Interactive Direct Volume Rendering of Time-Varying Data

John Clyne and John M. Dennis

Scientific Computing Division
National Center for Atmospheric Research

Abstract. Previous efforts aimed at improving direct volume rendering performance have focused largely on time-invariant, 3D data. Little work has been done in the area of interactive direct volume rendering of time-varying data, such as is commonly found in Computational Fluid Dynamics (CFD) simulations. Until recently, the additional costs imposed by time-varying data have made consideration of interactive direct volume rendering impractical. We present a volume rendering system based on a parallel implementation of the Shear-Warp Factorization algorithm that is capable of rendering time-varying 128^3 data at interactive speeds.

1 Introduction

In recent years Direct Volume Rendering (DVR) has proven to be a powerful tool for the analysis of time-varying, three-dimensional CFD datasets associated with the aerospace, atmospheric, oceanic and astrophysical sciences. The benefits of DVR, probabilistic data classification [6] and direct projection of volume samples, are key to producing insightful visualizations of simulation results, rich with amorphous and fluid-like features [5].

Visual data exploration is an inherently interactive process. In order to fully exploit the power of DVR as an analysis tool, interactive frame rates must be achieved. Static images or animations produced through a batch process, representing complex 3D phenomena, may be of limited value. Invariably, features of interest are obscured, improperly classified, or poorly lit. Much work has been done in the area of accelerating DVR methods for static data [8, 11, 18, 12, 15, 2, 13]. Interactive volume rendering of single time-steps is now within the reach of many researchers. However, temporal animations are typically produced through a batch process. Time-varying datasets impose additional costs that have made interactive rendering difficult to achieve. The principal culprit is I/O. Rendering even a moderately sized (256^3) dataset made up of 8-bit voxels at a frame rate of 10 Hz requires an input pipe capable of delivering a sustained bandwidth of over 150 MB/sec.

In this paper, we detail our efforts at making interactive DVR of time-varying data possible. We present a volume rendering system based on a parallel implementation of Lacroute's Shear-Warp Factorization algorithm [9]. The remainder of this paper is organized as follows: In Section 2 we discuss research efforts

related to our own. In Section 3 we present an overview of the serial and parallel implementations of the Shear-Warp algorithms, and we discuss steps we have taken to support time-varying data. Our implementation is presented in Section 4. In Section 5 we address performance. In Section 6 we draw conclusions.

2 Related Work

There have been numerous successful efforts in the area of performing real-time DVR of moderately-sized (256^3) static data sets [3, 7, 2, 8, 13, 15]. The approaches taken can be classified into hardware and software based methods. Hardware methods include the development of custom hardware, dedicated to performing volume rendering [7, 13], and the clever exploitation of conventional polygon engines in order to accelerate the volume rendering task [3, 17]. Interactive volume rendering can be achieved via software methods by mapping serial algorithms onto general-purpose parallel computers. The increasingly wide-scale availability of commercial multiprocessors as general compute platforms has made this strategy popular and practical as evidenced by the abundance of research published on this subject [11, 18, 2, 8, 15].

More closely related to our efforts of visualizing time-varying data is the work of Shen and Johnson [16], and Chiueh and Ma [4]. Shen and Johnson address the I/O problem by applying data compression. A preprocessing step is performed on the volume data to create a *differential file.* The first time-step in a series serves as a basis function and is recorded in its entirety to the *differential file.* For each subsequent time-step, only the differential information between the time-step being processed and the preceding time-step is recorded. Shen and Johnson report on a second optimization as well: they adapt their volume renderer to only update those pixels that are affected by voxels that have changed between the preceding and current time-step.

More recently, Chiueh and Ma take another approach, pipelining the multi-timestep rendering process and exploiting a combination of *intra-volume* and *inter-volume* parallelism [4]. Conventional parallel volume renderers operating on static data dedicate multiple processors to render a single data volume. Chiueh and Ma term this *intra-volume* parallelism. Alternatively, *inter-volume* parallelism is employed by simultaneously rendering multiple data volumes on multiple processors. In the extreme case, the number of time-steps rendered in parallel equals the number of processors available.

