Real-Time Rendering of Temporal Volumetric Data on a GPU

Biao She, Pierre Boulanger, and Michelle Noga
Computer Science Dept., Computer Science Dept., Radiology Dept.
biao.she@ualberta.ca, pierreb@ualberta.ca, mnoga@ualberta.ca

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 time-varying volumetric data. 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 the GPU, reducing significantly the data transfer between main memory and graphic memory.

1 Introduction

The ability to image a heart in real-time, to visualize it, and to plan for treatments is key to reducing the cost of surgeries and to improving treatments. All 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.

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 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 research project is to develop a GPU based algorithm capable of displaying large temporal volumetric data sets. The GPU contains multiple graphics processors that work in parallel, allowing the rendering of volumetric data in real-time. Using new volume texture based rendering 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. 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 the main computer bus, 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.

This paper is organized as follows, Section 2 reviews briefly the current state-of-the-art of medical volumetric visualization. Section 3 describes the system developed and its design objectives. Section 4 discusses the results of the implementation, and finally we conclude and present future research directions.

2 Literature Review

Volume rendering has been extensively studied for many years. Generally, there are five different optical rendering models for volume rendering, which are *Absorption Only [7] [2], Emission Only [8], Emission-Absorption Model [3] [1], Single Scattering [14] [4], and Shadowing [11] [9] and Multiple Scattering* [4]. Single Scattering and multiple scattering calculations are expensive in computer time, hence not suitable for real-time rendering yet.

The most widely used rendering technique is the Emission-Absorption Model. Both light emission and absorbtion in the volume are taken into account in this models. Solving the equation for volume rendering is both a computational and memory intensive process. 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. Most real-time rendering algorithms require 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. 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 the 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 that needs to be sent to the GPU. Many researchers [13] [5] [6] adopted data compression in their attempt to visualize real-time large volumetric data sets. Octree-structures, multi-resolution representation, wavelet-compression, run-length encoding and vector quantization are among the techniques used in some of those 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.

3 System Design

The 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 our shader programs can run on most GPUs, either NVIDIA or ATI.

Figure 1 shows an overview of the 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. The interaction hander module provides parameters from input devices to the rendering module to generate interactive feedback and manipulations and is not described in this paper.

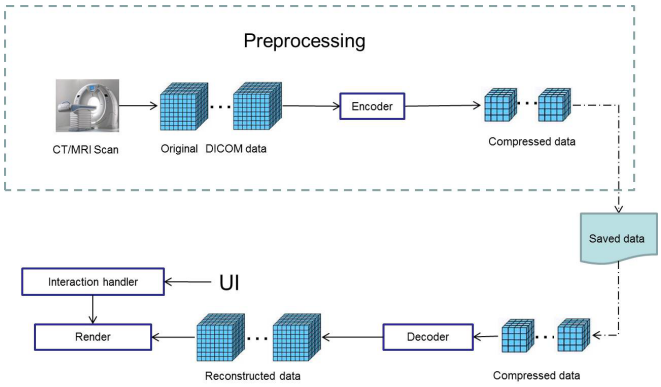


Figure 1: Overview of the rendering system.

The original volumetric data is usually obtained from a 4D CT or cardiac MRI scan. The scanner generates a large set of raw data in DICOM format which are saved into a hard disk or other massive storage devices. The encoder module then reads these DICOM files and processes them using the compression algorithms. The program then saves the compressed data into hard disk in a dedicated format. After preprocessing, the file reader reads the compressed data from the 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 impossible to be implemented on GPU. However, vector quantization, which is the algorithm we adopted, is almost perfect for GPU decompression because of its parallel nature. 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 must supports render to frame buffer object which can be directly read as 3D texture. After the decoder mod-

ule reconstructs the original data in the 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 generate the final image displayed on screen. The interaction module allow user to interact with the system in real-time. In our implementation, user can switch between stereo mode and normal rendering mode. Our implementation also allows clipping along all three axis of the volume coordinates.

4 Implementation

As mentioned previously our system is composed of four sub-modules that will now be describe in some details.

