Texture Hardware Assisted Rendering of Time-Varying Volume Data

Eric B. Lum* Kwan-Liu Ma* John Clyne†

University of California, Davis National Center for Atmospheric Research

Abstract

In this paper we present a hardware-assisted rendering technique coupled with a compression scheme for the interactive visual exploration of time-varying scalar volume data. A palette-based decoding technique and an adaptive bit allocation scheme are developed to fully utilize the texturing capability of a commodity 3-D graphics card. Using a single PC equipped with a modest amount of memory, a texture capable graphics card, and an inexpensive disk array, we are able to render hundreds of time steps of regularly gridded volume data (up to 45 millions voxels each time step) at interactive rates, permitting the visual exploration of large scientific data sets in both the temporal and spatial domain.

Keywords: Compression, high performance computing, out-of-core processing, PC, scientific visualization, texture hardware, time-varying data, transform encoding, volume rendering

1 Introduction

High-resolution, four-dimensional data sets are typical of problems in many areas of science and engineering. Specific examples include data from the study of neuron excitation, material crack propagation, thunderstorm evolution, unsteady flow surrounding an aircraft, seismic reflections from geological strata, and even galaxy merger. These data sets, rich with detailed information of complex physical or chemical processes, may be generated either by numerical simulation or collected through instrumentation. Regardless of how they are derived, the ability to quickly detect and explore the complex, dynamic phenomena contained within is essential to their analysis.

A typical time-varying data set from a computational fluid dynamics simulation can contain hundreds or even thousands of time steps and each time step may have millions of grid points, each potentially containing multiple variables. As a result, a single data set can easily occupy hundreds of gigabytes of storage, creating a formidable challenge for subsequent analysis. Traditional statistical methods of analysis, though relatively easy to compute, tend to filter out information, computed at great expense, by reducing data to a relative few numbers. Visual data exploration techniques, such as direct volume rendering, have arisen as powerful aids to researchers in the analysis of these vast data sets. Scientific visualization can greatly facilitate and expedite the exploration of these data sets by exploiting the brain's ability to process enormous amounts of visual information.

However, for visualization techniques to be most effective for enhancing the qualitative understanding of complex behavior, or for detection of features of importance, they must be interactive in every aspect. The ability to change classification functions and color mappings quickly and easily, animate forward and backward in time, change viewpoints, and zoom in and out on features of interest, all at interactive rates, is essential for maximizing scientific productivity [3].

In this paper, we discuss how time-varying volume data sets can be efficiently rendered by utilizing a PC, low cost graphics hardware, and an inexpensive disk array. Transform encoding of the volume data, followed by compression, not only reduces storage but also bandwidth requirements. We also exploit the 2-D texture features of a commodity graphics card to speed up the rendering as much as possible. The result is a rendering capability in which all essential aspects of the visualization process are completely interactive.

We have evaluated an implementation of our design using a number of time-varying data sets including the quasi-geostrophic (QG) turbulence data sets provided by scientists at the National Center for Atmospheric Research (NCAR) and the University of Colorado. The highly interactive rendering rates achieved by our PC-based system were previously available to NCAR scientists only through the use of costly parallel supercomputers or high-end multiprocessor graphics workstations, thereby greatly limiting their access. In the following sections, a survey of time-varying data visualization research is followed by a detailed description of our design. The performance and explorability of the prototype system we have built are demonstrated with timing results, images, and a video using three data sets with sufficiently different characteristics.

*CIPIC & Department of Computer Science, University of California, One Shields Avenue, Davis 95616, {lume,ma}@cs.ucdavis.edu

†National Center for Atmospheric Research, 1850 Table Mesa Dr., Boulder, CO 80303, clyne@ncar.ucar.edu

2 Previous Work

Time-varying data visualization has been an active research area. Various approaches have been proposed to reduce the storage, corresponding I/O, and rendering demands for visualizing time-varying data in a more efficient way.

Using proper encoding and exploiting temporal coherence or spatial coherence or both, the storage requirements and rendering cost of the data may be significantly reduced. Shen and Johnson [19] take advantage of temporal coherence, using difference encoding to significantly diminish storage and rendering requirements. Westermann [21] performs wavelet encoding of each time step separately to generate a compressed multiscale tree structure. Further compression can be obtained by examining the resulting tree structures and wavelet coefficients. Ma and Shen [11] discuss how non-uniform quantization along with octree and difference encoding can be employed to speed up rendering of time-varying volume data. They show that the octrees for consecutive time steps can be merged to share subtrees. Consequently, during rendering, partial images built from subtrees that have not changed over time may be reused in later time steps.

Wilhelms and Van Gelder [23] design hierarchical data structures for controlled compression and volume rendering. They extend octrees and a branch-on-need (BON) subdivision strategy [22] to handle multi-dimensional data. Sutton and Hansen [20] propose a temporal branch-on-need tree (T-BON) as an extension to the 3-D BON tree for time-varying isosurface extraction. Shen, Chiang, and Ma [18] introduce a hierarchical data structure, called a Time-Space Partitioning (TSP) tree, for better utilization of both spatial and temporal coherence. In essence, the skeleton of a TSP tree is a standard complete octree, which recursively subdivides the volume spatially until all subvolumes reach a predefined minimum size. To store the temporal information, each TSP tree node itself is a binary tree. Every node in the binary time tree represents a different time span for the same subvolume in the spatial domain. Most importantly, TSP trees allow the renderer to use data from subvolumes of different spatial and temporal resolutions, which is not possible for 4-d octrees. The TSP tree data structure has been also used to facilitate large scale volume rendering using 3-D texture hardware [6].

