Interactive Texture-Based Volume Rendering for Large Data Sets

Joe Kniss
University of Utah

Patrick McCormick, Allen McPherson, James Ahrens, Jamie Painter, and Alan Keahey
Los Alamos National Laboratory

Charles Hansen
University of Utah

To employ direct volume rendering, TRex uses parallel graphics hardware, software-based compositing, and high-performance I/O to provide near-interactive display rates for time-varying, terabyte-sized data sets.

Visualization is an integral part of scientific computation and simulation. State-of-the-art simulations of physical systems can generate terabytes to petabytes of time-varying data where a single time step can contain more than a gigabyte of data per variable. As memory sizes continue to increase, the size of data sets will likely increase at a comparably high rate. The key to understanding this data is visualizing the global and local relationships of data elements.

Direct volume rendering is an excellent method for examining these properties. It lets each data element contribute to the final image and allows querying of the spatial relationship of data elements and their quantitative relationships. Hardware-accelerated volume rendering lets users achieve interactive display rates for reasonably sized data sets. The size of interactive data sets is a function of the hardware's available texture memory and fill rate. Current high-end hardware implementations place an upper bound on data-set sizes at approximately 256 Mbytes. In this article, we present a scalable, pipelined approach for rendering data sets too large for a single graphics card. To do so, we take advantage of multiple hardware rendering units and parallel software compositing. (See the "Previous Work" sidebar on p. 54 for other approaches.)

The goals of TRex, our system for interactive volume rendering of large data sets, are to provide near-interactive display rates for time-varying, terabyte-sized uniformly sampled data sets and provide a low-latency platform for volume visualization in immersive environments. We consider 5 frames per second (fps) to be near-interactive rates for normal viewing environments and immersive environments to have a lower bound frame rate of 10 fps. Although this is significantly below most virtual environment update rates, we've found that the user can successfully investigate extremely large data sets at this rate. Using TRex for virtual reality environments requires low latency—around 50 ms per frame or 100 ms per view update or stereo pair. To achieve lower latency renderings, we either render smaller portions of the volume on more graphics pipes or subsample the volume to render fewer samples per frame by each graphics pipe. Unstructured data sets must be resampled to appropriately leverage the 3D texture volume rendering method.

TRex system overview

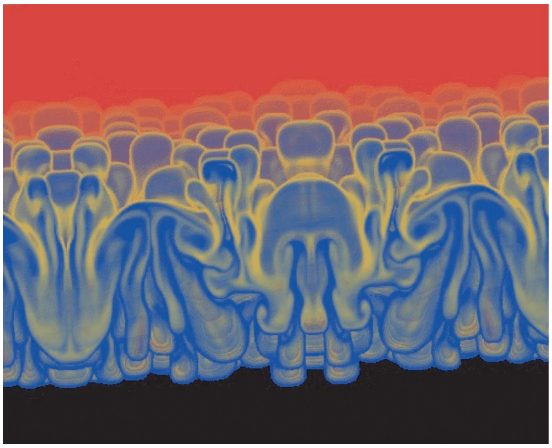
Our implementation is a hybrid parallel software and hardware volume renderer. We designed the system to render full-resolution time-varying data, such as the Raleigh–Taylor fluid flow data set in Figures 1 and 2, at nearly 5 fps on a 128-CPU, 16-pipe SGI Origin 2000 with IR-2 graphics hardware. Because we achieve more than 5 fps for a static volume, rendering isn't the bottleneck. The limiting factor for time-varying data sets is the high-performance I/O (as we describe later). For immersive environments, we achieve 10 fps using stereo pairs, albeit with half the sampling resolution. The primary difference from previous parallel hardware volume rendering work is the volume renderer's interactivity on extremely large data sets. We achieve this with a design that uses all available hardware components: streaming time-varying data sets from a carefully designed, high-performance I/O system; rendering with graphics pipes; and compositing the results with processors.

We found that interactively visualizing data sets is a critical first step in the analysis process, letting users rapidly gain a detailed understanding of their results. Because the hardware components can work independently, we can pipeline the parallel volume rendering process (see Figure 3). With overlapped stages, this pipeline lets us achieve an overall performance that closely matches an individual graphics pipe's performance.

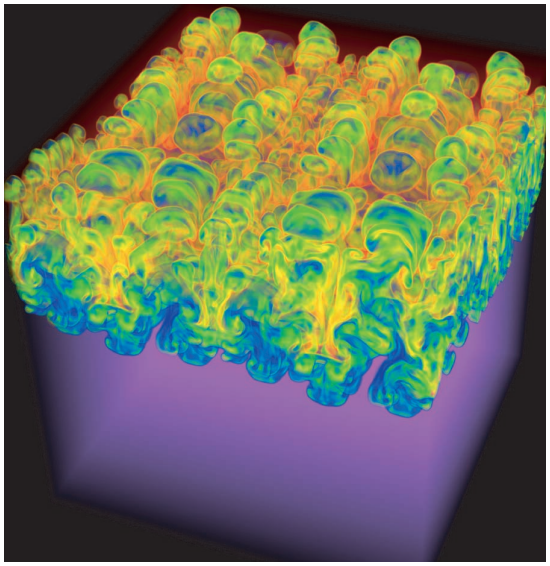
Preprocessing

Our implementation requires an offline preprocessing step in which the data is quantized from its native

