the $\vec{k}$ direction. The first case in the equation represents the case when the edge $\overrightarrow{e}_{i \to j}$ is perpendicular to the unit vector $\vec{k}$. The results are either 0 or 1 in this case. When $\vec{k} \cdot \overrightarrow{e}_{i \to j} < 0$ is less than 0, it means that the direction of the edge $\overrightarrow{e}_{i \to j}$ is opposite to the direction of $\vec{k}$. Hence, the correct coordinate in texture space should be $1 - \lambda$. Using vertex shader for the calculation of slicing planes improves its performance significantly. Normally, the vertex shader only passes the vertices to fragment shader. This implementation balances the workload between the vertex shader and the fragment shader, which contributes to the performance of the entire rendering phase.

After the vertex finishes calculation, the results are passed to fragment shader, which evokes step three executed by the GPU. The primary operations in step three are two texture mapping operations. As showed in Figure 3.10, the first texture mapping slices the 3D texture using texture coordinates from the vertex shader. The resulting slice maps to the corresponding polygon in eye space. The second texture mapping also uses a transfer function. As most volumetric data contain only one scalar value at a specific position most of the rendering algorithms use a color lookup table to define the color and transparency of the scalar value. In essence, the color table is a transfer function. The corresponding implementation of a transfer function in GPU use a 2D texture. During rendering, the scalar value returned from the first texture mapping process is used as index to the look up function stores in a 2D texture. The returned value from second texture mapping process is usually a RGBA quadruplet. The transparency element A in the quadruplet will be used when blending all the slice together.

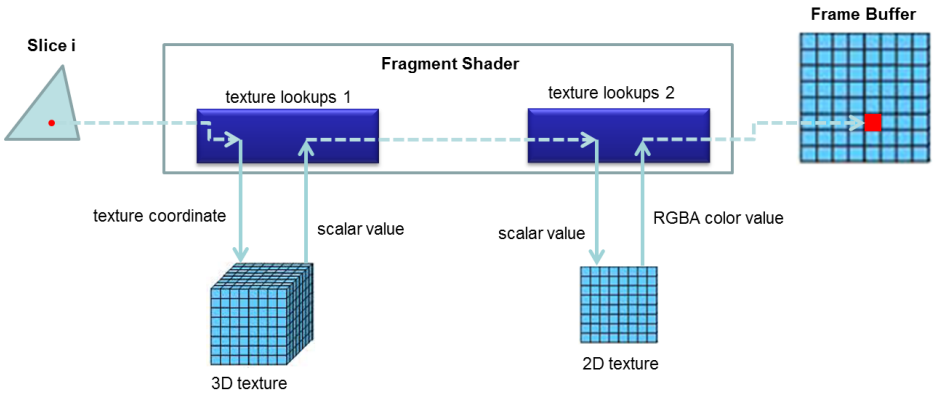


Figure 3.10: Texture mapping in fragment shader.

The texture mapping operations generate many quadruplets. The final step is to blend these quadruplets together. Each quadruplet has mapped to a position on screen. The final pixel color on screen is produced by the blending of all the corresponding quadruples at this pixel position. The transparency element in the quadruplet is the blending factor. Depending on the order of polygons filling in the fragment shader, the blending operation can be either in front-to-back or back-to-front order.

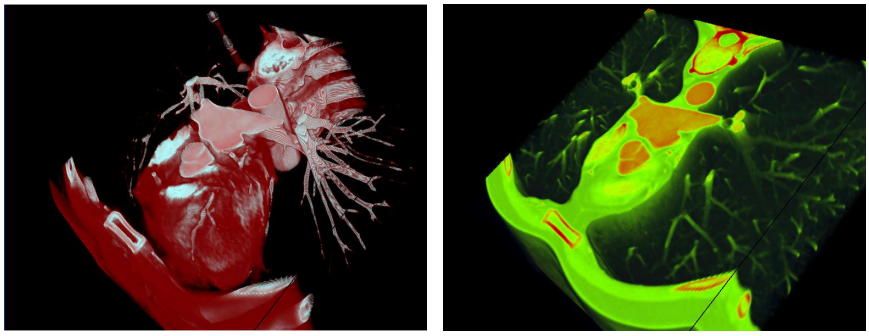
3.5 Interaction handler

Essentially, the interaction module takes the input from the keyboard or mouse, and translates the input parameters to the rendering algorithm. Any change capturing by interaction module will trigger the render module to update the rendering parameters and render the object once again. The real-time rendering constraints requires that the response time to the user input should be less than $100ms$ citeRTR3.

The type of interaction is only limited by imagination. It is impossible to include every interaction which an end user would expect. To have a sense of what kind of interaction is essential for our application, we refer to an open source software, ParaView [34]. This software is designed for 3D data visualization. It is a very famous software in the field of scientific visualization. Using interactions which are supported by ParaView as a guide line for our application is reasonable. The interactions for 3D volume data in ParaView include standard mouse interaction: rotation, moving and scaling. The end user can rotate the object arbitrarily when holding the left mouse button. Holding the middle button enables end user to move around the object. The right button is for scaling the rendering object. The method we choose for rotation is the Arc-ball. There are many different ways to allow user manipulate the viewpoint. Arc-ball considers to be the most intuitive method for end user. The Arc-ball model creates a sphere around the 3D object. End user can choose a point on the sphere by clicking. Dragging the selected point to another point on screen will do the rotation. The drawback of this method is the inability to rotate the object more than 180 degree in one clicking and dragging operations.