Further speedup of rendering may be made by utilizing parallel computers or graphics hardware. However, even though a parallel computer can render images at multiple frames per second, without high-speed network and parallel I/O support, two bottlenecks can still make it impossible to achieve interactive viewing. One bottleneck is the need to stream large volume files throughout the course of the visualization process. The other is the delay due to transferring the resulting images over a potentially non-dedicated network. Ma and Camp [10] developed a post-processing parallel visualization strategy based on pipelined rendering. They demonstrate remote visualization of time-varying volume data on a PC cluster over a wide-area network. Pipelining and careful grouping of processors are used to hide I/O time and to maximize processor utilization. Visually lossless compression is used to significantly cut down the cost of transferring output images from the PC cluster to a display device through a wide-area network. Clyne and Dennis [4] employ similar techniques, using double buffering to help mask both the costs of volume data I/O over a high-bandwidth channel as well as image transmission over a TCP/IP network.

The methods introduced in this paper can work equally well as, and by many metrics better than, most of the aforementioned techniques. Furthermore our methods have the added advantage running on low-cost, commodity hardware making them far more accessible to a broad range of researchers.

3 Hardware-Assisted Rendering

Commodity PC graphics cards are capable of performing rendering that only a few years ago required a high-end graphics workstation. In particular, the 2-D texture hardware that helps generate impressive graphics for video games can also be applied to make effective visualizations. For example, commodity PC graphics cards have been used for volume rendering of static volumetric data [15]. Volume rendering requires the loading of the volumetric data into the texture memory of the video card prior to rendering. The size of the volume that can be rendered is often limited by the amount of video memory the card contains, since the access and transfer of data from main memory across the graphics bus is relatively slow compared to the direct access of graphics memory.

3.1 Compression

The interactive rendering of time-varying volumetric data sets offers a number of challenges because of the sheer size of the data being visualized. These data sets can be reduced in size and therefore made more manageable through the use of compression. The advantages of compressing volumetric data are two-fold. First, it reduces the storage requirements needed for the data. This could allow a data set to fit in main memory that might not fit otherwise, eliminating the need for transferring data from disk. The reduction in storage can also be used to fit relatively small compressed data sets entirely into texture memory, thus eliminating the need for transferring data across the graphics bus. The other benefit of compression is a reduction in I/O. If a compressed volume fits entirely into main memory, the cost of transferring compressed data to the graphics card is much lower than the cost of transferring uncompressed data. If a data set does not fit into main memory, the transferring of compressed data from disk can be substantially faster than with uncompressed data, allowing for interactive visualization from disk.

Video and main memory can be thought of as a two-level cache for volume rendering. The compression of volumetric data not only increases the amount of data that can fit in each level, but also decreases the I/O costs of transfers between these levels. Through the use of compression, and careful management of the time costs associated with the transfers between levels, it is possible to load texture maps representing volume data into video memory at rates suitable for interactivity on a commodity PC.

If a compressed volume is to be rendered directly from video memory, it must also be uncompressed using the graphics hardware. This is a significant constraint since the operations supported in graphics hardware are limited compared to those found on a general purpose CPU. Another constraint is imposed by the desire to encode the scalar voxel values in terms of their scalar value rather than as a red, green, blue, alpha (opacity) set. Using scalar values and color indexed textures allows a scientist to manipulate the color palette to interactively change the opacity and color maps, leading to more intuitive visualization of the data. Storing voxels in terms of RGBA would require recompressing the entire data set when the transfer function is changed, which can be impractical for very large data sets. In addition, storing a single scalar value, rather than four color scalars reduces the amount of data by a factor of four.

Unfortunately, using indexed values puts a number of limitations on how graphics hardware can be used to decode data. Most screen and texture combining operations supported in hardware, such as Register Combiners, work in terms of the manipulation of RGBA values and not the manipulation of scalar map index values. In particular, one might consider a compression method that deals with difference images or volumes. These differences, however, would need to be combined in terms of RGBA and not indexed scalars. This would make the difference images color map and opacity map dependent, since the difference between two volumes, in terms of RGBA, depends extensively on the transfer function being used.

3.2 Palette based decoding

With these limitations in mind, we present a method for the temporal encoding of indexed volumetric data that can quickly be decoded in hardware. The method makes extensive use of hardware support for the changing of color palettes without the reloading of textures. The cycling of color palettes can be used to create simple animations from static images. In our work we use color palette manipulation to allow a single scalar index to represent grid points at several times steps.