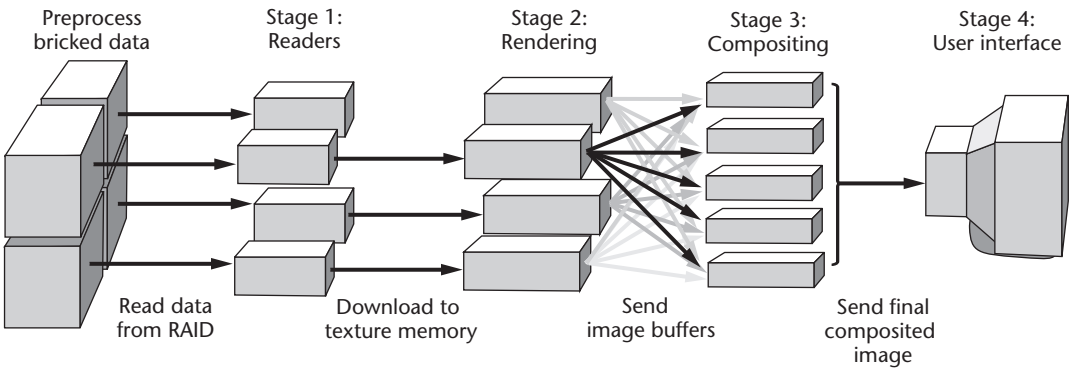
0272-1716/01/$10.00 © 2001 IEEE



1 Time-varying Raleigh–Taylor fluid instability data set (512^3).



2 Time-varying Raleigh-Taylor fluid instability data set (1024^3).



3 The TRex pipeline.

data type (commonly a floating point) to either 8-bit or 12-bit unsigned integer data. We then split the data into subvolumes with sizes matching the available texture memory on each graphics pipe. Note that most graphics-hardware implementations require that texture dimensions be a power of two. The original volume may need to be supersampled or padded to match this requirement. Users should perform these operations on the original floating-point data prior to bricking to avoid quantization errors and artifacts at subvolume interfaces. Boundary conditions in supersampling schemes can cause interface artifacts.

TRex pipeline

TRex's rendering pipeline has four stages. Each stage is a multithreaded process capable of executing simultaneously with the other stages. A stage consists of two main parts: an event manager that handles communication and a functional part that implements the thread's task(s). For a volume partitioned into N subvolumes, TRex will create N readers and N renderers. Note that it isn't necessary for the number of compositing threads to equal the number of subvolumes. The ideal number of compositor threads is a function of both image size and the number of images to be composited. A user interface thread displays the final composited image and sends user event messages to the other stages for supporting immersive TRex and direct manipulation widgets.

Stage 1: Subvolume reader. The first stage of the pipeline involves reading a time step from disk. TRex creates a separate reader thread for each of the subvolumes in a time step. Provided the data resides on a well-striped redundant array of inexpensive disks (RAID) and direct I/O is available, it can read the subvolumes from disk in parallel at approximately 140 Mbytes per second. (Direct I/O avoids kernel interrupt overhead and is a feature available on SGI systems.) Unfortunately, this data rate isn't fast enough to sustain our desired throughput of 5 fps for 1024^3 time-varying data sets, but it provides a parallel approach to I/O that will work on most SGI systems. For large time-varying data sets, we require a higher performance I/O design.

One method of achieving such high-performance I/O is to build a customized file system for streaming volume data directly into system memory and then into texture memory. On the SGI Origin 2000, it's possible to do this by first colocating the I/O controllers and graphics pipes, so that they share a common physical memory within the nonuniform memory access (NUMA) architecture. This configuration avoids the overhead associated with routing data through the sys-

Previous Work

We can group current volume rendering methods into three major categories. The first group of methods lend themselves to parallel software implementations. These include ray casting, shear warp, and splatting. Ray casting[1] is a special case of the ray-tracing method. Rays are cast from an eye point through the view plane and intersected with volume elements. Samples taken along the ray are typically trilinearly interpolated and composited in a front-to-back order.

Shear warp[2] takes advantage of precomputed coordinate axis-aligned slices through the volume and replaces the more expensive trilinear interpolation with bilinear in these slice planes. This method generates arbitrary view points by orthographically compositing the sheared slices along a major axis. This intermediate view produces a sheared version of the volume that then undergoes a 2D warping transformation in image space to generate the final image. Shear warp is considered one of the fastest software volume rendering methods, but it can suffer from artifacts. It also requires three copies of the volume to remain in memory, one copy for each major axis.

Splatting[3] is a projection-based method where each voxel is generalized into a contribution extent, typically achieved by convolving the voxel with a gaussian, which essentially eliminates the need for interpolated sample points. The convolution performs the data interpolation on a per-splat basis. This method also typically composites the splats in a front-to-back order.

The second grouping includes methods that we can implement on single graphics adapters. While at some level the hardware may take advantage of parallelism, we perform the volume rendering on a single graphics unit. Using texture mapping hardware, we can take 2D axis-aligned slices that take advantage of bilinear texture-mapping hardware.[4] These slices are alpha-blended to form the final image. Two-dimensional texture methods also suffer from the same sampling artifacts as the software shear-warp implementation and require three copies of the data sliced along the major axes. Three-dimensional texture-based methods[5] take advantage of trilinear interpolation in hardware. The voxels are mapped onto polygons aligned with the view direction using trilinear interpolation based on 3D texture-mapping hardware. Lamar introduced a method of polygon slice construction based on concentric shells[6] that better approximates the ray-casting method. The VolumePro volume-rendering board provides a hardware implementation of the shear-warp approximation to ray casting and can achieve 30 frames per second for 256^3 volumes.[7] In contrast to Lacroute's shear-warp algorithm, VolumePro does trilinear interpolation with supersampling, gradient estimation, Phong lighting, and real-time classification in hardware.