4.1 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. To be real-time the decompression algorithm must be executed on the GPU very efficiently. Original data from 4D CT or cardiac 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 2. In Figure 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. Trilinear interpolation is used to interpolate the value of the intermediate point from the original sample points. The reason for re-sampling is that our compression and rendering algorithm requires volume data with a size which is a multiple of the power of two in every dimension for efficient addressing.

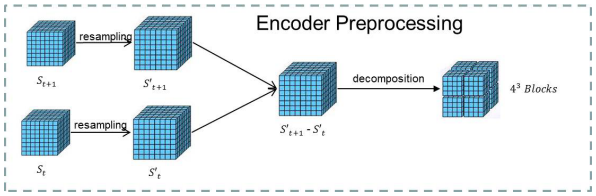


Figure 2: Preprocessing steps in the encoder module.

After re-sampling, all the data sets are at the desired dimension. The next step is to exploit the time redundancy of the re-sampled data. In our implementation, we subtract the re-sampled data of two consecutive volume, which is $S'_{t+1} - S'_t$. 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 opportunity to compress the signal. Vector quantization perform better on these sparse blocks. In our implementation, a higher signal-to-noise ratio (SNR) was observed when quantizing the sparse blocks instead of the 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 between neighboring sample points. Our system use the same decomposition method as in Schneider's paper [6]. The temporal difference $S'_{t+1} - S'_t$ is first divides into small blocks of size 4^3. Next, these blocks are decomposed into a multi-resolution representation. There are three steps to do this decomposition. Figure 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*. 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 $u = x + 4 * y + 16 * z$, where u 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 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 values 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.

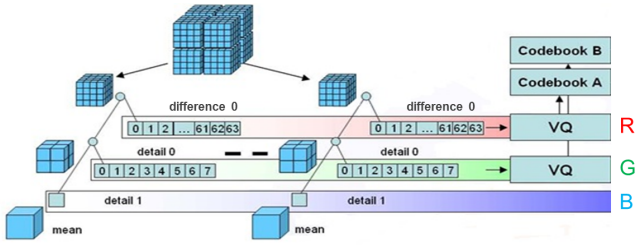
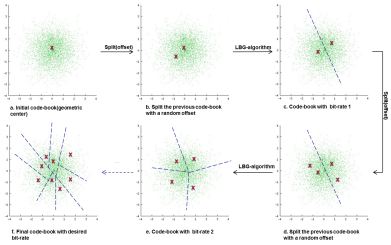


Figure 3: Data decomposition and vector quantization in the encoder module.

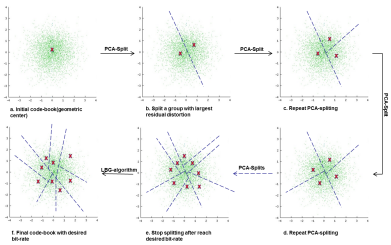
To quantize these blocks, we use a modified vector quantization algorithm. It is a more advanced algorithm than the conventional vector quantization algorithm. The advantages of the modified version comes from the way it gets the initial code-book. Normal algorithm, also called LBG-algorithm, recursively splits data set to get the code-book. Figure 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 a LBG-algorithm and processed. It stops until convergence is reached. The same splitting is then applied 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. [10] proposed a splitting technique where they addressed hierarchical clustering problems. The modified algorithm uses the same technique to split the original data as illustrated in Figure 4b. Like the normal splitting technique, 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 with similar distortion rates. 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 can be found in Schneider's paper [6].

After vector quantization, we now have two code-books and a group of RGB color triplets. As shown in Figure 3, 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 $D'_x/4 * D'_y 4 * D'_z/4$ arrays in total for the difference level 0. Those vectors are approximated by 256 different vectors after vector quantization. The R component 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 component in the RGB of the color triplets. The last element in the RGB color triplets stores the mean value of the corresponding 4^3 block. There are $D'_x/4 * D'_y/4 * D'_z/4$ such RGB color triplets in total. These



(a) LBG vector quantization



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

Figure 4: Procedures of two different vector quantization algorithms.

triplets, as well as the code-books A and B, are then saved to the hard drive for later use.