With paletted textures, a single scalar index is used to represent an RGB or RGBA color. The palette consists of a limited set of colors that sample the RGBA color space. Each of these colors is encoded in a single value, often a single byte. In our approach we encode a sequence of temporally changing scalar values into a single index. In this way, the value stored in each texel represents an approximation of a sequence of scalar values. Each index is therefore a sample in the space of possible time varying scalar values. The scalar values that an indexed texel represents is decoded to its temporally changing values through the frame to frame manipulation of the palette. For each frame, the color for each palette entry is set to the color found in the transfer function for the scalar encoded by that index value during that frame, as shown in the following pseudocode which renders N time steps using a single indexed texture. Note that 8-bit indexed textures are assumed.

```
{
   // stores colormap from the transfer function
   Color colormap[256];
   // stores the N time varying scalars encoded by each
   // of the 256 possible texel values. This array is created
   // during the compression process
   int decoder[N][256];
   // the color palette to be calculated for each time step
   Color palette[256];

   for each timestep t (0 to N-1)
   for each palette entry i (0 to 255){
      palette[i]=colormap[decoder[t][i]];
      // set the palette for current frame
      setPallette(pallette);
      renderTexture();
   }
}
```

The textures are rasterized to the screen using linear interpolation. Linear interpolation occurs in terms of RGBA values after they have been looked up from the palette. If interpolation occurred in terms of palette indices, the resulting images would show severe artifacts, since the mapping between palette indices and decoded scalar values is far from linear.

3.3 Temporal encoding

The encoding process consists of mapping sequences of scalars into single scalar indices. This operation can be approached as a vector quantization problem. We perform this process using transform encoding, specifically using the Discrete Cosine Transform (DCT) [7, 17]. Transform encoding is a compression method that transforms data into a set of coefficients that are then quantized to create a more compact representation. The transform by itself is reversible, and does not compress the data. Rather, a transform is selected that puts more energy into fewer coefficients, thus allowing the less important, lower energy coefficients to be quantized more coarsely, thus requiring less storage.

The DCT is defined by:

$$C(u) = \alpha(u) \sum_{x=0}^{N-1} f(x) cos[\frac{(2x+1)u\pi}{2N}]$$

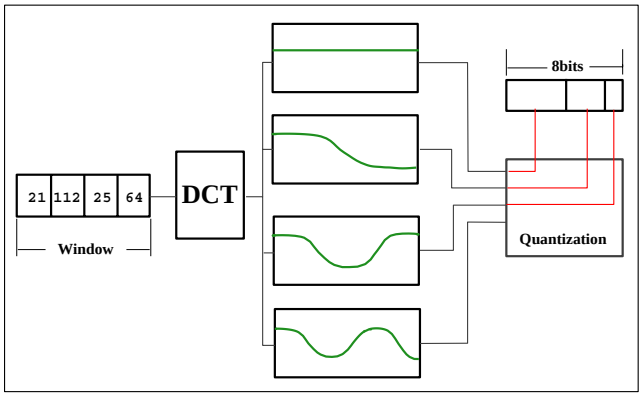


Figure 1: DCT-based encoding. In this example, the window size is 4 and only the first 3 coefficients are stored into an 8-bit value.

and

$$\alpha(u) = \begin{cases} \sqrt{\frac{1}{N}} & \text{for } u = 0 \\ \sqrt{\frac{2}{N}} & \text{for } u = 1, 2..., N-1 \end{cases}$$

where $C(u)$ are the transformed coefficients, N is the number of input samples, and $f(x)$ are the input samples. We use the DCT since it is known to have good information packing qualities and tends to have less error at the boundaries of a sequence [7]. Boundary performance is important in order to avoid discontinuities during the transition between blocks. Since our application compresses temporally, discontinuities would appear in the form of flashes between compressed sequences.

The encoding process is shown in Figure 1. First a window size is selected, which will be the length of the time sequence that will be encoded into a single value. The longer the window size, the greater the compression that can be achieved, at the expense of temporal accuracy. For each window of time-evolving scalars the DCT is applied. The result is a set of coefficients equal in number to the size of the window used. The first coefficient stores the average value over the window, and tends to be largest in value. The remaining coefficients store increasingly higher frequency components contained in the windowed sequence. These coefficients tend to represent decreasing amounts of signal as the frequency gets higher.

These coefficients are then quantized and combined into a single scalar value. Bits are adaptively allocated for each coefficient based on the variance of each coefficient [17]. Those coefficients with the highest variance are allocated more bits than those coefficients with low variance. Using this technique, bits are allocated based on the temporal characteristics of the windowed sequence of the data set. For example, a data set with minimal amounts of movement would use fewer bits to store the temporal changes in the data, allowing more bits to be used to more precisely represent the stationary values in the sequence. On the other hand, a sequence with high speed motion (low temporal coherence) would use more bits to encode this motion at the expense of precision for the static values.

Once bit allocation for the transformed coefficients is determined, the coefficients are quantized to their respective precision. Uniform quantization is not well suited for quantizing these coefficients since they often have fairly non-uniform distributions. Instead, quantization is performed using Lloyd-Max quantization [9, 12], which adaptively selects quantization levels that minimize mean square error. The quantized coefficients are then combined into a single scalar which is stored as an index in a paletted texture. Each encoded value represents a sequence of time-varying scalars that can be determined using the inverse DCT. Rather than using the inverse DCT, however, each palette entry is mapped to the average scalar value that the index represents for each time step. The encoding process is repeated for every window in time.