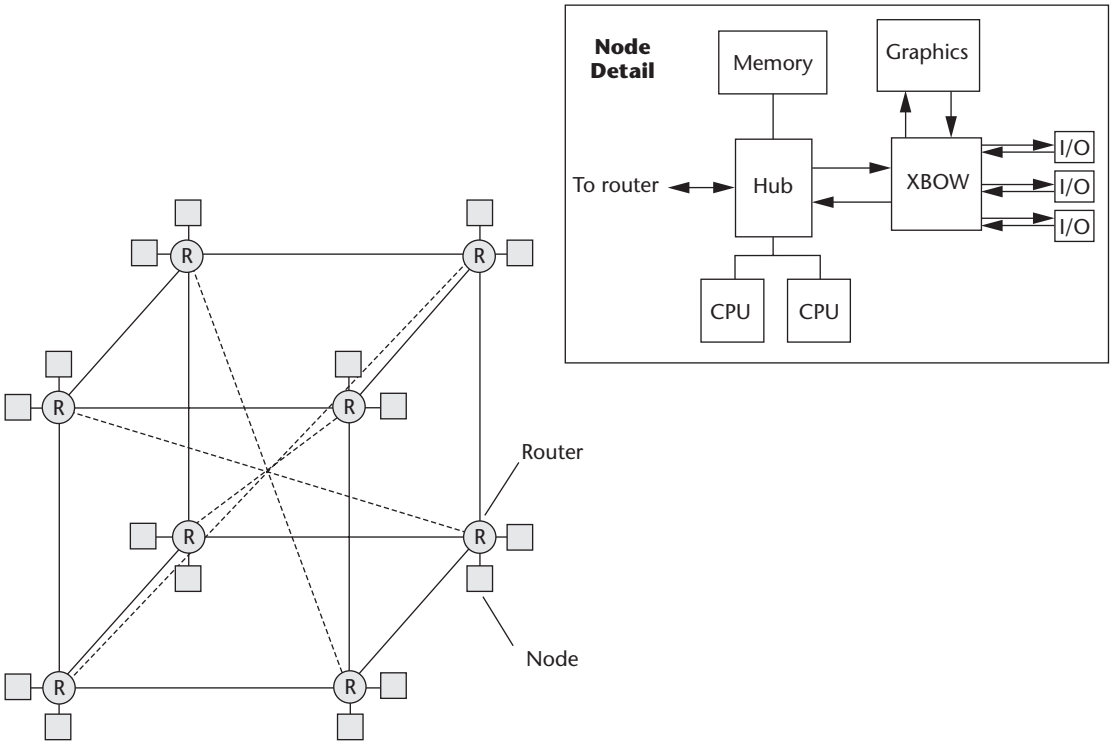
The third grouping includes methods that use hardware graphics units in parallel. Previous work in this area includes the Minnesota batch mode parallel hardware volume-renderer implementation by Paul Woodward.[8] He implemented this volume renderer for use on an Origin 2000. It wasn't designed to provide interactive visualization, rather users create key frames that the application uses to produce animations. Volumizer, a proprietary application programming interface from SGI for hardware volume rendering, also supports parallel volume rendering. The frame latency of this API, however, increases linearly as users add more graphics pipes. Both implementations composite in hardware. This requires each partial image to be downloaded and composited in the user interface's frame buffer. Downloading multiple images to graphics hardware can take considerable time, thus limiting these implementations' interactivity. Furthermore, parallel rendering using Volumizer pipelines the compositing sequentially along the different graphics pipes. This leads to limited scaling with an n frame latency, where n is the number of graphics pipes.

References

1. M. Levoy, "Display of Surfaces from Volume Data," *IEEE Computer Graphics and Applications*, vol. 8, no. 5, Sept. 1988, pp. 29-37.
2. P. Lacroute and M. Levoy, "Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transform," *ACM Computer Graphics (*Siggraph 94 Proc.), ACM Press, New York, 1994, pp. 451-458.
3. L.A. Westover, *Splatting: A Parallel, Feed-Forward Volume Rendering Algorithm*, doctoral thesis, Dept. of Computer Science, Univ. of North Carolina at Chapel Hill, Chapel Hill, N.C., 1991.
4. B. Cabral, N. Cam, and J. Foran, "Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware," *ACM Symp. Volume Visualizatio*n, A Kaufman and W. Krueger, eds., ACM Press, New York, 1994, pp. 91-98.
5. O. Wilson, A. Van Gelder, and J. Wilhelms, *Direct Volume Rendering via 3D Textures*, tech. report UCSC-CRL-94-19, Jack Baskin School of Eng., Univ. of California at Santa Cruz, Santa Cruz, Calif., 1994.
6. E. LaMar, B. Hamann, and K.I. Joy, "Multiresolution Techniques for Interactive Texture-Based Volume Visualization," *Proc. Visualization 99*, ACM Press, New York, 1999, pp. 355-361.
7. H. Pfister et al., "The VolumePro Real-Time Ray-Casting System," *ACM Computer Graphics* (Siggraph 99 Proc.), ACM Press, New York, 1999, pp. 251-260.
8. P.R. Woodward, "Interactive Scientific Visualization of Fluid Flow," *Computer*, vol. 26, no. 10, Oct. 1993, pp. 13-25.

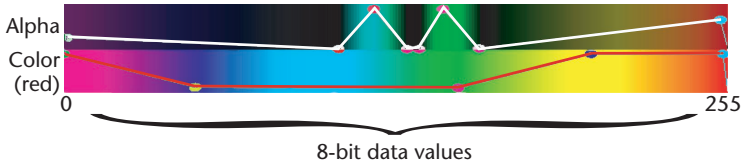
tem's interconnection network, thus minimizing data-transfer latency. To maximize performance, the I/O rates must match the approximate 300 Mbytes per second texture download rate of the infinite reality pipes. For 16 pipes operating in parallel, this is equivalent to a sustained rate of approximately 5 Gbytes per second. These rates require the use of a striped file system built using 64 dual fiber channel controllers and 2,304 individual disks. We do this by placing four fiber channel controllers next to each pipe, with each fiber channel controller capable of 70 Mbytes per second. This gives us a rate of 4×70 Mbytes per second, equaling 280 Mbytes per second for each pipe.

Assuming that we achieve the best case performance,



4 A 32-processor SGI Origin 2000 architecture.

we're limited by this 280 Mbytes per second rate of the I/O system. In addition, we also discovered that it's necessary to store subvolumes in contiguous blocks on disk to achieve these data rates and minimize the amount of traffic over the system interconnect. We can do this by using SGI's real-time file system (RTFS). Our initial benchmarks make this configuration capable of approximately 4 Gbytes per second. We're continuing our efforts to reach the desired 5-Gbytes-per-second rate.