Though both of these techniques are capable of greatly accelerating overall rendering rates, their utility as interactive tools may be restricted. Chiueh's and Ma's approach suffers from the pipeline effect: changes in viewing parameters invalidate the pipeline, requiring it to be drained and restarted. These restarts, which result in inter-frame delays, may be acceptable for short pipelines. For longer pipelines, this latency may be great enough to significantly impact interactivity. The differential volume renderer is also impacted by viewing parameter changes, such as changes in viewpoints or classification, which force the entire

image to be updated. Additionally, differential volumes do not readily support random access of time-steps.

Our goal is to produce a portable, high-frame-rate, low-latency rendering system that is unencumbered by these restrictions and is thus better suited for interactivity. We exploit a number of advances in computing to accomplish this task. We implement a fast, parallel rendering algorithm, which includes several optimizations that address time-varying data. Our principal optimizations include a fast, table-based gradient estimator and the overlapping of I/O and computational tasks. We utilize a commercially-available, shared-memory multiprocessor. We take advantage of readily available, inexpensive, high-speed disk arrays to help address I/O bandwidth requirements. We utilize commodity, high-bandwidth (100Mb/sec) networking to deliver imagery from our rendering platform to a display host.

3 Shear-Warp Factorization Algorithm

The Shear-Warp Factorization algorithm operates by applying an affine viewing transformation to transform object space into an intermediate coordinate system that Lacroute calls *sheared object space*. Sheared object space is defined by construction such that all viewing rays are parallel to the third coordinate axis and perpendicular to the volume slices. Volumes are assumed to be sampled on a rectilinear grid. For a parallel projection, the viewing transformation is simply a series of translations. Note that the resampling weights are invariant within each slice because all voxels are translated by the same amount within each slice. After the slices have been translated and resampled, they may be efficiently composited in front-to-back order using the *over* operator to produce an intermediate, warped, *baseplane* image. The distorted, baseplane image must then be resampled into final image space.

Parallel implementations of the Shear-Warp algorithm, demonstrating good speedup, have been reported using both message-passing [2, 15] and shared-memory [8] systems. Our target platform is an HP Exemplar shared-memory multiprocessor. We therefore elected to extend an existing serial implementation of the algorithm, the VolPack library [10], following Lacroute's successful parallelization of this same package [8]. We discuss our parallel implementation, pointing out the deviations between Lacroute's efforts and our own. We next discuss steps taken aimed at accommodating time-varying data volumes.

The VolPack library implements two rendering algorithms that are of interest to us. The first operates on preprocessed, run-length encoded (RLE) data volumes. The pre-processing step computes the view-independent opacity and gradient information for the data samples. Viewing and shading parameters may subsequently be changed, but classification information is fixed. There are two advantages to the RLE algorithm: 1) Opacity and gradient information do not need to be computed during rendering. 2) The RLE data structure permits the algorithm to take advantage of data coherency by skipping over transparent voxels. For many datasets this optimization can yield a significant savings. The

second algorithm simply operates on raw data. No preprocessing is performed. Consequently, the renderer does not utilize volume data coherency. For static data, the RLE algorithm generally has far superior performance than the raw data algorithm. However, as we discuss later, the raw data rendering algorithm has some attractions for time-varying data, namely, reduced I/O requirements.

Execution time for the RLE algorithm is dominated by two calculations: 1) projection of the volume into the baseplane image and 2) warping the baseplane image into the final image. The raw data algorithm must perform the additional step of computing gradient vectors and determining voxel opacity values. We parallelize each of these computational phases, synchronizing processors in between.

Projection of the 3D volume, which involves resampling and compositing the volume slices, is the most computationally expensive task in the RLE algorithm and also the most challenging to efficiently implement in parallel. To minimize processor synchronization, a task partitioning based on an image-space decomposition of the baseplane image is employed. Each processor is responsible for computing a number of scanlines in the baseplane image. The baseplane image is partitioned into small groups of contiguous scanlines. Processors are initially statically assigned groups of scanlines in cyclical fashion. As the calculation proceeds, dynamic task stealing is utilized to perform load balancing. Each processor maintains its own queue of groups of scanlines. As soon as a processor finishes all of its work, it tries to steal a group of scanlines from its neighbors. This process continues until projection is complete. Our task-stealing implementation differs from Lacroute's in that processors only look to a limited number of neighbors for additional work. While this may lead to load imbalance under rare pathological conditions, it simplifies termination logic and reduces synchronization requirements.