Other than mouse interaction, the ability to change the color look-up table at run time is also valuable. Figure 3.11 shows the rendering results of the same object with two different color tables. The left side picture looks much more realistic than the right side one. Especially for medical image data, a perceptually accurate and realistically colorized color table enhances the rendered data significantly. Our system uses a dedicated file to interactively change the color table. The interface is not very visually intuitive. But it is quite flexible. In order to change the color table, user needs to open the color map file and changes the value in the file manually. The

file is composed of several two-tuples and four-tuples. The two-tuples are parameters for vtkPiecewiseFunction class in VTK SDK. Let us denote any two-tuples as (x, a), where x represents the scalar value of volumetric data and a represents the corresponding opacity value. The vtkPiecewiseFunction takes these two-tuples and interprets the value between two sperate points linearly. The four-tuples are provided to vtkColormapPoint class. One element in the four-tuples represents the scalar value and the others represent a RGB color value. End users can add, edit or remove any value in the file. After saving the modified color table file, the final step is pressing "u" or "U" on keyboard to update the new file to render module.

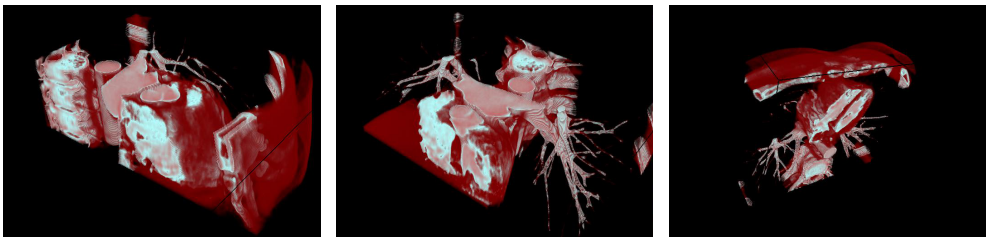


(a) (b)

Figure 3.11: Rendering results of the same volumetric data with different color tables.

Slicing the volumetric data is also a common need in volumetric data rendering, especially for medical image analysis. Our system has a GPU slicing feature. End user can slice the volumetric data along x, y or z axes. When slicing, all the mouse interactions are still available. Figure 3.12 displays the clipping results of the same object along x, y and z axes. By default, our system will render the volumetric data step by step. Using the acquisition time stored in volumetric data, the rendering algorithm can synchronize the time interval of rendering two different time step with the actual acquisition time interval. Pressing "p" or "P" on keyboard can pause the current rendering flow, which gives the end user enough time to look at the details at one specific time step.

Human vision has the ability to determine relative depths using several clues. One primary clue is called stereopsis. In visual perception, stereopsis describes the



(a) (b) (c)

Figure 3.12: Volumetric data clipping along x, y and z axes.

process leading to the illusion of depth from two slightly different projections of the world onto the retinas of the two eyes. In order to take the advantage of stereopsis, stereo rendering needs to generate two slightly different projections of the same object for left eye and right eye. The horizontal distance of two eyes are called binocular disparity. If we set up two cameras with appropriate horizontal distance in the scene which we are rendering, the generated images could be used to trick the human visual system. There are generally two different ways to deliver the 2D images to the corresponding eyes, time parallel and time multiplexed. Hodges et al. [20] give a good description of the two different methods. Most of the stereo rendering require end users to wear glasses. Some of the new technology, such as auto-stereoscopic displays, can create a 3D effect without having to rely on any extra devices [24]. But they either require expensive hardware or the view angle is very narrow. In our system, we implemented a side-by-side stereo mode. This method displays two slightly different images side-by-side. In our lab environment, we use two projectors to project each side of image onto the same silver screen. Each projector has a polarizing filter in front of it and the two filters are perpendicular to each other. The polarized filter only passes light which is similarly polarized and gets rid of the orthogonally polarized light. Our end user needs to wear an eyeglasses which also contain a pair of orthogonal polarizing filters. Each eye then only sees one of the projected image, and the stereo effect perceived. We also have a DTI 3D display in our lab. No glasses is required to view stereo effect. The left and right eye images are interlaced across the LCD screen by a special illumination plate which is placed behind it. Hence, the left eye and right eye zones are formed

in front of the screen. The viewer needs to find the sweet point for the best stereo effect. The drawback of this hardware is the narrow view zone. It is impossible to have several people view it at the same time. The final images through polarized filter or interlacing appear darker than normal, because only half of the light energy reaches human eyes.

The extensibility of our system allows users to add their own data filters easily. The separated interaction module allows to add independent filter with little work. However, since the reconstructed volumetric data is only available in GPU memory, any new data filter needs to be able to execute by GPU directly.

3.6 Implementation Details

Volumetric rendering is a well studied subject in computing science. There are numerous open source SDK on the internet. Our implementation carefully customizes some of them and puts them together to form a system. The software libraries which we have used in our system include the following:

1. **CMake**

   CMake [43] is a software build system which allows developers to develop cross-platform software. It can generate native makefiles for different system or workspaces for different integrated development environments(IDE). The only requirement is a simple configuration file. Developer needs to specify some build parameters in this file and CMake will take care of the rest. We have tested it with Microsoft Visual Studio 2008, Scientific Linux and Eclipse CDT(C/C++ Development Tooling). Our system compiled successfully for Windows and Linux. In other word, the portability of our system is improved by taking advantage of CMake.