3.4 2-D texture & sub-volume optimizations

As described in the previous section, the quantization step is adapted based on the characteristics of the transformed coefficients. Since the temporal properties of a data set can vary widely across a volume, it is advantageous to adaptively quantize small sections of the volume at a time. Our volume rendering implementation uses axis aligned 2-D textures. To minimize the amount of error introduced in the quantization step, we encode each texture slice independently. This allows bit allocation to vary based on the temporal characteristics of each slice, as well as allowing for the quantization levels to vary based on the characteristics of the coefficients for each slice. We have found per-slice encoding shows noticeably fewer compression artifacts for a given bit rate. One negative effect of this process is that an uncompressed scalar value can be encoded into different compressed values depending on the slice in the volume. In practice, however, we have found this effect to be minimal, in part because quantization occurs in slices that are nearly perpendicular to the viewing direction, thus variations from slice to slice of a scalar value are softened by the volume rendering integral.

Usually, when bit allocation occurs, most bits are used for storing the average value over a windowed sequence. As a result, when the transition occurs between two compressed sequences, the shift in average value can cause a perceived jump in the animation. We therefore interleave the starting times of the windows for each slice. Figure 2 shows such an interleaving scheme. This decorrelates temporal transitions so the jump occurs during every frame but for interleaved slices in the volume, rather than the whole volume. This is analogous to interlaced video, except rather than being interlaced vertically, the textures are interleaved along the viewing direction. As with per slice quantization, the volume rendering integral helps to soften the interleaving effect.

For a transform window of length N, without interleaving an entire new compressed volume must be loaded every N frames. Since the loading of data across the graphics bus

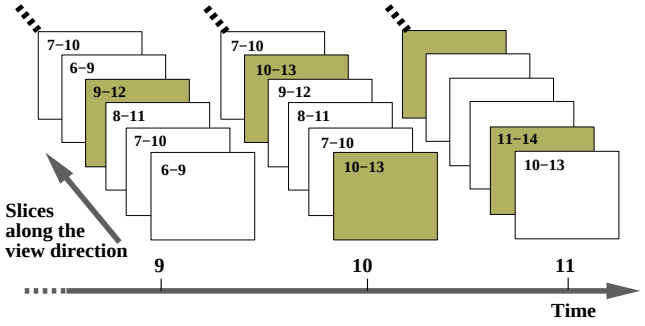


Figure 2: When 2-D texture interleaving is utilized, for every time step, every Nth 2-D texture is replaced starting with the t modulo Nth texture slice, where t is the time step and N is the compression ratio. In this example, N is four. The numbers on each slice indicate which time steps the texture stores. The shaded slice is the slice that is updated at time t.

Table 1: Three Test Data sets.

data set	time steps	spatial resolution
Turbulent vortex flow	100	128^3
Quasi-geostrophic turbulent flow	1492	256^3
Shock-bubble flow	265	$640\times256\times256$

is relatively slow, this can cause a substantial drop in frame rate every N frames. This problem can be solved by loading $\frac{1}{N}$ of the next compressed volume every frame, but requires storing a copy of the next volume in texture memory. This, however, is not necessary if the textures are interleaved, since for every frame $\frac{1}{N}$ of the volume can be flushed from texture memory and replaced with a new texture. Thus by amortizing data movement costs, interleaving allows for a more consistent frame rate without the expense of needing the texture memory to store a second compressed volume.

4 Test Results

Using our compression method we are able to render large, time-varying volumetric data sets at interactive rates. Table 1 lists the three data sets that have been used for our study. Color Plate 1 shows one frame from each data set. An animation of the turbulent vortex jets data displays a fairly random pattern over time as the vortices spread through the whole domain. In the QG turbulent flow data, we witness the formation of coherent turbulent structures akin to Jupiter's red spot. The shock-bubble flow data exhibits a slowly developing structure starting from one end of the domain and eventually reaching the other end of the domain.

Tables 2 and 3 show frame rates for different compression cases using the NCAR's quasi-geostrophic data set and the LBL's shock-bubble data set, respectively. We obtained these results with an AMD 1.2 GHz Athlon with 768 megabytes of

Table 2: Frame rates for rendering the QG data with different compression levels.

compression ratio	fps (time steps rendered)	
	in-core	out-of-core
8×	31.6 (280)	13.4 (1492)
4×	25.8 (140)	6.8 (1492)
2×	17.3 (70)	3.5 (1492)
1×	11.5 (35)	2.0 (1492)

Table 3: Frame rates for rendering the Shock-Bubble data with different compression levels.

compression ratio	fps (time steps rendered)	
	in-core	out-of-core
8×	11.7 (112)	5.8 (265)
4×	9.3 (56)	3.1 (265)
2×	6.3 (28)	1.6 (265)
1×	4.4 (14)	0.9 (265)

main memory and a NVIDIA GeForce3 based graphics card with 64 megabytes of texture memory. Figure 3 displays the storage configuration of our current testbed. Compressing each time step of a 256^3 data set takes between 5 and 15 seconds depending on the level of compression. Our implementation uses eight-bit paletted textures, although our technique could be applied to hardware that supports higher precision textures for encoding strategies that allocate more bits to each transformed coefficient. The results were obtained when rendering the volume to a 512×512 window with the volume occupying approximately $\frac{1}{3}$ of the window area.