Resampling the baseplane image typically represents 5% of the total calculation for the RLE renderer. We partition the workload by dividing the post-warped image into P groups of adjacent scanlines (where P is the number of processors) and statically assign one group of scanlines to each processor. There are no pixel interdependencies in the post-warped image, thus no synchronization is required. Our implementation differs from Lacroute's which uses an interleaved assignment of tile-shaped tasks, in an effort to provide load balancing, instead of groups of contiguous scanlines.

The raw data algorithm requires the additional step of classifying the data and computing normal vectors. Classification is performed via a user-defined lookup table. Normal vectors are estimated using central differences. We easily parallelize these operations by partitioning the volume into contiguous blocks and assigning each processor a single block.

3.1 Time-Varying Data

Time-varying data incur additional rendering costs primarily due to I/O. The principal technique we employ to address these costs is to "hide" the I/O behind the rendering calculations by using a separate I/O thread to double-buffer *reads*

of volume files. A parallel memory copy is then employed to move the input data from buffer space to user space. We note that double-buffering entails keeping two copies of data in memory, which may limit the dataset's size on machines with smaller memories.

There are tradeoffs to be considered between the RLE and raw data rendering algorithms. The RLE algorithm is typically faster because it exploits data coherency. However, the RLE data volumes, which contain three copies of the data and include gradient information, may be substantially larger than the original raw data volumes and have correspondingly greater I/O requirements. The exact size of the RLE volume is largely determined by the user-defined classification function: the more voxels whose opacity is mapped to zero, the smaller the RLE volume becomes. Finally, there is another issue that may make the raw data algorithm more attractive: RLE datasets prohibit changes in classification.

4 System Overview and Implementation

We have developed a comprehensive rendering system based on the extensions to the VolPack library discussed above. *Volsh* is an interactive application with an X11-based graphical user interface (GUI). Since large multiprocessor configurations typically do not have directly attached display devices, Volsh is implemented as two separate UNIX processes communicating via internet-domain sockets. One process runs on the parallel machine, performing rendering and managing the GUI. The second process runs on the display host and is responsible for ingesting post-warped imagery transmitted by the renderer, resampling the post-warped imagery to final imagery at the desired screen resolution, and posting the imagery to the frame buffer. The resampling and posting operations are performed via the OpenGL API. The rendering host and display host are networked via a 100Mb/sec FDDI ring. The network bandwidth requirements between the renderer and the display process are non-negligible. To minimize bandwidth requirements, the renderer transmits reduced-resolution post-warped imagery (typically 256x256 pixels) to the display system. As with the input stream, we dedicate a single thread to double-buffer the output image stream.

4.1 Hardware

The results reported in Section 5 were collected from an HP Exemplar SPP2000 X-Class technical server. The X-Class Exemplar SPP2000 is a cache coherent Non Uniform Memory Access (cc-NUMA) scalable shared memory computer consisting of as many as 512 180-MHz Hewlett-Packard PA-RISC 8000 processors, each with a 1 MB data and instruction cache. The Exemplar SPP2000 hardware architecture can be thought of as a tightly coupled cluster of Symmetric Multi-Processor (SMP) "hypernodes." Each hypernode consists of up to 16 PA-RISC 8000 processors, on 8 dual-processor boards, each connected to 8 memory boards via a crossbar interconnect.

We report results obtained from a single hypernode configured with 16 processors, 2 GB of memory, and a high-bandwidth disk array. The disk array is a level 0 RAID, implemented with software striping, consisting of 9 drives striped across 3 UltraSCSI controllers. The sustained bandwidth that we have measured for *read* operations on the disk array is 80MB/sec.

4.2 Software

Volsh is implemented in C and C++. Parallelization is accomplished using POSIX threads (pthreads), making the software portable to most shared-memory architectures.[1]

Voxels in raw datasets are 8-bit quantities. RLE voxels are 32-bits: the original sample value is represented by 8 bits, the remaining bits are used to represent gradient magnitude (8 bits) and an encoded representation of a gradient (13 bits). Shading is performed using a shade tree [1] representing the Phong lighting equations. The shade tree is implemented as a lookup table, indexed by the voxel's scalar value and encoded normal vector. The calculation of the shade tree itself is entirely based on lookup tables and does not contribute measurably to the overall computation time.