2. **The Insight Toolkit(ITK)**

   ITK is developed by National Library of Medicine. Its primary usage is medical image segmentation and registration. A repository of fundament algorithms in this field can be found in this toolkit. A more detailed introduction

can be found in [22]. Our system uses ITK as a DICOM data reader. GDCM library (part of ITK) is a quite powerful and flexible in terms of reading raw DICOM data. The same series of DICOM files in a directory can be identified and stored in a ITK internal format for latter use. Besides the support for reading DICOM-format images, the ITK pipeline can be seamlessly connected to another toolkit which we selected for visualization.

3. **The Visualization Toolkit(VTK)**

   Before we introduce VTK, an other software named ParaView, which is based on VTK, is worthwhile to mention. Our initial idea of this project was to write a plug-in which specialize in large time-varying data volume rendering for ParaView. However, the plug-in mechanism is not as flexible as we thought. To make matters worse, we needed to use GPU for decompression. We then decided to implement a stand alone system with the help of VTK. VTK is a well-known open source C++ class library for 3D graphics. Object factories and virtual functions are widely used in the VTK implementation, which makes adding new class into the library easy. Our system uses some functions like color look-up table and volume rendering from the VTK toolkit.

3.7 Summary

By using the temporal redundancy between volumetric frames and spatial redundancy in each frame, we are able to reduce significantly the size of the data transfer between the central memory and the GPU memory. Essential to this scheme is the ability to perform decompression on the GPU as if the GPU was a client to a video server on the Internet. By using a simple quantization scheme and the concept of I frame and P frame borrowed from video compression, we were able to develop a decompression algorithm that can be perform in parallel on a GPU and is fast enough to not reduce the target frame rate of 10Hz. At the base of this implementation, we used a quantization algorithm proposed by Schneider [25]. As a courtesy, we were able to get access to the source code of Schneider's paper [25] allowing us to compare our GPU implementation to its counterpart. We then modified the

vector quantization and decompression algorithms to meet our needs and integrated them into our system. To the best of our knowledge, there is no standard benchmark for our proposed system. Schneider's work is the closest one that we were able to find. The volume rendering program is based on a trilinear interpolation scheme introduced in [56], which take advantage of the vertex shader. In the following section, we will analyze the speed and efficiency of the system. We will also show numerous examples of real-time rendering using data from a University of Alberta real-time CT machine.

Chapter 4

Experiments and Results

In order to test of our system, we use a desktop running Windows XP with a 4G memory Quadro FX 5800 graphics card. The interface between the GPU and main memory is PCI Express 2.0. The volumetric data which we use in our system is a series of chest CT scan slices. We thank Dr. Michelle Noga at department of Radiology & Diagnostic Imaging, University of Alberta, Edmonton, Canada, for providing us the data.

4.1 Data Preparation Time Measurement Results

The usefulness of our system relies on the assumption that the implemented GPU decompression mechanism saves time compare to direct data transfer through the PCI-e bus. Otherwise, there is no point to use our system when rendering large time-varying data set. The first experiment which we did is to get measurements of the data transmission time for direct transmission and our proposed method. As our proposed method involves transmission of I frame and P frame, we present the results in two tables. In both tables, t_1 represents the transfer time between the main memory and the GPU without compression and t_2 is the total time to transfer the compressed data from main memory to GPU and to decompress the data on the GPU. Table 4.1 is the results for I frame. We define I frame as the original data in this thesis. So t_1 is the same as t_2 and the compression ratio is 1. Table 4.2 shows the results for P frame transmission. In this table, p is defined as $(t_2 - t_2\prime)/t_2\prime$. It is essentially the time spend on decompression versus the time spend on transmitting

Data Dimension	$t_1(ms)$	$b(Gb/s)$	$t_2(ms)$	$S_o(Mb)$	$S_c(Mb)$	r
512x512x32	4.2	1.905	4.2	8	8	1
512x512x64	8.9	1.798	8.9	16	16	1
512x512x128	17.8	1.798	17.8	32	32	1
512x512x256	35.1	1.823	35.1	64	64	1
512x512x512	70.9	1.805	70.9	128	128	1

Table 4.1: The data preparation time and compression ratio for an I frame:t_1 Transfer time between the main memory and the GPU without compression of a I frame, b(Gb/s) The effective bandwidth, t_2 Total time to transfer the compressed data from main memory to GPU and to decompress the data on the GPU, $S_o(Mb)$ The original size in byte of the volume time step, $S_c(Mb)$ The compressed size in byte, and r The compression rate S_o/S_c. As we use the original data as I frame, $t_1 = t_2$ and the compression ratio is 1.

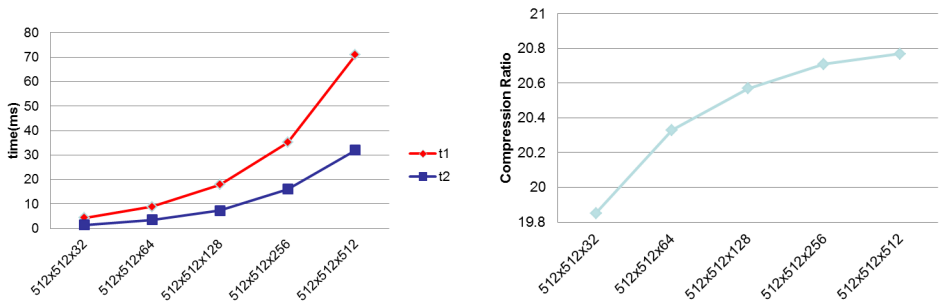
the compressed data. As the data size gets larger, more portion of time is dedicated to GPU decompression. These two examples are sufficient to justify the assumption which our system builds on. In essence, they are the amount of time it takes to get volumetric data ready for the rendering module. We call them *data preparation time* thereafter. If t_2 is less than t_1, it proves the GPU decompression mechanism is effective in terms of reducing GPU data waiting time. The two tables also contains the original data size(S_o), the compressed data size(S_c) and compression ratio(r). The compression ratio is defined by S_o/S_c.