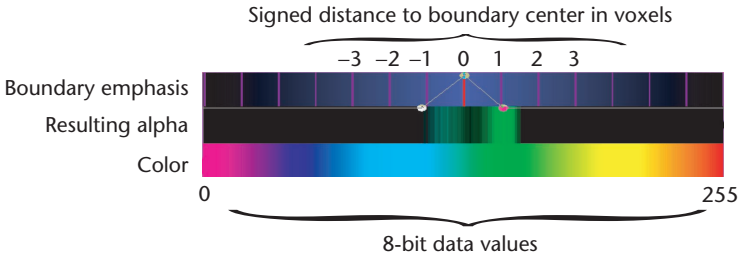


5 Texture lookup table. Note that the alpha band (top) has been multiplied by the color band (bottom) to show the resulting alpha-weighted colors.

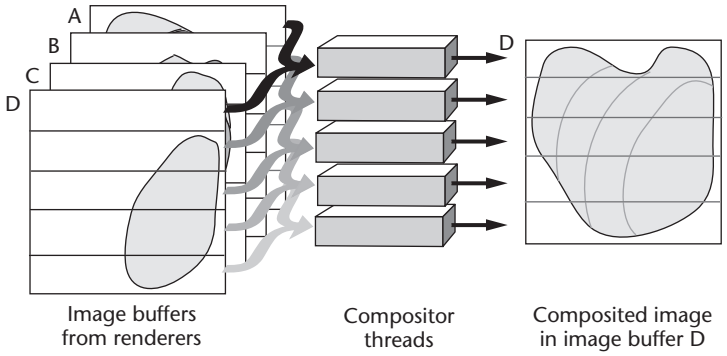
Stage 2: Rendering. The second stage of the pipeline renders subvolumes in parallel. A separate rendering thread manages each graphics pipe. These threads are initially responsible for creating OpenGL rendering windows. A renderer initializes multiple image buffers for simultaneous rendering with the other stages because the compositing and user interface stages both rely on the image buffers. Because of the Origin 2000's shared-memory architecture, we can locate allocated memory on any of the nodes within the machine. To improve performance by reducing the latency of remote data transfers, we specifically place the subvolume buffers used by the reader stages on the same node as the graphics hardware and I/O controllers. Figure 4 shows a diagram of the architecture for a 32-processor Origin 2000. The number of raw data buffers depends on the amount of time-step buffering desired plus one for the simultaneous reading of data and downloading to texture memory.

Renderers receive a render message from the user interface. This message includes information about the current frame's rotation, translation, scale, and sample rate. The rendering processes set the OpenGL model-view matrix for the frame and renders the geometry and volume data. Each renderer supports a simplistic scene graph that orders geometric primitives and subvolumes (if a pipe renders more than one). Our volume rendering approach uses 3D textures with either view-aligned slicing[1] or an approximation to Lamar's concentric shells method.[2] The final image's color and alpha values are read from the frame buffer and stored in memory. Finally, the renderer sends a message to the compositors that a new image is available, along with a pointer to the image buffer and the subvolume's distance from the eye point.

A texture lookup table encodes the transfer function, which assigns color and alpha values to the scalar texture elements. The user makes changes to the transfer function by manipulating control points in color and alpha space, as Figure 5 illustrates. We've extended the transfer function control to let the user select a boundary distance function based on Kindlmann's semiautomatic transfer function generation.[3] The user can then



6 Texture lookup table with the semiautomatic generation of alpha mappings. The top band lets the user select data values based on their distance from an ideal boundary detected in the volume. The middle band shows the generated alpha mapping multiplied by the color band.



7 Stage 3 (compositing) detail. In this example, the system composites four image buffers from Stage 2 as D over C over B over A. It then downloads image buffer D to the user interface's frame buffer in Stage 4.

select the appropriate portions of this automatic transfer function by manipulating control points in the alpha band (see Figure 6). If the transfer function or sample rate has changed since the last frame, the renderers update and redownload the transfer function to the graphics hardware prior to rendering a subvolume.

Stage 3: Compositing. Compositing begins once the compositor process(es) receive a completion message from each of the *N* renderers. The message includes a pointer to the renderer's shared-memory image buffer and the subvolume's distance from the eye point. A compositing thread composites *N* images across the final image's horizontal stripe. We determine composite order by comparing the locations of the subvolumes and compositing them back to front. Figure 7 shows that when image *A* is composited over image *B*, the resulting image resides in *A*'s buffer. This eliminates the need for additional memory in the compositing stage. The first composite thread waits for the other compositors to finish before sending a message to the user interface that a new image is ready for display.

We decided to use software compositing over hardware compositing because the compositing task is embarrassingly parallel and there wasn't a custom compositing network attached to this system. When using the existing graphics hardware for compositing, the cost of downloading *N* images is prohibitively time consuming. Also, the graphics hardware is the critical resource in this system. By using available CPUs to composite the partial results, we can achieve better scaling than a graphics hardware approach. Employing the available CPUs also lets us overlap compositing with the rendering of the next frame resulting in only a one-frame latency rather than the multiple frame latency imposed by the Volumizer and Minnesota hardware-based compositing systems.

Stage 4: User interface. The user interface thread manages the input from the user and sends messages that trigger other stages of the pipeline. If the user changes a viewing parameter such as rotation, scale, translation, or the transfer function, the user interface sends a request to the renderers along with the frame's new view parameters. When the system receives a message indicating that a new frame is available, the user interface downloads the raw image data from the shared-memory image buffer directly to the display's frame buffer.

In addition, the user has access to a quality parameter that adjusts the number of samples through the volume. The system also sets this parameter automatically. When the user is in an interaction state such as a pending rotation or translation, the system sets the sample rate lower to increase the frames per second. Once the interaction state is complete—that is, when the user releases the mouse button—the quality parameter is set higher and the volume is rendered at a higher sample rate allowing automatic progressive refinement. When the window size changes, the user interface will send a resize request to the renderers. The slave display's window size will be changed as well as the image buffers.