If a compressed data set fits entirely in main memory, then the bottleneck in the rendering process is the transfer of textures from main memory to the graphics card. Compression helps with both of these limitations, increasing not only the number of time steps that fit in main memory, but also decreasing the amount of time necessary for transferring data across the graphics bus. If only one set of axis aligned textures is stored in main memory, then the number of time steps that can be stored in memory increases by a factor of three at the expense of the user not being able to view the data set from an arbitrary angle without swapping from disk.

One example of a large volumetric data set that we can render at interactive rates is the QG turbulent flow data set.

Table 4: Frame rates for rendering the 100 time steps of the Turbulent vortex flow data in-core with different compression levels.

compression ratio	fps
8×	76.1
4×	70.7
2×	51.6
1×	28.7

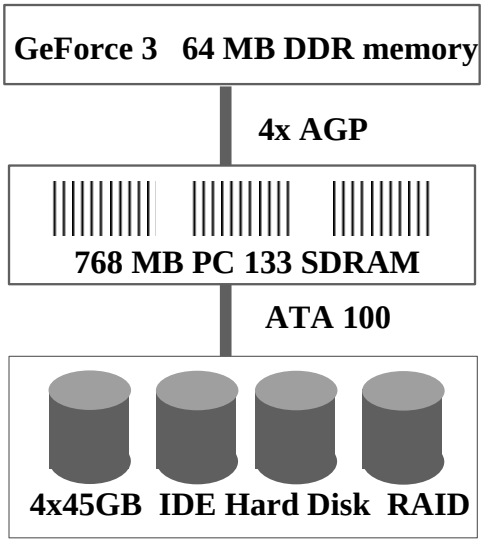


Figure 3: Storage configuration for a PC-based system for rendering large-scale time-varying data.

The QG calculations simulate large-scale motions in the Earth's atmosphere and oceans and are representative in size and complexity of many Eart Sciences turbulent fluid flow simulations. We can render 140 time steps of a 256^3 volumetric data set compressed by a factor of four at approximately 25.8 frames per second using 256 axis aligned textured polygons. If 128 axis aligned textured polygons are used, which requires transferring and drawing only half the data, the frame rate doubles and we can render 280 time steps.

Without compression, the same 140 time steps no longer fit in into main memory and would need to be swapped into main memory in an out-of-core manner. A subset of this uncompressed data that does fit in into main memory can be rendered at about 11.5 frames per second, compared to the 25.8 frames per second with compression. Although the amount of data transferred with compression is one fourth of that without, the frame rate does not scale linearly. This is caused by the time required to rasterize the textured polygons to the screen. The performance would scale more linearly if a graphics card with a higher fill-rate were used, or if the rendered volume were displayed at lower screen resolution.

Often a data set is too large to fit the desired number of time steps into main memory even with compression. In this case it is necessary to load and render the volume from disk. Compression can substantially decrease the amount of data that must be loaded for each frame, resulting in a noticeably higher frame rate, as shown in Table 2. For example, all 1492 time steps of the 256^3 QG data set can be rendered at 13.4 fps when compressed by a factor of 8 versus only 2.0 frames per second when rendered uncompressed from disc. Once the user finds a shorter temporal region of interest, that data can then be loaded into main memory and rendered at a faster frame rate, or higher image fidelity. Color Plate 3 shows visualizations of the QG data set for the same time step using different transfer functions defined through interactive exploration.

Table 5: NCAR quasi-geostrophic data set error

compression ratio	MAX	MIN	AVG
2x	0.000109	0.000005	0.000015
4x	0.000369	0.000009	0.000037
8x	0.000875	0.000012	0.000075

Table 6: Vortex data set error

compression ratio	MAX	MIN	AVG
2x	0.000199	0.000061	0.000136
4x	0.000423	0.000169	0.000304
8x	0.001147	0.000510	0.000856

By changing the window size used in the encoding step, the compression ratio and quality can be varied. Tables 5, 6, and 7 show the minimum, maximum, and average mean square error for each data set where scalar values have been normalized to be between 0 and 1. Color Plate 2 shows volumes that have been rendered using varying degrees of compression. As the amount of compression increases, some of the more subtle features as well as the faster moving features can become blurred. Thus, there is a distinct trade off between the compression ratio and rendering performance versus the quality of the compressed volume. This gives users a degree of flexibility in choosing compression ratios that best meet their needs. For example, if a scientist is interested in viewing a short time sequence at high quality, a lower compression ratio can be used. On the other hand, to view a very long sequence of data at high speeds, a higher compression rate can be selected. The scientist can combine compression ratios to preview a data set at a coarser temporal resolution and then view a specific time sequence of interest with less compression.

4.1 Discussion

Although our implementation does not use volumetric textures, our temporal encoding method does apply to 3-D volumetric textures. In particular we expect our method would work well with hardware that supports volumetric indexed textures with more than 256 entries. The use of 2-D textures has the limitation that for a volume to be viewable from an arbitrary angle, three copies of the textures must be stored

Table 7: LBL shock bubble data set error

compression ratio	MAX	MIN	AVG
2x	0.000017	0.000009	0.000013
4x	0.000040	0.000023	0.000032
8x	0.000082	0.000046	0.000065

for each of the principle viewing directions. However, this limitation is tempered by the fact compression can reduce the amount of texture storage required by beyond a factor of 3. When a volume is viewed out-of-core, storing extra compressed copies of the data set becomes much less of an issue since these copies are stored on disk. In addition, the use of 2-D textures provides a natural way to decompose a volume for adaptive quantization.