Fast Gradient Estimator Normal vectors are estimated prior to projection by applying the 6-neighborhood central-difference method. Before the normal vector can be applied in lighting equations and voxel classification, it must be normalized and its magnitude must be calculated. Computing these quantities directly requires a square root and three divides. We avoid these expensive operations by transforming the Cartesian coordinates into normalized, spherical coordinates and then mapping the longitude and latitude angles, λ and ϕ, back into normalized, rectangular form. Both of these transformations are performed via lookup tables. The mapping from Cartesian to spherical coordinate space and its inverse are given by equations (1) and (2) below, respectively.

$$\begin{aligned} \lambda &= tan^{-1}\frac{y}{x} \\ \phi &= tan^{-1}\frac{z}{\sqrt{x^2+y^2}}, \end{aligned} \tag{1}$$

$$\begin{aligned} x &= cos\lambda \times cos\phi \\ y &= sin\lambda \times cos\phi \\ z &= sin\phi. \end{aligned} \tag{2}$$

The angular distances returned by equation (1) are restricted to the range $-\pi \leq \lambda < \pi$ and $-\pi/2 \leq \phi \leq \pi/2$. Because we are using 8-bit voxels, the

[1] Due to performance problems with the Exemplar pthread implementation, we were forced to use HP's native thread library when benchmarking on the Exemplar.

domain for each Cartesian coordinate a_i in equation (1) is restricted to the range $-255 \leq a_i \leq 255$. Consequently, lookup tables small enough to fit into cache may be readily constructed and used to evaluate the arc tangent and square root operations. Note that using a lookup table to normalize the Cartesian coordinates directly would require a table with 255^3 entries.

The arc tangent lookup tables return quantized, integer, angular measurements. We represent λ as a 7-bit quantity and ϕ as a 6-bit quantity, giving us a 13-bit encoded normal which can subsequently be mapped into normalized, floating-point, Cartesian space using a lookup table in place of equation (2). Empirical evidence suggests that the 13-bit normal representation is adequate and additional precision is not warranted (see Plate 1, top). The gradient's magnitude, used during classification to attenuate opacity, is approximated using a *Manhattan Distance* approximator, accurate to $\pm 8\%$ [14].

5 Performance

We use a number of different datasets to evaluate the performance of our rendering system. The Quasi-geostrophic (QG) data are computer simulation results depicting turbulence in the Earth's oceans. These data and the classification functions chosen for them are of particular interest to us in exploring performance characteristics because earlier time-steps are very dense, dominated by amorphous (semi-transparent) features, and exhibit little spatial coherence. Later time-steps are relatively sparse and opaque, exhibiting substantially more coherence (see Plate 2). Our second dataset was produced from a simulation of the wintertime stratospheric polar vortex. The polar vortex (PV) data are relatively sparse and opaque, exhibiting a great deal of coherence throughout the entire simulation (see Plate 1 (middle), (bottom)). Lastly, we include some results from the familiar UNC Chapel Hill Volume Rendering Test Dataset. Though these medical data are not time-varying, their performance characteristics are well-known to the volume rendering community (see Plate 1 (top)). Table 1 lists the datasets, their resolution, the size of each raw time-step, and the average size of each RLE time-step. We note from Table 1 that the relative size of raw and corresponding RLE datasets provide a measure of spatial coherency. The larger the RLE dataset relative to the raw dataset, the less coherency exists. The sizes of the RLE datasets also provides a measure of how much work is required to render them; larger RLE files require increased rendering time.

In the experiments reported below, all data are read directly from disk. The kernel buffer cache was flushed prior to running each experiment. Unless otherwise specified, all results are for Phong-illuminated, monoscopic, three-channel (RGB), 256x256 resolution imagery, rendered with a parallel viewing projection. Lighting is provided by a single light source with fixed lighting parameters.