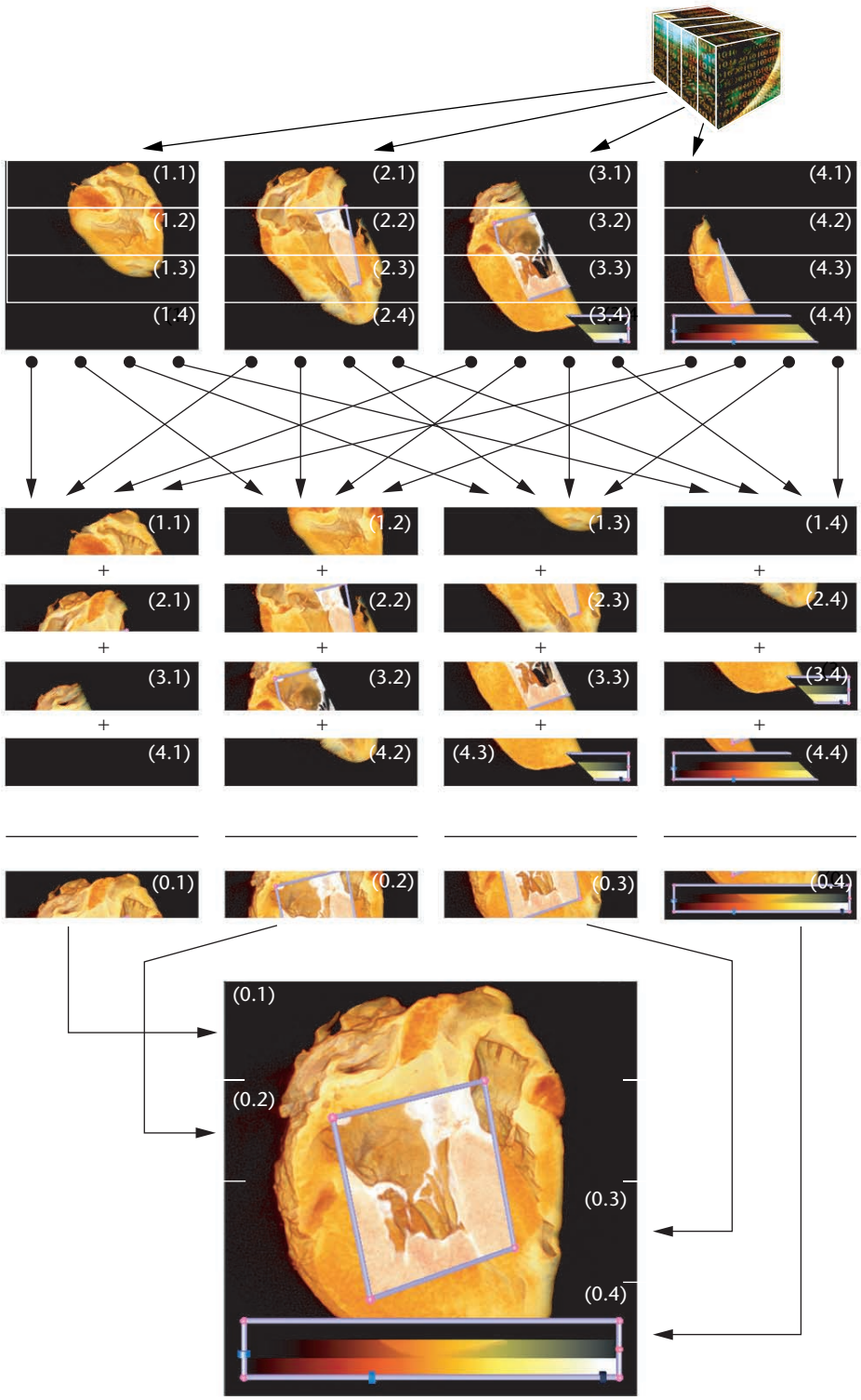
Figure 8 shows a schematic of the system. At the top, we divided the volume into subvolumes distributed to different renders. Each render generates an image of its subvolume. These are labeled subvolume and image stripe. After all subvolumes are rendered, as the middle of Figure 8 shows, the compositor threads composite the appropriate subimages. In the final step, each compositor contributes its portion to the final image, as the bottom of Figure 8 shows.

Discussion

Applications rendering transparent objects, from back to front, generate new color values by using Equations 1 and 2:[4]

$$c_{out} = \alpha_{source} \times c_{source} + (1 - \alpha_{source}) \times c_{target} \quad (1)$$

$$\alpha_{out} = \alpha_{source} + (1 - \alpha_{source}) \times \alpha_{target} \quad (2)$$



8 The entire system with renders at the top, compositors in the middle, and the final image at the bottom. The labels are subvolume and image stripe.

where α_{target} and c_{target} are the alpha and color values in the frame buffer. α_{source} and c_{source} are the incoming alpha and color values. α_{out} and c_{out} are the new alpha and color values to be written to the frame buffer.

Standard hardware implementations only let us set one function, which applies to both color and alpha channels. Because the equations for color and alpha differ, we can't simply apply color and alpha bending with Equation 1. Doing so would cause errors in the accumulated alpha value. Equations 3 and 4 demonstrate what happens when alpha compositing is treated the same as color compositing.

$$c_{out} = \alpha_{source} \times c_{source} + (1 - \alpha_{source}) \times c_{target} \quad (3)$$

$$\alpha_{out} = \alpha_{source} \times \alpha_{source} + (1 - \alpha_{source}) \times \alpha_{target} \quad (4)$$

Notice that α_{source} is squared and then added to the complemented α_{source} times α_{target}. This contrasts with Equation 2, and we can't easily correct the error.

The solution is to premultiply the color values in the texture lookup table by their corresponding alpha values. The resulting Equations 5 and 6 match Equations 1 and 2 respectively, since $c\alpha_i$ expands to $c_i * \alpha_i$:

$$c_{out} = c\alpha_{source} + (1 - \alpha_{source}) \times c_{target} \quad (5)$$

$$\alpha_{out} = \alpha_{source} + (1 - \alpha_{source}) \times \alpha_{target} \quad (6)$$

This correction is only necessary when the alpha values are used at some later time, such as compositing. For display on a single graphics pipe, the accumulated alpha value isn't important, because we only use the incoming fragment's alpha value to compute the color value—that is, we never use the alpha value in the frame buffer for computing color.