4.2 Decoder Module

The encoder preprocessing ends after the resulting files is saved on the hard drive. The first thing the decoder module needs to do is to read those saved files from the hard drive into main memory. The code books and the RGB color triplets are parsed and saved into main memory. 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. We use the texture memory in GPU to store the code-books and the triplets from main memory for a time-step. 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, a 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 5 shows the overview the texture memory structure of 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

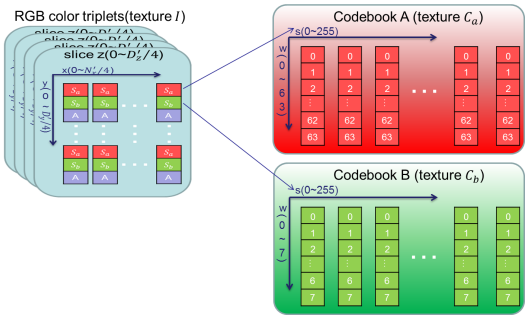


Figure 5: GPU textures memory structure for the decoder module.

equation is:

$$w(x,y,z) = I(x',y',z').b + C_b(s,w) + C_a(s,w). \quad (1)$$

In this equation, $w(x,y,z)$ is the scalar value at location (x,y,z); $I(x',y',z').b$ 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 1 to a function of (x,y,z). Because (x,y,z) is 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 of the re-sampled data. The coordinates (x',y',z') 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 between (x,y,z) and (x',y',z') 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',y',z') can be expressed by:

$$\begin{aligned} x' &= \lfloor x/4 \rfloor, where\ 0 \le x' \le (D'_x/4-1)\ and\ 0 \le x \le (D'_x-1) \\ y' &= \lfloor y/4 \rfloor, where\ 0 \le y' \le (D'_y/4-1)\ and\ 0 \le y \le (D'_y-1) \\ z' &= \lfloor z/4 \rfloor, where\ 0 \le z' \le (D'_z/4-1)\ and\ 0 \le z \le (D'_z-1) \end{aligned} \quad (2)$$

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.

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 5. The w coordinate is an integer between 0 and 63. It is essentially a 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:

$$(s,w) = \begin{cases} I(x',y',z').r \\ x\%4 + 4*(y\%4) + 16*(z\%4) \end{cases} \quad (3)$$

where $I(x',y',z').r$ 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 the following:

$$(s',w') = \begin{cases} I(x',y',z').g \\ \lfloor \frac{x\%4}{2} \rfloor + \lfloor y\%4 \rfloor + 2*\lfloor \frac{z\%4}{2} \rfloor. \end{cases} \quad (4)$$

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 2, 3 and 4 can be inserted into Equation 1 to get:

$$\begin{aligned} w(x,y,z) \quad = \quad & I(\lfloor x/4 \rfloor, \lfloor y/4 \rfloor, \lfloor z/4 \rfloor).b \\ & + C_b(s',w') \\ & + C_a(s,w) \end{aligned} \quad (5)$$

where (x,y,z) are integer values ranging from $[0,D'_x-1]$, $[0,D'_y-1]$, and $[0,D'_z-1]$ respectively.

Equation 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 potable 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 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 this 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. Instead of integers, the coordinates need to be transformed 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 that remains to be solved. The first problem is to figure out a way to get coordinates (x',y',z') 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 one 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 functions yields the correct coordinates for texture I:

$$\begin{aligned} x' &= (x*0.5+0.5)*\frac{D_x-1}{D_x-4} - \frac{3}{2*D_x-8} \\ y' &= (y*0.5+0.5)*\frac{D_y-1}{D_y-4} - \frac{3}{2*D_y-8} \\ z' &= (z*0.5+0.5)*\frac{D_z-1}{D_z-4} - \frac{3}{2*D_z-8} \end{aligned} \quad (6)$$

where $D_n = \{2^n \mid n >= 3\}$. 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.