We used five different data size ranging from 512x512x32 to 512x512x512 for the first test. They are actually the same data set. The original data dimension is 512x512x355. We performed up-sampling or down-sampling on the data to get different size. Trilinear interpolation is used on the data for different sampling rate. The experiment results for different data size help us understand how the system performs once we take into account the data transfer between the main memory and the CPU and the decompression on the GPU. Figure 4.1a shows the curves of data size versus data preparation time of P frame for the direct transfer without compression (red line) and our GPU decompression method (blue line). The GPU decompression method performs very well regardless of the data size. It saves more than half of the direct transferring time. When data size is small, it might not necessary to use GPU decompression method since the direct transfer time is already

Data Dimension	t_1	b	t_2	$t_2\prime$	p	S_o	S_c	r
512x512x32	4.2	1.905	1.4	0.2	6.00	8	0.403	19.85
512x512x64	8.9	1.798	3.5	0.4	7.75	16	0.787	20.33
512x512x128	17.8	1.798	7.3	0.9	7.11	32	1.555	20.57
512x512x256	35.1	1.823	16.1	1.7	8.49	64	3.091	20.71
512x512x512	70.9	1.805	32.0	3.4	8.42	128	6.163	20.77

Table 4.2: The data preparation time and compression ratio for a P frame: $t_1(ms)$ Transfer time between the main memory and the GPU without compression, $b(Gb/s)$ The effective bandwidth, $t_2(ms)$ The total time to transfer the compressed data (P frame) from main memory to GPU and to decompress the data on the GPU, $t_2\prime(ms)$ The compressed data (P frame) transmission time, p The time spend on decompression verses the time spend on transmitting the compressed data, $S_o(Mb)$ The original size in byte of the volume time-step, $S_c(Mb)$ The compressed size in byte, and r The compression rate S_o/S_c.

short. Moreover, the GPU decompression algorithm is not a lossless decompression. The speedup is not worthwhile compared to the sacrifice of image quality. However, when the data size is large, the speedup is a desired factor rather than image quality for real-time rendering. From the experiment result, we conclude that our assumption is correct and the GPU decompression mechanism is effective in terms of reducing the data preparation time. The compression ratio is illustrated in Figure 4.1b. The bit rate of the vector quantization in our test is 8. Hence, the upper bound of the compression ratio is around 21.3, which is verified by Figure 4.1b.



(a) Data preparation time of uncompressed data to the GPU vs and data preparation time with GPU decompression. (P frame)

(b) Compression ratio for a P frame.

Figure 4.1: System performance for single time step with a P frame

4.2 Volumetric Data Compression Experiment Results

As our system is optimized for time-varying data set, we also need to prove that our proposed method deals with time-varying data set better than the conventional none optimized method. The benchmark with which we selected to compare is based on Schneider's method. In their paper, they use mean square error(MSE), signal-to-noise ratio(SNR) and peak signal-to-noise ratio(PSNR) as performance indicator. They are defined as the following:

$$MSE = \sum_{i=1}^{n} (w_i - r_i)^2 / n \tag{4.1}$$

$$SNR = 10 * \log(\frac{\sum_{i=1}^{n} w_i^2 - (\sum_{i=1}^{n} w_i)^2 / n}{n * MSE}) / \log 10 \tag{4.2}$$

$$PSNR = 10 * \log(max(w_i)^2 / MSE) / \log 10 \tag{4.3}$$

where w_i is the original scalar value and r_i is the reconstructed value, n is the amount of scalar values and $max(w_i)$ is the maximal scalar value in the original data. We adopt the same performance measurements to evaluate our system. The data we use is of size 512x512x256 and there are 6 different time steps in total. Table 4.3, 4.4, 4.5 show results which we got from experiment. We did three different tests on these data. Table 4.3 is the benchmark test on Schneider's method. Their method treated different time steps individually and compressed them separately. Table 4.4 and table 4.5 are tests both based on our proposed idea of exploring time redundancy. The difference lies in the way they calculate the difference between two different time steps. In table 4.4, we get the difference from two original time steps; and in table 4.5 we subtract the current original time step with the reconstructed previous time step. For example, suppose we have time step 2 and 3, the second method would compress the difference of time step 3 and time step 2, while the third method would reconstruct time step 2 at first, and then calculate the difference of time step 3 and the reconstructed time step 2*I*. We call the second method the naive scheme and third method the progressive scheme.

Figure 4.2 gives a visual overview of the performance for the three methods. The green lines in the graph are the testing results for the benchmark method. As the different time steps are separated from each other, the compression algorithm

Time Step	1	2	3	4	5	6
MSE	15.6504	15.8601	15.1026	15.2743	15.4127	14.9541
SNR(dB)	24.06	24.11	24.20	24.22	24.21	24.21
PSNR(dB)	36.01	36.02	36.20	36.26	36.22	36.35

Table 4.3: Compression errors of our benchmark method. This method treated different time steps individually and compressed them separately.

Time Step	1	2	3	4	5	6
MSE	–	2.5064	12.4756	25.6068	44.2489	71.4553
SNR(dB)	–	32.12	25.03	21.98	19.63	17.42
PSNR(dB)	–	44.04	37.03	34.01	31.64	29.56

Table 4.4: Compression errors using the naive scheme. We compress the difference of two original time steps in this schema. The error accumulates very fast.