Since our system can render volumes from disk at interactive rates we feel it is very scalable with respect to the size of a data set temporally. With regards to size of the data set in the spatial domain, the amount of texture memory can be a limiting factor. Since our work compresses temporally, it does not reduce the amount of texture memory utilized to below what would be required to render a single static volume. With next generation graphics cards having increasingly larger amounts of texture memory, this limitation should become diminished. For out-of-core rendering, the cost of swapping textures from the graphics card to main memory is much lower than the cost of reading from disk, thus texture memory capacity restraints become less of a concern.

The use of compression by our methods presents two potential shortcomings that are worth addressing. First, since our compression scheme is lossy there is the potential for modest, but noticeable, image quality degradation that increases with the degree of compression. However, a moderate loss of image fidelity due to compression or other optimization strategies is an acceptable tradeoff for enabling interactive exploration of many temporal data sets, provided that the gross features of evolving structures is preserved as it is in our test cases [14, 2]. It is worth noting that many NCAR researchers commonly perform crude data reduction using simple zero-order subsampling in order to accommodate interactive exploration with the tools presently available to them. In essence, they have already demonstrated a willingness to sacrifice image quality to gain interactive exploration capabilities that are essential to maximizing scientific productivity [3]. Once a feature of importance is detected in the reduced data set, the full resolution data may be further analyzed if necessary. Second, compression requires additional storage (for maintaining both the raw and compressed versions of the data), and it takes time to perform the encoding. Similarly to loss of image fidelity, researchers are already bearing these costs by their use of subsampled data to achieve interactive rendering.

TSP Tree based methods reduce the amount of texture memory utilized by exploiting temporal and spatial coherence to reuse textures [18, 6]. They represent several similar textures as a single static texture. Our DCT based encoding method stores several time slices in terms of lower precision averages and differences stored in a single texel. Through palette manipulation, these texels dynamically represent several time slices. This compressed encoding comes at the expense of the numerical precision used to store these averages and differences. The method exploits temporal coherence by using more bits to represent the average value over a set of slices, but also reserves bits for storing the change over a set of slices. Our method could be combined with TSP based techniques to store textures at varying degrees of both spatial and temporal resolution.

Our method differs from compression methods supported in hardware like those supported with DXTC or S3TC [5] in that those methods compress 32-bit RGBA data, not 8-bit index data. Since these methods do not apply to paletted textures, it would be necessary to recompress the volume when changes to the color or opacity map are made. This stands in contrast to the use of paletted textures that allow for the interactive changing of the opacity and color map with no effect on frame rate. In addition, these methods compress RGBA texels by up to a factor of eight using four-bits per texel, while our method starts with eight-bit paletted textures and compresses them into four, two or one bits per texel, per frame.

5 Conclusions

We have presented a hardware texture assisted technique for rendering time-varying volume data, and demonstrated it with experimental results. This technique is very attractive to the scientists with whom we are working because of its low cost and interactive rendering rates. It is now feasible to put such a PC-based system on every scientist's desktop, making interactive data exploration accessible to a far broader group of scientists and engineers. Researchers can browse through data in a highly interactive manner, efficiently filtering unimportant from important features, to obtain valuable qualitative information about their data content. The compression scheme used is controllable and results in visualizations suitable for interactive data exploration.

We plan to improve our hardware-assisted technique in many ways. One important goal is to make it possible for the scientists to conduct interactive exploration of data obtained from even larger data sets than those discussed, such as the 512^3 Earth Sciences turbulence calculations many of our researchers are beginning to investigate. Furthermore, 1024^3 problem domain calculations are on the horizon. To address these extremely large data sets we plan to investigate an extension of our hardware-assisted rendering technique with a PC cluster, develop more intuitive data interaction mechanisms in both the spatial and temporal domains, and design effective hardware-assisted shading techniques.

Acknowledgements

This work has been sponsored by the National Science Foundation under contracts ACI 9983641 (PECASE Award) and ACI 9982251 (LSSDSV). The quasi-geostrophic turbulent flow data set were generated by Jeffrey Weiss and Clive Bail-

lie at University of Colorado at Boulder, James McWilliams at University of California at Los Angeles, along with Irad Yavneh at Technion. The shock-bubble flow data set was provided by scientists at the Lawrence Berkeley National Laboratory. We obtained the turbulent vortex flow data set through the VIZLAB of CAIP at Rutgers University. The authors are grateful to them for providing data sets for our study.

References

[1] Ext_shared_texture_palette, September 1997. OpenGL Extension Registry,

http://oss.sgi.com/projects/ogl-sample/registry/EXT/shared_texture_palette .txt.

[2] BRUMMELL, N. Senior Research Associate at the University of Colorado, Personal Communication, 2001.

[3] CLYNE, J., SCHEITLIN, T., AND WEISS, J. Volume Visualizing High-Resolution Turbulence Computations. *Theoretical and Computational Fluid Dynamics* (1998), pp. 195–211.

[4] CLYNE, J., AND DENNIS, J. Interactive direct volume rendering of time-varying data. In *In Proceedings of Data Visualization '99* (May 1999), pp. 109–120.