TRex lets us use a variable sampling rate by allowing an arbitrary number of slices through the volume. In this situation, it's important to properly scale the alpha values of incoming slices to maintain the overall look of the volume regardless of the sample rate. The relationship between the sample rate and scaled alpha values isn't linear. Equation 7 approximates this relationship:

$$\alpha_{new} = 1 - \left(1 - \alpha_{old}\right)^{\frac{sr_{old}}{sr_{new}}} \quad (7)$$

where sr_{old} is the sample rate used with α_{old} and sr_{new} is the new sample rate used with α_{new}.

It's also important to note that this nonlinear scaling of alpha values is only critical for alpha values near or less than 0.2. The scaling of alpha values as they approach 1 have a near-linear behavior. In practice, the computation required to scale alpha values based on Equation 7 is expensive. This, however, is compensated for because the texture lookup tables are relatively small, 256 or 4,096 elements, when compared to the volume data's size.

Using polygonal objects, such as direct manipulation widgets and isosurfaces in conjunction with volume data, requires that we take steps to ensure that the geometry is composited with the volume correctly. A scene with geometry and volume composited correctly allows geometry to appear embedded in the volume. We limit the type of geometric objects to those that are fully opaque. Geometry is rendered first with depth test and depth write. Next, the system renders the volume data from back to front with depth test only. This lets volume data be rendered over geometric data but not behind. One difficulty of compositing subvolumes with geometric data is handling polygons that reside in two or more subvolumes or only partially in a subvolume. We can solve this by clipping geometry to planes corresponding to the faces of the subvolume that border other subvolumes. This requires at most six clipping planes for a subvolume that's completely surrounded by other subvolumes.

TRex takes advantage of several platform specific optimizations. Our benchmarks on the InfiniteReality graphics subsystem revealed that unsigned shorts were best for frame-buffer reads and writes. The platform's topology is also a concern, especially in NUMA platforms like the Origin 2000. Because two CPUs share a physical block of memory and a common I/O subsystem, references to nonlocal memory or nonlocal I/O devices must traverse through one or more routing nodes of the system's interconnection network. Data transfer latency can be significantly reduced by placing processes and their memory nearest the I/O devices that they use. In this case, the I/O device is the graphics hardware. This configuration avoids the overhead associated with routing data through the system's interconnection network. We saw a significant speedup by placing the renderers and incoming data buffers near the IR pipe that they manage for the same reasons.

Immersive TRex

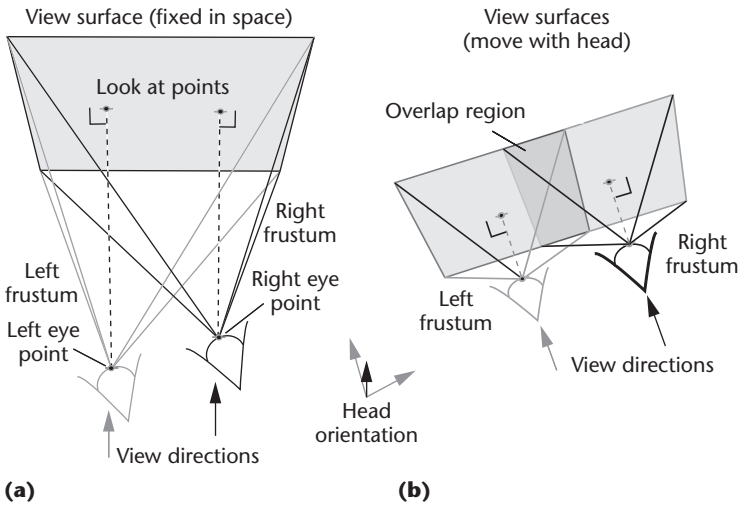
Using TRex in an immersive environment adds another level of complexity to the rendering pipeline. First, the system must generate a stereo pair for each new viewpoint. This requires twice the fill rate as a monocular viewpoint. Achieving target frame rates requires that we decrease the number of samples that each graphics pipe must process per frame by half. One solution is to increase the number of graphics pipes, rendering threads, and compositors. This requires rebricking the data set so that smaller subvolumes are rendered on each pipe. Reducing the overall size of the data set via subsampling is another option if additional graphics units aren't available. Subsampling, however, blurs fine details. The number of compositer threads in either case must increase to match the lower latency of the rendering stage.

Second, tracking devices are essential for creating an immersive environment. Typically, a separate daemon process communicates with the tracking device and reports position and orientation to a shared-memory arena. A TRex VR session will create an additional thread for monitoring and reporting head and hand device data to the user interface. Because multiple tracking devices are available, such as Polhemus Fast Track and Ascension Flock of Birds, and interaction devices may vary, the VR thread must map events from the current tracking and interaction devices to a set of events that the TRex user interface understands.

The system generates viewpoints using parallel viewing directions with asymmetrical, or sheared, frustums (see Figure 9). Head orientation in semi-immersive environments such as the Fakespace Responsive Workbench is essentially disregarded, because we can treat eyes as points and assume the portal to the virtual space (view surface) is fixed in the real space. View direction for semi-immersive environments is determined by the line from the eye point in real space that perpendicularly intersects the view surface plane. For fully immersive environments, such as ones we can achieve with the n–Visions Datavisor head-mounted display, we still treat eyes as points but require head orientation to specify the virtual portal. View direction for fully immersive environments is specified by head orientation, and we assume the eyes are looking forward.

View-aligned slicing causes artifacts when the vol-

ume is close to the user and rendered with perspective projection. Lamar's spherical shell slicing reduces this problem by assuring that the volume is rendered with differential slices that are perpendicular to the line from the center of projection to the volume element being rendered. Our approach uses an adaptive tessellation of the spherical shell. Although coarse tessellations can cause artifacts, fine tessellations can cause significant latency in the rendering. We let the user select a tessellation that's appropriate for the visualization.