Time Step	1	2	3	4	5	6
MSE	–	2.5064	5.5752	5.7719	11.0158	20.5845
SNR(dB)	–	32.12	28.52	28.45	25.67	22.82
PSNR(dB)	–	44.04	40.53	40.48	37.68	34.96

Table 4.5: Compression errors using the progressive scheme. We compress the difference of original time step with its previous reconstructed time step. The error accumulates in a much slower pace than the naive scheme.

performance is quite stable. The green lines are almost horizontal in the graph. The naive scheme compression method is represented by the blue lines. As seen in the graph, the blue lines increase or decrease dramatically compared to other lines. The accumulated errors account for it. After reconstruction, the error due to compression is added to all previous errors, which also explains the monotone increase in MSE and monotone decreasing in SNR and PSNR. This is not the best method to compress the time-varying data set. As shown in Figure 4.2, the benchmark algorithm outperformed this method in every aspect after time step 3. The red lines, which represents the progressive scheme compression method, have much smoother slopes than the blue curve. It indicates that the compression error rate is smaller. At the sixth time step, the MSE for the progressive scheme compression method (red lines) begins to exceed the benchmark algorithm. We describe it as the turning point. The turning point is a good indicator to upload I-frame time step to reset the accumulated error.

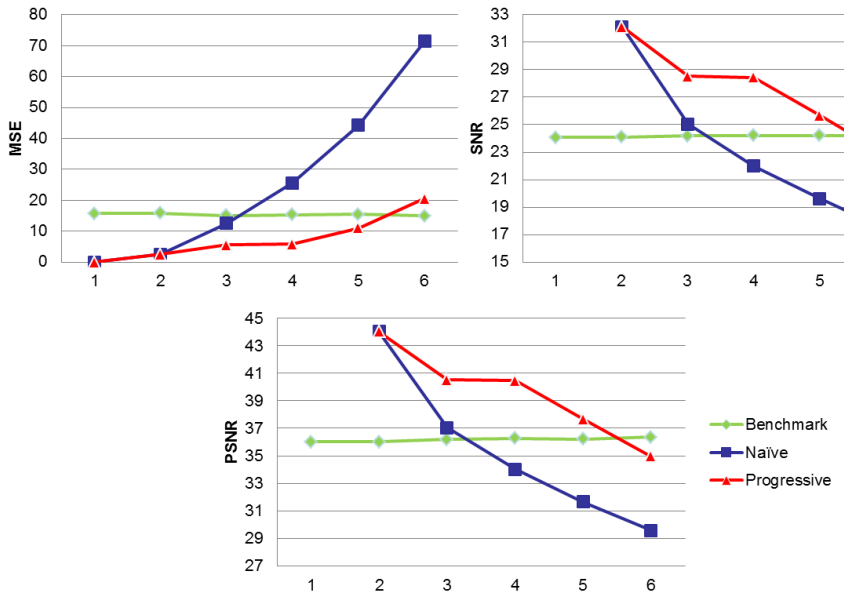
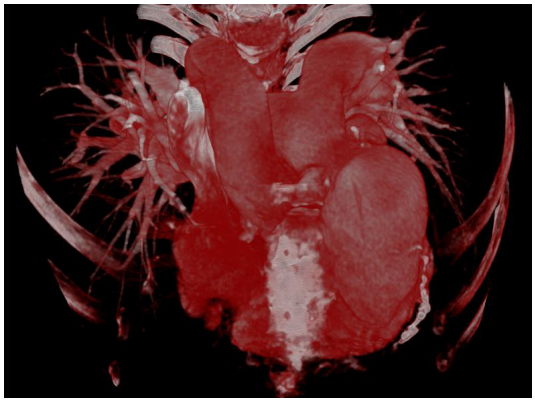


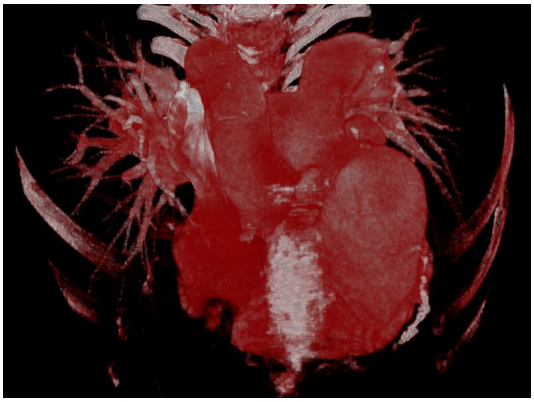
Figure 4.2: The error measurements of different methods

Figure 4.3 are screen shots of the rendering results of time step 6. We use 3D texture-based volume rendering as described in the rendering module. On average, we managed to get 15 to 20 frames per second with our experiment data. In this

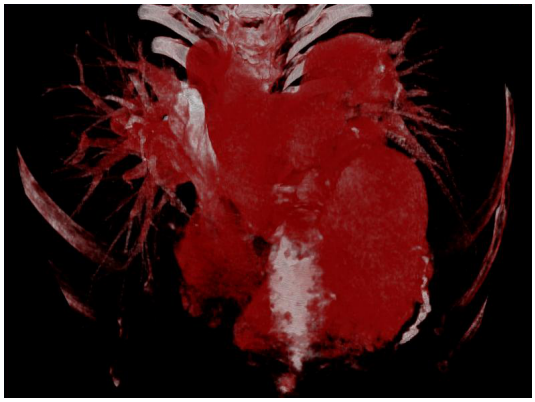
graph, the distortion of Schneider's method is hardly noticable at the given resolution. However, when we zoom the volume in our program, the distortion is fairly obvious. Both the naive scheme compression and progressive scheme compression yields severe distortions.