[5] DOMINE, S. Using texture compression in OpenGL, May 2000. http://www.nvidia.com.

[6] ELLSWORTH, D., CHIANG, L., AND SHEN, H.-W. Accelerating time-varying hardware volume rendering using tsp trees and color-based error metrics. In *Proceedings of 2000 Symposium on Volume Visualization* (2000), ACM SIGGRAPH, pp. 119–128.

[7] GONZALEZ, R., AND WOODS, R. *Digital Image Processing*. Addison Wesley, 1992.

[8] HAIMES, R. Unsteady visualization of grand challenge size cfd problems: Traditional post-processing vs. co-processing. In *Proceedings of the ICASE/LaRC Symposium on Visualizing Time-Varying Data* (1996), pp. 63–75. NASA Conference Publication 3321.

[9] LLOYD, S. P. Least squares quantization in PCM. *IEEE Transactions on Information Theory IT-28* (March 1982), 129–137.

[10] MA, K.-L., AND CAMP, D. High performance visualization of time-varying volume data over a wide-area network. In *Proceedings of Supercomputing 2000 Conference* (November 2000).

[11] MA, K.-L., AND SHEN, H.-W. Compression and accelerated rendering of time-varying volume data. In *Proceedings of the 2000 International Computer Symposium - Workshop on Computer Graphics and Virtual Reality* (December 2000), pp. 82–89.

[12] MAX, J. Quantizing for minimum distortion. *IRE Transactions on Information Theory IT-6* (March 1960), pp. 7–12.

[13] PARKER, S. G., AND JOHNSON, C. R. SCIRun: A Scientific Programming Environment for Computational Steering. In *Proceedings of the 1995 Supercomputing Conference* (1995). http://scxy.tc.cornell.edu/sc95/proceedings/.

[14] RAST, M. Scientist at the National Center for Atmospheric, Personal Communication, 2001.

[15] REZK-SALAMA, C., ENGEL, K., BAUER, M., GREINER G., AND ERTL, T. Interactive volume rendering on standard pc graphics hardware using multi-textures and multi-stage rasterization. In *Proceedings of SIGGRAPH/Eurographics Graphics Hardware Workshop 2000* (2000).

[16] ROWLAN, J., LENT, E., GOKHALE, N., AND BRADSHAW, S. A distributed, parallel, interactive volume rendering package. In *Proceedings of the Visualization '94 Conference* (1994), pp. 21–30.

[17] SAYOOD, K. *Introduction to Data Compression*, second ed. Morgan Kaufmann Publishers, Inc., 2000.

[18] SHEN, H.-W., CHIANG, L., AND MA, K.-L. A fast volume rendering algorithm for time-varying field using a time-space partitioning (tsp) tree. In *Proceedings of Visualization '99* (1999), pp. 371–377, IEEE Computer Society Press, Los Alamitos, CA.

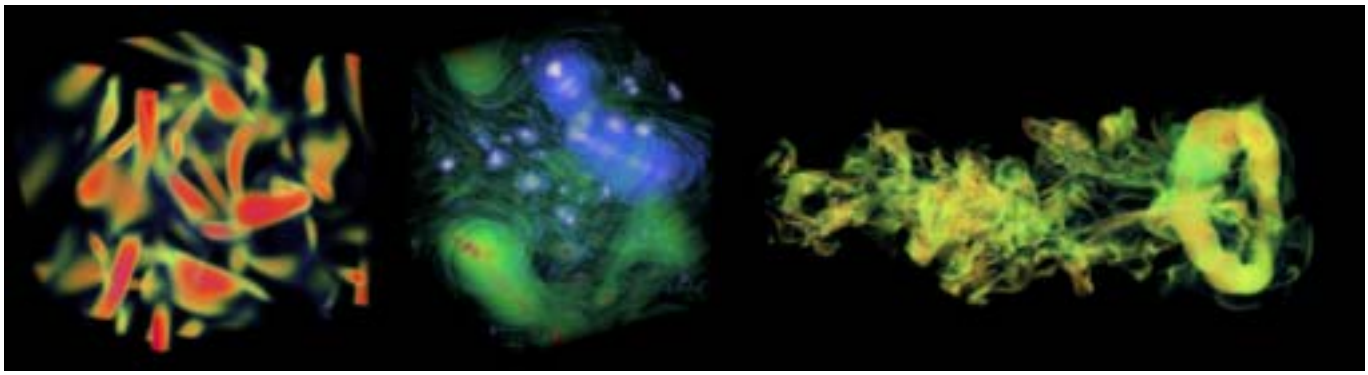
[19] SHEN, H.-W., AND JOHNSON, C. Differential volume rendering: A fast volume visualization technique for flow animation. In *Proceedings of the Visualization '94 Conference* (October 1994), pp. 180–187.

[20] SUTTON, P. AND HANSEN, C. D. Isosurface extraction in time-varying fields using a temporal branch-on-need tree (T-BON). In *Proceedings of IEEE Visualization '99 Conference* (October 1999), pp. 147-153.