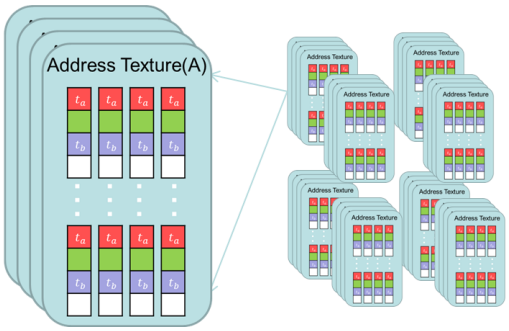


Figure 6: The structure and repeat pattern of address texture.

The second problem is to get the w coordinates for each small blocks. Address texture, as proposed by Schneider et al. [6], is a good solution for this problem. The basic idea is to use the address texture to hold the w coordinates of one single 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 6 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 codebooks 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 that 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 5. One can use a progressively larger integer in every dimension to replace (x,y,z). 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 7 describes the transformation:

$$\begin{aligned} x_A &= (x*0.5+0.5)*(D_x/4) \\ y_A &= (y*0.5+0.5)*(D_y/4) \\ z_A &= (z*0.5+0.5)*(D_z/4) \end{aligned} \quad . \quad (7)$$

where x_A, y_A, and z_A represent the texture coordinates for accessing texture A. Figure 6 illustrates 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 5 describes the decoding process mathematically as performed on the GPU. By combining Equation 5 with Equation 7 it is not hard to write its implementation in Cg. Because our decompressed data are 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 previous 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 codebook 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. We decided to 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 as in MPEG coding allowing us 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.

4.3 Render Module

The render module is responsible for rendering in real-time the volumetric data. Numerous rendering algorithms have been proposed by researchers in the past decades. We use 3D texture-based volume rendering in our system. It is an object-order volume rendering algorithm because it use 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, 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 using the CPU. Rezk-Salama et al. [12] 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.

4.4 Implementation Results

In order to test our system, we use a desktop computer running Windows XP with a 4G memory Quadro FX 5800 graphics card. The interface between the GPU and main memory is performed by a PCI Express 2.0 bus. 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. The usefulness of our system relies on the assumption that the implemented GPU decompression mechanism saves time compare to direct data transfer through the PCIe 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 1 is the results for I frame. So t_1 is the same as t_2 and the compression ratio is 1. Table 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 vs the time spend on transmitting the compressed data. As the data size gets larger, more portion of time is dedicated to GPU decompression. These two examples are sufficient to validate the assumption which our system is based on. In essence, they are the amount of time it takes to get volumetric data ready for the rendering module. We call them *data*

Table 1: The data preparation time and compression ratio for an I frame.

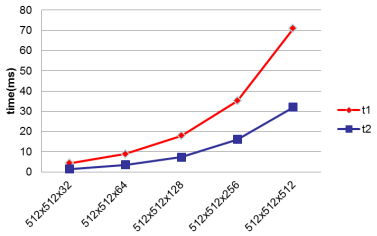
Data Dimension	$t_1(ms)$	$b(Gb/s)$	$t_2(ms)$	$S_o(Mb)$	$S_c(Mb)$	r
512x512x32	4.2	1.905	4.2	8	8	1
512x512x64	8.9	1.798	8.9	16	16	1
512x512x128	17.8	1.798	17.8	32	32	1
512x512x256	35.1	1.823	35.1	64	64	1
512x512x512	70.9	1.805	70.9	128	128	1

Table 2: The data preparation time and compression ratio for a P frame.

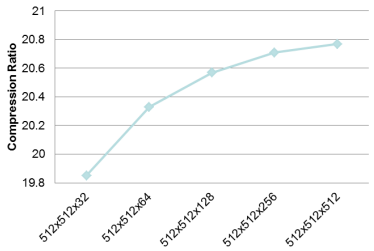
Data Dimension	$t_1(ms)$	$b(Gb/s)$	$t_2(ms)$	$t_2{}'(ms)$	p	$S_o(Mb)$	$S_c(Mb)$	r
512x512x32	4.2	1.905	1.4	0.2	6.00	8	0.403	19.85
512x512x64	8.9	1.798	3.5	0.4	7.75	16	0.787	20.33
512x512x128	17.8	1.798	7.3	0.9	7.11	32	1.555	20.57
512x512x256	35.1	1.823	16.1	1.7	8.49	64	3.091	20.71
512x512x512	70.9	1.805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

preparation time. 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. Figure 7a shows the curves of data size vs. 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 be necessary to use GPU decompression method since the direct transfer time is already short. Moreover, the GPU decompression algorithm is not a lossless decompression. The speedup is not worthy compare 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 7b. 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 7b. 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 our system with is based on Schneider's method. In their paper, they use mean square error (MSE), signal-to-noise ratio (SNR) and peak signal-



(a) Data 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 7: System performance for single time step with a P frame.