5.1 Static Data

Figure 1 plots rendering rate in frames per second vs. number of processors for static data, using both the RLE and raw datasets. The frame rates shown are

the averages for a 180-frame animation, with a 2-degree rotation about the Y axis between each frame. The time to display the image is not included. We see from Figure 1 that for static data, the RLE data algorithm performs significantly faster than the raw data algorithm. However, there are variations in the relative differences in performance. The highly coherent PV datasets exhibit the greatest increase in performance between the raw and RLE data (roughly an order of magnitude improvement, from 1.7Hz to 20Hz on 15 processors). Conversely, the QG dataset does not benefit as much from the RLE encoding (about a factor of 2 improvement from 5Hz to 10Hz on 15 processors). We also observe that the performance of our implementation on the Brain dataset is comparable to that reported by Lacroute [8], both in terms of speedup (approximately 11) and maximum frame rate (approximately 10Hz). [2]

Lastly, we note that all the performance curves dip after the 15-processor run. This event occurs when processors are forced to perform non-rendering tasks. For static data, these non-rendering tasks are comprised of normal UNIX system activities. In subsequent experiments involving time-varying data, we dedicate two processors for double-buffering (one for input and one for output), leaving 14 processors available for rendering.

5.2 Time-Varying Data

In this section we discuss our results with time-varying data. Unlike our static-data experiments, the experiments reported below include the time to display imagery. Unless otherwise stated, both the ingestion of data volumes and output of image streams are double buffered.

Figure 2 plots the frame rate in frames per second vs. the number of processors for the QG and PV datasets using both RLE and raw data. We observe that each of the RLE curves go flat at some point. This occurs when the task becomes I/O bound as evidenced in Figure 3 by the growth of the *i_time* parameter. The *i_time* parameter depicts the unmaskable, double-buffered I/O time for reading data. It is the amount of time the rendering processes must wait for input. When the task is computationally (render) bound, *i_time* is close to zero. As the rendering process is sped up through the addition of processors, the task can become I/O bound, and *i_time* will grow. The less voluminous raw data do

[2] Lacroute's benchmarks were performed using only a single color channel.

Dataset	Resolution	Raw Size (MB)	Average RLE Size (MB)
Brain128	128x128x84	1.38	3.34
Brain256	256x256x167	10.94	16.27
QG128	128x128x128	2.10	10.78
PV128	128x128x75	1.23	1.92
PV256	256x256x149	9.76	13.70

Table 1. Datasets, their respective voxel resolutions, individual time-step sizes for raw data, and average time-step sizes for RLE data.

not become I/O bound in this plot, and their corresponding performance curves display better speedup.

We also observe that as we would expect from our static data measurements, the highly coherent PV RLE data perform considerably better than the PV raw data. However, the QG RLE data initially perform much better than the QG raw data, but in higher processor runs the difference in performance is not as great. The rising QG raw curve is rapidly approaching the flat RLE curve. The QG RLE data very quickly become I/O bound because of their large size relative to the raw data. Witness the growth of the *i_time* parameter for QG RLE data in Figure 3 after 6 processors. Adding additional processors may speed up rendering, but it does not affect the dominating I/O time. On the other hand, though the raw data algorithm is more computationally expensive, the I/O requirements are much lower. Adding processors continues to reduce the rendering and gradient calculation time, which dominate the lower-processor-number runs for the raw data.

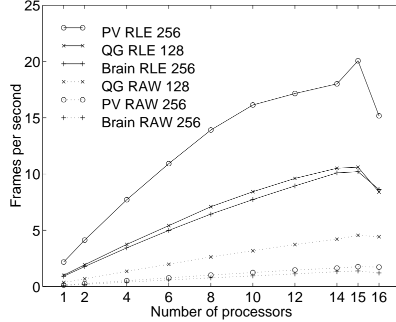


Fig. 1. Frame rates of static data runs using RLE (solid lines) and raw datasets (dashed lines)

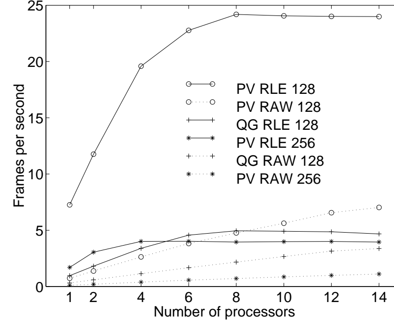


Fig. 2. Frame rates of time-varying data runs using RLE (solid lines) and raw datasets (dashed lines)