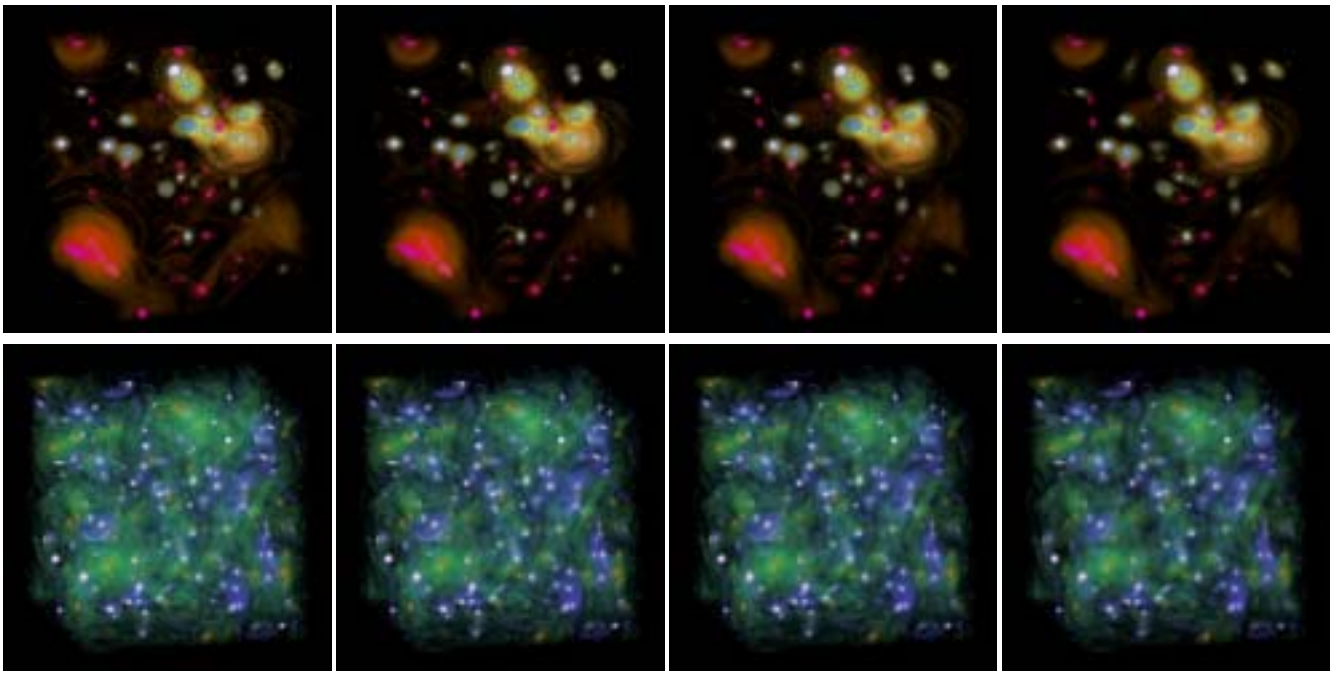
[21] WESTERMANN, R. Compression domain rendering of time-resolved volume data. In *Proceedings of the Visualization '95 Conference* (1995), pp. 168–174.

[22] WILHELMS, J., AND VAN GELDER, A. Octrees for faster isosurface generation. *ACM Transactions on Graphics 11*, 3 (July 1992), pp. 57–62.

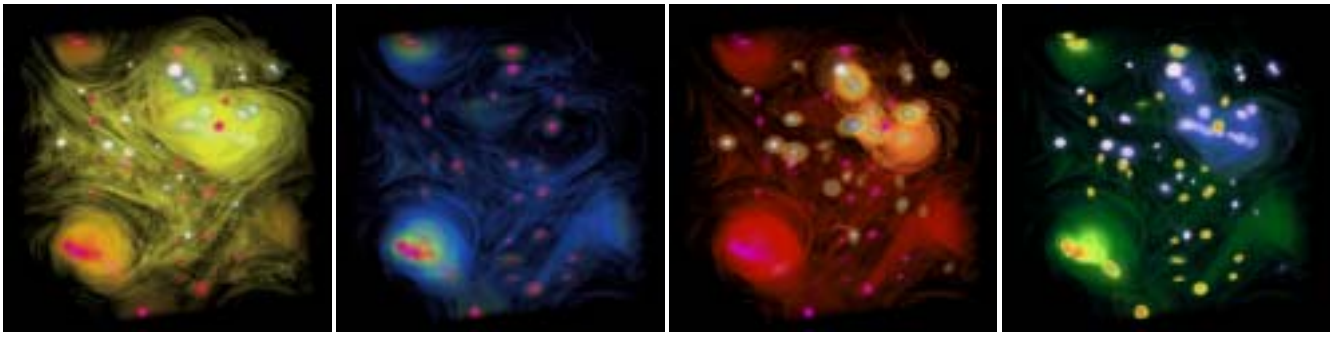
[23] WILHELMS, J., AND VAN GELDER, A. Multi-dimensional trees for controlled volume rendering and compression. In *Proceedings of the 1994 Symposium on Volume Visualization* (October 1994), pp. 27–34.



Color Plate 1: One selected frame for each data set.



Color Plate 2: Visualizations of the quasi-geostrophic data set at difference compression levels. As the level of compression increases, some of the finer features become blurred. Top row: time step 980. Bottom row: time step 210. Left to right: 1x 2x 4x 8x compression



Color Plate 3: Selected visualizations of the quasi-geostrophic data set produced by varying transfer functions.