9 Viewpoint construction for semi-immersive environments (a) and fully immersive environments (b) for an arbitrary head orientation. Note that the view directions are parallel and perpendicularly intersect the view surface. The overlap region in (b) is variable on many head-mounted displays.

TRex widgets

Direct manipulation widgets[5] can improve an interactive visualization's quality and productivity. Widgets also let the user have a uniform experience when using either the desktop or an immersive environment.

We created our widget sets using the Brown University widget paradigm. The widgets are object-oriented, extendable entities that maintain state similar to a discrete finite automaton. They're based on simple parts such as spheres, bars, and sliders. We constructed complex widgets from these subparts. Each subpart represents some functionality of the widget. For instance, the bars that make up a frame widget's boundary, when selected, translate the whole frame; the spheres that bracket the corners scale the frame; and the sliders attached to the bars alter some scalar value associated with the widget's functionality.

To facilitate parallel rendering, the system breaks a typical widget into $N + 1$ lightweight threads, where N is the number of subvolumes in the session. A parent thread handles events from the user interface and communicates with the child threads. The N child threads render the widget and perform subvolume-specific tasks. Each child thread is associated with a subvolume and is clipped to the half planes corresponding to the faces of the subvolume that border other subvolumes. We developed three custom widgets for TRex.

The color-map widget places the transfer function in the scene with the volume being rendered. This provides a tighter coupling between the actual data values and their representation as a color value in the image. The color-map widget consists of three bands: one for color to data-value mapping, one for opacity to data-value mapping, and one for the semiautomatic generation of opacity mappings.[3] This widget also includes sliders (see Figure 10a, next page) for manipulating the high-quality sample rate, interactive sample rate, and opacity scaling.

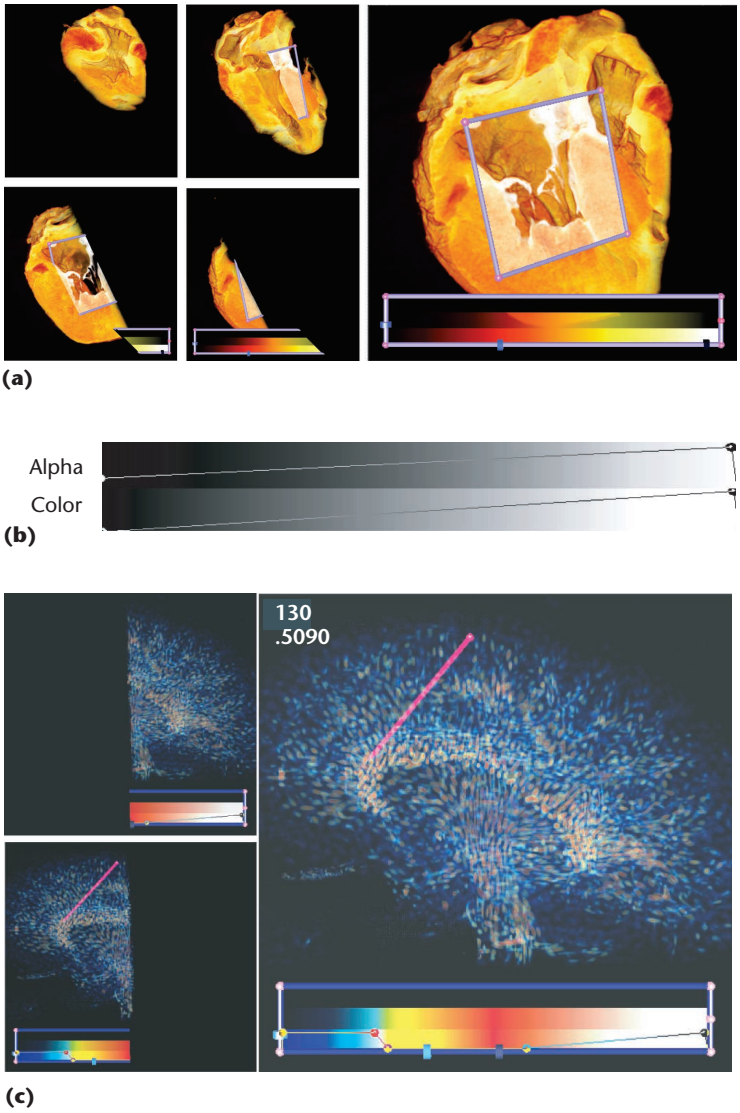
We developed a data-probe widget to let users query the visualization for local quantitative information. We can use this widget to point to a region of the volume and automatically query the original data for the values at that location. This widget is particularly useful for studying data from physical simulations where the actual value at a location is of interest. Figure 10c shows a data set employing the data-probe widget.

The internal structure of volumetric data is often obscured by the volume's other portions. One method for revealing hidden information uses clipping planes to remove the regions that occlude. For this purpose, we developed a widget that lets users position and orient an arbitrary clipping plane in the scene. Because of the amorphous quality of some volume renderings, it's necessary to map a slice to the clipping plane with a different transfer function to make the clipped boundary apparent. One useful mapping is a simple data value to gray scale and a linear alpha ramp from low to high (see Figure 10b). This sort of mapping is particularly useful for radiology data sets where users are more accustomed to viewing slices rather than the whole volume.

Conclusion and future work

Hardware volume rendering is a highly effective interactive visualization modality. Unfortunately, it imposes limits on the size of volumetric data sets that we can render with adequate update rates. We've presented a scalable solution for the near-interactive visualization of data-set time steps that are potentially an order of magnitude larger than the capabilities of a modern graphics card. Our implementation is also flexible enough to support advanced interaction tools and serves as a platform for future volume rendering and visualization research. We're currently integrating the results of our research in a production-quality application for scientists at Los Alamos National Laboratory.