5.3 Double Buffering

Figure 4 shows the effects on overall rendering performance of overlapping the I/O and computational tasks for the QG RLE dataset. The ideal speedup that may be achieved is given by

$$s = \frac{T_{input} + T_{output} + T_{compute}}{max(T_{input}, T_{output}, T_{compute})},$$

where T_{input}, T_{output}, and $T_{compute}$ are the times required for reading data, displaying imagery, and performing the computational tasks, respectively. We observe from this equation that the maximum possible speedup is three, and this

maximum occurs when all T are equal. In our experiments, the input requirements are fixed and completely determined by the size of the volume dataset and the available bandwidth of the storage device. Similarly, the output bandwidth requirements, which are relatively low, are fixed and determined by the resolution of the post-warped image and the bandwidth of the display device (in our case, a FDDI-attached host).

Comparing the single-buffered and double-buffered runs in Figure 4, we observe that on fewer processors, double-buffering has little effect on overall performance which is dominated by the rendering calculation. For the single-processor run, the input and output time combined represent only about 25 percent of the total run-time. Hence the best speedup we can realize by double-buffering is only $1/0.75 \approx 1.3$. However, as the number of processors is increased, rendering time is reduced. By the six-processor run, rendering time and input time are nearly equal, and we see a speedup over the single-buffered performance of about two. Adding even more processors reduces rendering time, but does not affect I/O. Hence the overall performance indicated by the double-buffered curve goes flat after six processors. Contrarily, the performance indicated by the single-buffered curve continues to improve beyond six processors, but never reaches the performance of that of the double-buffered runs. Lastly, we observe that double buffering has completely masked the cost of output. Though insignificant on fewer processors, for greater processor runs the output time has a measurable impact on overall performance.

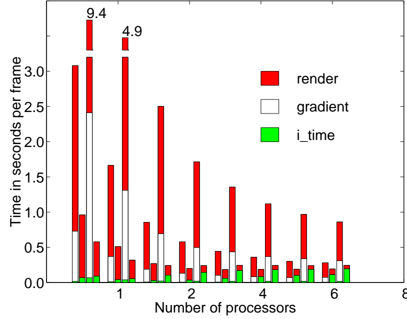


Fig. 3. Timing distribution of rendering process showing non-maskable read time, *i_time*; rendering time, *render*; and gradient calculation time for the raw data algorithm, *gradient*. Shown for each processor from left to right are: QG128 Raw, QG128 RLE, PV256 Raw, and PV256 RLE

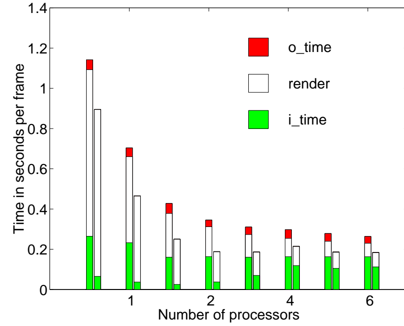


Fig. 4. Effects of double buffering on the QG RLE dataset. The first and second time distribution bar for each processor show single-buffered and double-buffered performance, respectively. The I/O times, *i_time* and *o_time*, include the overhead of double-buffering and non-maskable I/O

6 Conclusions

We have developed a Direct Volume Rendering system capable of interactively rendering moderately sized, time-vary datasets. The system is based entirely on commercially available components. The performance for a given volume resolution is largely determined by the amount of spatial coherency that exists within the data and the user-chosen classification function. For highly coherent, sparse data the pre-classified, RLE-encoded volumes are comparable in size to the raw data, and the RLE algorithm performs best. For data that are dense and exhibit little spatial coherency, the increased I/O costs associated with the more voluminous RLE-encoded files may make the raw data algorithm more attractive if sufficient processing power is available.