Table 3: Compression errors of our benchmark method.

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

to-noise ratio (PSNR) as performance indicators. The data we use is of size 512x512x256 and there are 6 different time steps in total. Table 3, 4, 5 show results which we got from the experiments. We did three different tests on these data. Table 3 is the benchmark test on Schneider's method. Their method treated different time steps individually and compressed them separately. The difference lies in the way they calculate the difference between different time steps. In Table 4, we get the difference from two original time steps; and in Table 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. We call the second method the naive scheme and the third method the progressive scheme.

Figure 8 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 dif-

Table 4: Compression errors using the naive scheme.

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 5: Compression errors using the progressive scheme.

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

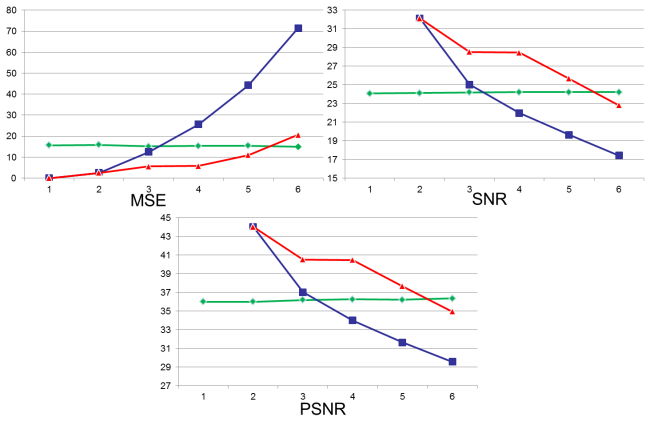


Figure 8: The error measurements of different methods

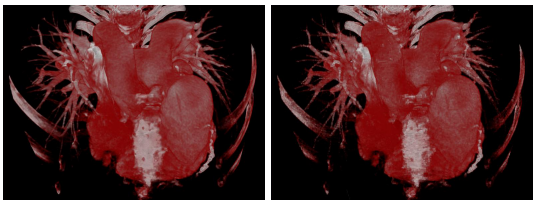
ferent time steps are separated from each other, the compression algorithm performance is quite stable. The green lines are almost horizontal in the graph. The naive scheme compression method is represented by the blue lines. As seen in the graph, the blue lines increase or decrease dramatically compare 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 8, 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 9 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 noticeable 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.

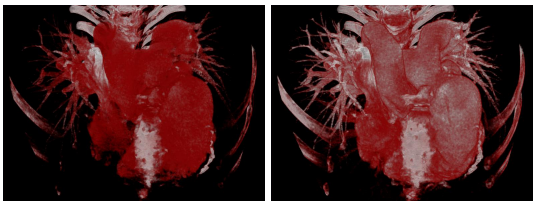
From those experiments, one can safely conclude that the progressive scheme compression yields a reduced compression distortion compare to the benchmark method. It efficiently exploit the time redundancy by encoding the difference of continuous time steps.

Conclusion

The goal of this paper 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 12 Gb/s. In this paper, 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 com-



(a) Original volumetric data (b) Volumetric data reconstructed using Schneider's scheme



(c) Volumetric data reconstructed using naive scheme (d) Volumetric data reconstructed using progressive scheme

Figure 9: The rendering result of time step 6 with different compression methods

pression 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 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. 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 exploit time redundancy by encoding the difference of continuous time steps.

References

[1] 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.

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

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

[4] 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.

[5] 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.

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

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

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

[9] 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.

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

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

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

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

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