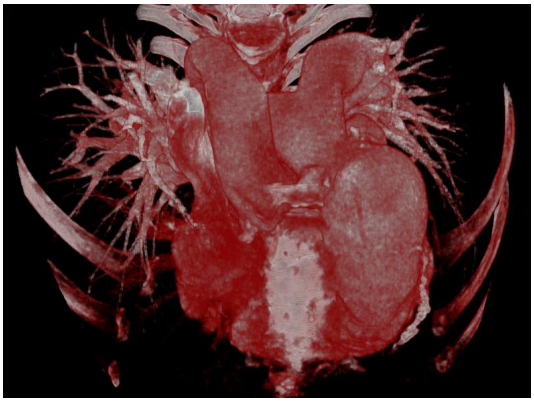
(a) Original volumetric data



(b) Volumetric data reconstructed from Schneider's compression method



(c) Volumetric data reconstructed from naive scheme compression method



(d) Volumetric data reconstructed from progressive scheme compression method

Figure 4.3: The rendering result of time step 6 with different compression methods

From those experiments, one can safely conclude that the progressive scheme compression yields a reduced compression distortion compare to the benchmark method. It efficiently exploits the time redundancy by encoding the difference of continuous time steps.

4.3 Influence of Quantization Parameters on Compression

We briefly mentioned some parameters which we set for our tests. In this section, we will discuss how these parameters affect the compression performance.

Bit-rate is the primary factor affecting the compression error. In our experiment, we used a value of 8 as the vector quantization bit-rate. As the original voxel data was also acquired in the format of 8 bits scalar values, the compression did not introduce large distortion. One limitation of this vector quantization is that as soon as the bit-rate is lower than 8, artifacts become very obvious. A higher bit-rate will definitely reduce the compression distortion, however it also significantly reduce the compression ratio. The size of index and code-books grow very fast with the increased bit-rate. To avoid large distortion or inefficient compression, a rule of thumb is to use the same bit-rate as the original voxel data. As the use of higher bit-rate voxel data is more popular for the sake of accuracy nowadays, a 16 bit or even higher bit-rate may be required. To implement a higher bit-rate decompression in GPU, one can take advantage of α-channel component in the texture.

The space and time redundancy of the original data also have influence on compression errors and ratio. Generally, encoding contiguous and homogeneous voxel data can improve the compression ratio, yet result in even better quality. Our proposed methods are proofs of this theory. The compression errors of our methods are less than the error of benchmark method which only exploits the space redundancy. Medical data usually has high time redundancy. This makes vector quantization very appropriate for this kind of data.

The last but not the least influence factor is the turning point threshold. A turning point is the point we update I-frame to reset the accumulated error. The I-frame is usually much larger than P-frame in terms of size. In this thesis, I-frame is the uncompressed original data. If we select a smaller turning point threshold, it will trigger the I-frame updating more often. Overall, it will give us a less compressed data and a higher fidelity. On the contrary, a higher turning point threshold will give us a more compressed data and a lower fidelity.

These factors are the key factors of our compression method. Except for the space and time redundancy factor, one can have control over the bit-rate and the turning point threshold parameters. Essentially, changing parameters is changing the trade off between compression distortion and compression ratio. One should select these two parameters based on the precision needs of end users.

4.4 Summary

In a nutshell, our experiment results suggest that our proposed system preforms the way we expected. The GPU decompression mechanism reduces almost by half the data preparation time for rendering module. The progressive compression scheme efficiently takes advantage of time redundancy, hence reducing the compression error. To deal with different quality requirements, there are also two adjustable parameters available for end users to balance compression error and compression ratio. For volumetric data with high spacial and time redundancy, such as medical data, we also expect a better performance than random volumetric data. In the domain of real-time rendering of large time-varying volumetric data, our proposed system proves useful and performs better than other existing system found in the literature.

Chapter 5

Conclusion

The goal of this thesis is to explore an efficient way to render large sequence of volumetric data over time so that the end users can visually observe their evolution in time and space at real-time performance (min rate of 10Hz). This type of rendering is very memory and computationally intensive and is currently impossible with standard GPU algorithms. Even with one of today's fastest GPU, the transmission speed between main memory and GPU memory is still too slow for real-time rendering as the bandwidth of the PCI-e bus is at best 2 Gb/s. One way to solve this problem would be to transfer all the frames to the GPU and then render them using the GPU hardware. The problem with this scheme is that the GPU memory is limited to a few frames and will always be as volumetric time sequence continues to grow in resolution/duration and could in theory have an infinite size.

In this thesis, we proposed and tested a new compression scheme to overcome this bottleneck. This compression scheme decompresses the data in GPU memory as if the GPU acted as a client in a video client-server configuration connected by a band limited network. Similar to standard video compression technology like MPEG, the proposed algorithm takes advantage of time and spatial redundancies to reduce the data size to be transferred by the bandwidth limited PCI-e bus. Compare to other method which does not utilize time redundancy information, our proposed compression scheme manages to reduce compression distortion by a generalization of the concept of P and I frames used in the MPEG 2 compression scheme. Once one time step of the volume data is transferred and decompressed in the GPU texture memory, it is immediately rendered using a fast GPU based 3D texture-

based volume rendering algorithm. The interaction handler module allows the user to interact in real-time with their data through different ways, such as: mouse interaction, stereo rendering, clipping, and interactive manipulation of the color and transparency look-up table.