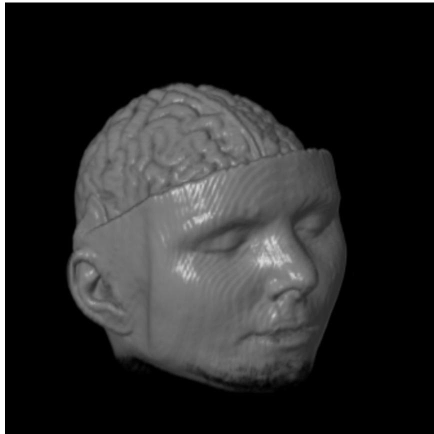
Double buffering was shown to be an essential component of our system. We have seen that the technique is maximally effective when the processing requirements of all the overlapped tasks are similar. In our experiments, the performance of the two I/O tasks is fixed. The third task we chose to overlap, the computational task, can be sped up by employing additional processors. For fewer processors, the computational task is the most expensive of the three overlapped tasks and can therefore mask the cost of I/O. As we increase parallelism, the computation task may be sped up until it no longer dominates. Once I/O begins to dominate, the limit of the benefit of adding processors has been reached. For the RLE data, this limit was generally reached on a relatively few processors. For raw data, which are computationally more expensive to render, the computational limit was never reached; the rendering task remained compute bound. We can view this result as somewhat encouraging. Given that improvements in microprocessor technology far outpace improvements in storage bandwidth, we speculate that the performance of the raw data algorithm may surpass the RLE data algorithm in the near future.

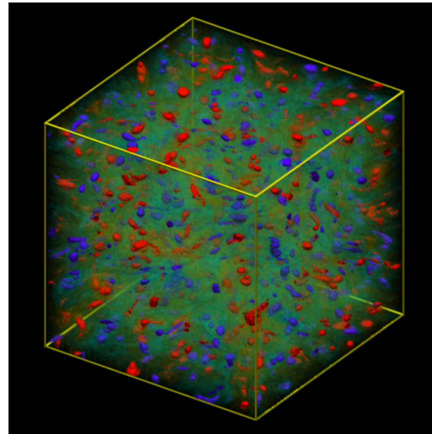
Although we predict that next-generation processors may permit the raw data algorithm to outperform the RLE algorithm, for the present the converse is true for the computing platform we tested. In the case that time-varying raw data cannot be rendered at sufficient frame rates, and the user is forced to work with RLE data, we do not view the inability to change classification of RLE data as a serious detriment. Our observations have been that researchers perform classification while working on a small number of time-steps, one static time-step at a time. Once a satisfactory classification has been arrived at, the researcher will then begin to explore the data temporally. At this point the classification function has been chosen, and the data may be RLE encoded for improved performance.

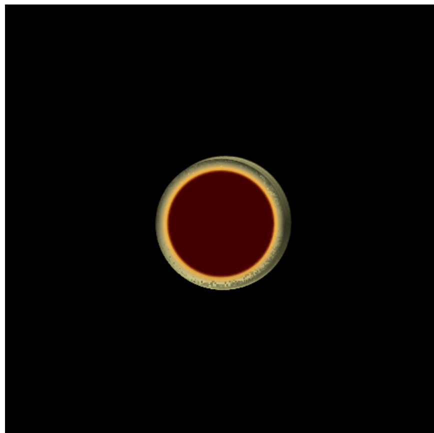
Lastly, we note that the RLE method in our current implementation stores three copies of the data for each time-step, one for each principal viewing axis. With some modification, significant savings in I/O requirements could be realized by loading only the RLE encoding required by the current viewing direction. Though this complicates the double-buffering scheme somewhat, we believe the potential performance improvements are well worth pursuing.