With the recent performance gains in the commodity graphics card market, we're investigating PC clusters as a replacement for the Origin 2000 system. Although we've successfully managed to get TRex running on a PC cluster, this task presents several significant chal-



10 **TRex widgets. (a) Using the color-map and clip widgets, this visualization is a magnetic resonance image of a sheep heart (512^3) with a cutaway showing the atrium, ventricle, and valve. (b) Linear ramp transfer function used with the clip widget. (c) Demonstrating the data-probe widget, this is a diffusion tensor MRI of a human brain 512^3 on two graphics pipes. The left side shows intermediate renderings.**

lenges. The limitations in both the internal PC architecture and interconnection networks will make it difficult to maintain our near-interactive rendering rates. The I/O systems on PCs aren't concurrent, and syncing frames would add additional latency into the system. There are also several areas in which we must modify our current algorithms so that they operate efficiently in a distributed-memory environment, such as the compositing phase. In addition, the new functionality available in the rasterization hardware of recent commodity graphics cards offer a number of shading options at a per-pixel level. We intend to implement parallel, diffuse shaded volumes and explore alternative shading methods.

Currently, TRex only supports opaque geometry, which it must download and render on each graphics pipe. We intend to extend TRex to render both opaque and transparent geometry embedded within the volume data in a parallel, load-balanced fashion.

Finally, we're enhancing TRex's VR capabilities with intuitive interaction devices and new widgets. We'll be exploring optimizations to TRex's pipeline, such as predictive tracking that exploit the known interframe latency. Direct manipulation widgets have proved to be an indispensable tool for interactive visualization and immersive environments. We're developing a suite of widgets to perform various operations on volumetric data such as classification, segmentation, annotation, editing, and multiple-channel and vector-volume visualization. In addition to their virtual-world representation, widgets can have a physical representation associated with the interaction devices employed. To this end, we intend to develop custom interaction devices derived from common handheld and desktop devices specifically for this type of visualization. We're also considering the addition of other visualization modalities such as haptic feedback and auralization. ■

Acknowledgments

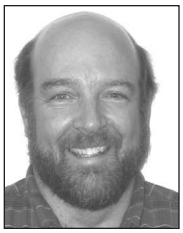
This work was supported in part by grants from the US Department of Energy ASCI Views program, the DOE Advanced Visualization Technology Center, National Science Foundation's Advanced Computational Research, and NSF's Major Research Instrumentation. We used the Raleigh–Taylor data set, courtesy of Robert Weaver at the Los Alamos National Laboratory. We acknowledge the Advanced Computing Laboratory of Los Alamos National Laboratory, where we performed portions of this work.

References

1. O. Wilson, A. Van Gelder, and J. Wilhelms, *Direct Volume Rendering via 3D Textures*, tech. report UCSC-CRL-94-19, Jack Baskin School of Eng., Univ. of California at Santa Cruz, Santa Cruz, Calif., 1994.
2. E. LaMar, B. Hamann, and K.I. Joy, "Multiresolution Techniques for Interactive Texture-Based Volume Visualization," *Proc. Visualization 99*, ACM Press, New York, 1999, pp. 355-361.
3. G. Kindlmann and J.W. Durkin, "Semi-Automatic Generation of Transfer Functions for Direct Volume Rendering," *ACM Symp. Volume Visualization*, IEEE CS, Los Alamitos, Calif., 1998, pp. 79-86.
4. T. Porter and T. Duff, "Compositing Digital Images," *ACM Computer Graphics* (Siggraph 84 Proc.), ACM Press, New York, 1984, pp. 253-259.
5. D. Brookshire Conner et al., "Three-Dimensional Widgets," *Proc. 1992 Symp. Interactive 3D Graphics*, ACM Press, New York, 1992, pp. 183-188.

Joe Kniss *is an MS student at the University of Utah, working in the Scientific Computing and Imaging Institute. His research interests are computer graphics, parallel hardware volume rendering, immersive environments, human–computer interface design, and sound localization. He has a BS in computer science from Idaho State University.*

Patrick McCormick *is a visualization researcher at the Los Alamos National Laboratory. His research interests include scientific visualization, parallel and distributed computing, and computer graphics. He has a BS and MS in computer science from the University of New Mexico.*

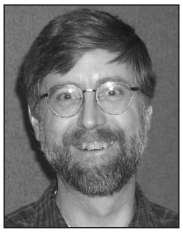


Allen McPherson *is a visualization researcher at Los Alamos National Laboratory. His research interests include hardware-accelerated rendering, visualization and rendering algorithms, and parallel computing. He holds a BS and MS in computer science from Southern Illinois University and the University of New Mexico, respectively.*

James Ahrens *is a technical staff member at the Los Alamos National Laboratory. His research interests include scientific visualization, computer graphics, parallel and distributed systems, and component architectures. He has a BS in computer science from the University of Massachusetts and an MS and PhD in computer science from the University of Washington. He is a member of the IEEE Computer Society.*

James (Jamie) Painter *is a research scientist at TurboLinux. At the time of this work, he was the project leader for scientific visualization at the Los Alamos National Laboratory Advanced Computing Laboratory. His research interests include computer graphics, scientific visualization, and cluster-based parallel and distributed computing. He has a BS in mathematics and an MS and PhD in computer science from the University of Washington.*

Alan Keahey *is a technical staff member with the Advanced Computing Group at the Los Alamos National Lab. His research interests include visual data exploration, visual navigation systems, and information visualization. He has a PhD from Indiana University, where his research was funded by a Graduate Assistance in Areas of National Need (GAANN) fellowship from the US Department of Education.*



Charles Hansen *is an associate professor of computer science at the University of Utah. His research interests include large-scale scientific visualization and parallel computer graphics algorithms. He has a BS in computer science from Memphis State University and a PhD in computer science from the University of Utah. He was a Bourse de Chateaubriand Post-Doc Fellow at the French National Institute for Research in Computer Science and Control (INRIA), Rocquencourt, France, working with Olivier Faugeras' group. He is a member of IEEE Computer Society and ACM Siggraph.*

Readers may contact Hansen at the Scientific Computing and Imaging Institute, School of Computing, 50 S. Central Campus Dr., Rm. 3490, Salt Lake City, UT 84112, email hansen@cs.utah.edu.