We have demonstrated that the progressive compression scheme using the I and P frames yields a reduce compression distortion compare to the benchmark method found in the literature. It efficiently exploits time redundancy by encoding the difference of continuous time steps.

Our implementation is able to run on most commodity desktop computers. For potential end users, such as radiologists, the system is easy to deploy without the need to buy specialized hardware or software. We use CMake to compile the source code, which give the system cross-platform capability; and the GPU programs in our system can work both on NVIDIA and ATI graphic cards. For users who wants to extend the system capabilities, our modular design makes it easy to incorporate new algorithms. There is one restriction though. All new algorithms must be able to run directly in GPU, since the reconstructed volumetric data is only available in GPU memory.

5.1 Future Work

Before this work, we had a chance to work on Paraview software to visualize large volumetric data. We tested the client/server mode of Paraview using MPI on distributed and shared memory systems. On the server side, volumetric data is broken into pieces to be processed and rendered by different processors and composited later using depth buffer. The client side provides end user an interface to interact with the data. This mode provides scalable rendering for large data. As our compression scheme is also parallel, one interesting future direction is to incorporate this scheme into Paraview. This way, the compression scheme can take advantage of the client/server mode and scale to even larger volumetric data.

Another future direction related to the exploitation of time redundancy. The most straightforward way to exploit the temporal redundancy is storing the differ-

ence between the current and the previous volume rather than the current volume. Although this already reduces the compression distortion remarkably according to our experiment, there are more advanced techniques, such as motion compensation. Adopting these techniques will further improve the compression performance as it does with MPEG video encoding.

Bibliography

[1] J. Ahrens C. Law B. Geveci K. Moreland B. King A. H. Squillacote, D. E. Demarle. *ParaView Guide*. Kitware, Inc., 2007.

[2] K. Akeley. Reality engine graphics. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 109–116, New York, NY, USA, 1993. ACM.

[3] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008.

[4] Anonymous. Vector quantization. http://www.data-compression.com/vq.shtml, 2010.

[5] J. Foran B. Cabral, N. Cam. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *VVS '94: Proceedings of the 1994 symposium on Volume visualization*, pages 91–98. ACM, 1994.

[6] D. Lischinski D. Salesin J. Snyder B. Chamberlain, T. DeRose. Fast rendering of complex environments using a spatial hierarchy. pages 132–141. In: Graphics interface '96, 1996.

[7] A. P.D. Binotto, J. L.D. Comba, and C. M.D. Freitas. Real-time volume rendering of time-varying data using a fragment-shader compression approach. In *PVG '03: Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, page 10, 2003.

[8] J. Blinn. Light reflection functions for simulation of clouds and dusty surfaces. volume 16, pages 21 – 29. Computer Graphics, July 1982.

[9] E. E. Catmull. *A subdivision algorithm for computer display of curved surfaces*. Phd thesis, University of Utah, 1974.

[10] L.Y Li Han-Wei Shen C.L. Wang, J.Z Gao. A multiresolution volume rendering framework for large-scale time-varying data visualization. *International Workshop on Volume Graphics*, 0:11–223, 2005.

[11] J. Neider T. Davis D. Shreiner, M. Woo. *OpenGL Programming Guide: The Official Guide to Learning OpenGL*. 3rd edition, 1999.

[12] R. A. Drebin, L. Carpenter, and P. Hanrahan. Volume rendering. In SIGGRAPH88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques, August 1988.

[13] Jose A. I. Guitian E. Gobbetti, F. Marton. A single-pass gpu ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer*, 24:797–806, July 2008.

[14] D.H. Salesin E.J. Stollnitz, A.D. DeRose. Wavelets for computer graphics. *Computer Graphics and Applications, IEEE*, 15(3):76–84, May 1995.

[15] R. Fernando and M. J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley, 2004.

[16] A. G. Gallagher, E. Matt Ritter, H. Champion, G. Higgins, M. P. Fried, G. Moses, C. D. Smith, and R. M. Satava. Virtual reality simulation for the operating room: Pro?ciency-based training as a paradigm shift in surgical skills training. Annals of Surgery, 2005.

[17] E. Groller, I. Fujishiro (editors, Chaoli Wang, Jinzhu Gao, Liya Li, and Han wei Shen. A multiresolution volume rendering framework for large-scale time-varying data visualization abstract, 2008.

[18] S. Guthe and W. Straer. Real-time decompression and visualization of animated volume data, 2001.

[19] J. Knittel H. Lauer L. Seiler H. Pfister, J. Hardenbergh. The volumepro real-time ray-casting system, 1999.

[20] L. F. Hodges. Tutorial: time-multiplexed stereoscopic computer graphics. IEEE Computer Graphics and Applications, 1992.

[21] S. Park I. Ihm. Wavelet-based 3d compression scheme for very large volume data. In *In Proceedings of Graphics Interface '98*, pages 107–116, 1998.

[22] L. Ibanez, W. Schroeder, L. Ng, and J. Cates. *The ITK software guide*. Kitware Inc, second edition edition, 2005.

[23] Medical Imaging and Technology Alliance. Dicom homepage, 2010. http://medical.nema.org/.

[24] Dimension Technology Inc. Dimension technology inc., 2010. http://www.dti3d.com/.

[25] R. Westermann J. Schneider. Compression domain volume rendering. In *IEEE Visualization*, pages 293–300, 2003.

[26] A. V. Gelder J. Wilhelms. A coherent projection approach for direct volume rendering. volume 25, pages 275 – 284. Computer Graphics, July 1991.