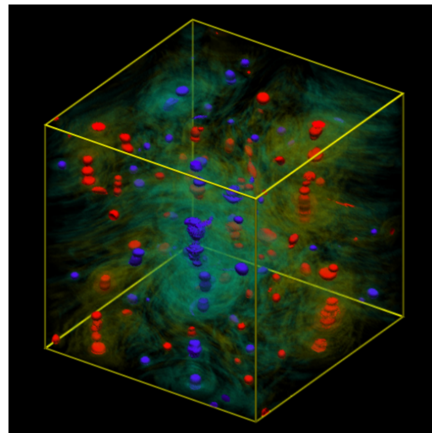
References

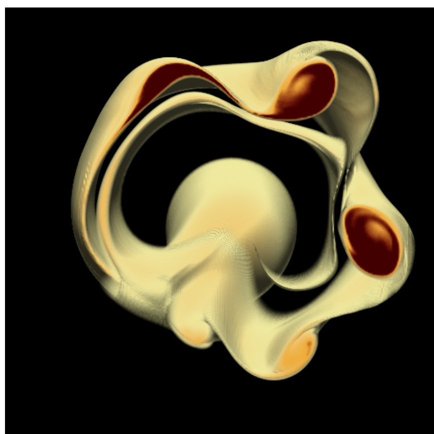
1. G. Abram and T. Whitted. Building block shaders. In *Computer Graphics*, pages 283–288, Dallas, TX, August 1990.
2. M. B. Amin, A. Grama, and V. Singh. Fast volume rendering using an efficient, scalable parallel formulation of the shear-warp algorithm. In *Parallel Rendering Symposium*, pages 7–14, Atlanta, GA, October 1995.
3. B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Symposium on Volume Visualization*, pages 91–97, Washington, D.C., October 1994.
4. T.-C. Chiueh and K.-L. Ma. A parallel pipelined renderer for the time-varying volume data. Technical Report 97-90, ICASE, Hampton, VA, December 1997.
5. J. Clyne, T. Scheitlin, and J. Weiss. Volume visualizing high-resolution turbulence computations. *Theoretical and Computational Fluid Dynamics*, 1998.
6. R. A. Drebin, L. Carpenter, and P. Hanrahan. Volume rendering. *Computer Graphics*, 22(4):51–58, August 1988.
7. K. Knittel and W. Straber. A compact volume rendering accelerator. In *Symposium on Volume Visualization*, pages 67–74, Washington, D.C., October 1994.
8. P. Lacroute. Analysis of a parallel volume rendering system based on the shear-warp factorization. *IEEE Transactions on Visualization and Computer Graphics*, 2(3):218–231, September 1996.
9. P. Lacroute and M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *Computer Graphics Proceedings*, pages 451–458, Orlando, FL, July 1994.
10. P. G. Lacroute. *Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation.* PhD thesis, Stanford University, Stanford, CA, September 1995.
11. K.-L. Ma, J.S. Painter, C.D. Hansen, and M.F. Krogh. A data distributed parallel algorithm for ray-traced volume rendering. In *Parallel Rendering Symposium*, pages 15–19, San Jose, CA, October 1993.
12. U. Neumann. Parallel volume-rendering algorithm performance on mesh-connected multicomputers. In *Parallel Rendering Symposium*, pages 97–104, San Jose, CA, October 1993.
13. H. Pfister and A. Kaufman. Cube-4 - a scalable architecture for real-time volume rendering. In *Symposium on Volume Visualization*, pages 47–54, San Francisco, CA, October 1996.
14. J. Ritter. A fast approximation to 3d euclidian distance. In A. Glassner, editor, *Graphics Gems*, pages 432,433. Academic Press, 1990.
15. K. Sano, H. Kitajima, H. Kobayashi, and T. Nakamura. Parallel processing of the shear-warp factorization with the binary-swap method on a distributed-memory multiprocessor system. In *Symposium on Parallel Rendering*, pages 87–94, Phoenix, AZ, October 1997.
16. H.-W. Shen and C. Johnson. Differential volume rendering: A fast volume visualization technique for flow animation. In *Visualization '94*, pages 180–187, Washington, D.C., October 1994.
17. R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. In *Computer Graphics Proceedings*, pages 169–177, Orlando, FL, 1998.
18. C. M. Wittenbrink and A.K. Somani. Permutation warping for data parallel volume rendering. In *Parallel Rendering Symposium*, pages 57–60, San Jose, CA, October 1993.











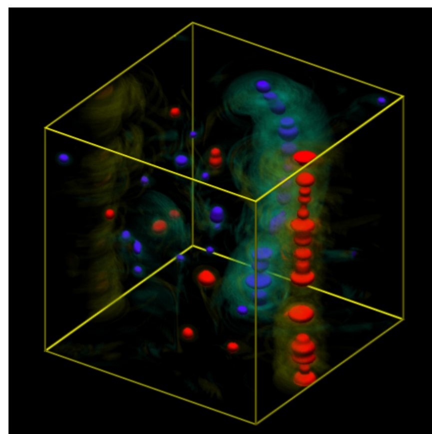


Plate 1: MRBrain (top), Polar Vortex data at time 50 (middle), and 400 (top)

Plate 2: QG data at time 100 (top), 499 (middle), and 1492 (bottom)