[27] J. M. Kniss C. Rezk-Salama D. Weiskopf K. Engel, M. Hadwiger. *Real-Time Volume Graphics*. A K Peters. Ltd, 2006.

[28] E. Nakamae K. Kaneda, T. Okamoto and T. Nishita. Highly realistic visual simulation of outdoor scenes under various atmospheric conditions. n Proceedings of CG International 90, August 1990.

[29] A. Kaufman. Voxel-based architectures for three-dimensional graphics. pages 361–366. Dublin, Ireland, September 1986.

[30] A. Kaufman and R. Bakalash. Memory and processing architecture for 3d voxel-based imagery. *IEEE Comput. Graph. Appl.*, 8(6):10–23, 1988.

[31] A. E. Kaufman. Volume visualization. ACM Computing Surveys, 1996.

[32] A. E. Kaufman. Introduction to volume graphics, 2000.

[33] G. Kiefer, H. Lehmann, and J. Weese. Visualization of large medical data sets using memory-optimized cpu and gpu algorithms. In Proc. SPIE International Symposium on Medical Imaging, 2005.

[34] Kitware. Paraview - open source scientific visualization, 2010. http://www.paraview.org/.

[35] E.C. LaMar and B. Hamannand K.I. Joy. Multiresolution techniques for interactive texture-based volume visualization. In *VISUALIZATION '99: Proceedings of the 10th IEEE Visualization 1999 Conference (VIS '99)*. IEEE Computer Society, 1999.

[36] A. Lefohn. Gpu memory model overview. In *SIGGRAPH '05 ACM SIGGRAPH 2005 Courses*, 2005.

[37] M. Levoy. Efficient ray tracing of volume data. Technical Report 88-029, June 1988.

[38] M. Levoy. Design for a real-time high-quality volume rendering workstation. In *VVS '89: Proceedings of the 1989 Chapel Hill workshop on Volume visualization*, pages 85–92, 1989.

[39] M. Levoy. *Display of Surfaces from Volume Data*. PhD thesis, University of North Carolina at Chapel Hill, 1989.

[40] W. Li, K. Mueller, and A. Kaufman. Empty space skipping and occlusion clipping for texture-based volume rendering. page 42, 2003.

[41] L. Kobbelt M. Pauly, M. Gross. Efficient simplification of pointsampled surfaces. In Proceedings of IEEE Visualization 2002, 2002.

[42] C. Hansenand K. Zimmermann T. Ertl M. Weiler, R. Westermann. Level-of-detail volume rendering via 3d textures. In *VVS '00: Proceedings of the 2000 IEEE symposium on Volume visualization*, pages 7–13. ACM, 2000.

[43] K. Martin and B. Hoffman. *Mastering Cmake: a cross-platform build system*. Kitware Inc., 2006.

[44] N. Max. Optical models for direct volume rendering. pages 99 – 108. IEEE Transactions on Visualization and Computer Graphics, June 1995.

[45] Nelson Max. Atmospheric illumination and shadows. Computer Graphics, August 1986.

[46] K. G. Nguyen and D. Saupe. Rapid high quality compression of volume data for visualization. In: Computer Graphics Forum, 2001.

[47] OpenGL.org. Framebuffer object, 2009. http://www.opengl.org/wiki/Framebuffer_Object.

[48] M. Levoy P. Lacroute. Fast volume rendering using a shear-warp factorization of the viewing transformation. Proc. SIGGRAPH '94, July 1994.

[49] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive volume on standard pc graphics hardware using multi-textures and multi-stage rasterization. In *HWWS '00: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 109–118. ACM, 2000.

[50] C. Rezk-Salama and A. Kolb. A vertex program for efficient box-plane intersection. 2005.

[51] J. Gonser W. Straer S. Guthe, M. Wand. Interactive rendering of large volume data sets. pages 53–60, 2002.

[52] F. Neyret S. Lefebvre, J. Darbon. Unified texture management for arbitrary meshes. Technical Report 5210, Institut National de Recherche en Informatique et en Automatique, 2004.

[53] D. Weiskopfand T. Ertland W. Strasser S. Roettger, S. Guthe. Smart hardware-accelerated volume rendering. In *VISSYM '03: Proceedings of the symposium on Data visualisation 2003*, pages 231–238. Eurographics Association, 2003.

[54] K. Sayood. *Introduction to Data Compression*. Morgan Kaufmann Publishers, second edition, 2000.

[55] P. Schroder and Steven M. Drucker. A data parallel algorithm for raytracing of heterogeneous databases, 1992.

[56] R. Shen. Medvis: A real-time immersive visualization environment for the exploration of medical volumetric data. Master's thesis, University of Alberta, 2007.

[57] Canadian Cardiovascular Society. Ccs consensus document on congestive heart failure. 2006.

[58] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. A simple and flexible volume rendering framework for graphics-hardware-based raycasting. In *Volume Graphics, 2005. Fourth International Workshop on*, pages 187–241, June 2005.

[59] B. Sun and R. Ramamoorthi. A practical analytic single scattering model for real time rendering. *ACM Trans. Graph*, 24:1040–1049, 2005.

[60] wikipedia. Comparison of nvidia graphics processing units, 2009. http://en.wikipedia.org/wiki/ Comparison_of_Nvidia_graphics_units.

[61] wikipedia. Transistor count, 2010. http://en.wikipedia.org/Transistor_count.

[62] R. Gray Y. Linde, A. Buzo. An algorithm for vector quantizer design. IEEE Transactions on Communications, Jan 1980.