Distributed and Collaborative Visualization

Vinod Anupam, Chandrajit Bajaj, Daniel Schikore, and Matthew Schikore
Purdue University

A network of low-cost workstations can be harnessed to render large volume data sets efficiently and allow group interaction with the resulting images in a distributed setting.

With ongoing advances in high-speed networking and computer processor and memory technology, distributed systems provide a mechanism for effectively harnessing the total computational power of multiple workstations available on a network. Adopting a hybrid strategy that combines output distribution and task partitioning allows us to obtain the maximum benefit from distributed systems. Distributing the output of a large computational task emphasizes sharing of resources among applications. Partitioning a large computational task into independent subtasks and then distributing those subtasks accords us the benefit of parallelism. The harnessed distributed system is thus made to serve as a cost-effective high-performance virtual machine for performing large computations.

Visualization typically involves large computational tasks, often performed on supercomputers. The results of these tasks are usually analyzed by a design team consisting of several members. Our goal is to depart from traditional single-user systems and build a low-cost scientific visualization environment that enables computer-supported cooperative work in the distributed setting. A synchronously conferenced collaborative visualization environment would let multiple users on a network of workstations and supercomputers share large data sets, simultaneously view visualizations of the data, and interact with multiple views while varying parameters. Such an environment would support collaboration in both the problem-solving phase and the review phase of design tasks.

In this article we describe two distributed visualization algorithms and the facilities that enable collaborative visualization. These are all implemented on top of the distribution and collaboration mechanisms of an environment called Shastra,[1] developed by our research group and executing on a set of low-cost networked workstations.

Volume rendering

To demonstrate our distributed and collaborative visualization environment, we will use the compute-intensive task of volume visualization. Volume visualization is a highly intuitive way to interpret volumetric data.[2] Measurement-based volumetric data sets result from sampling and may consist of geological and geophysical measurements or data collected by three-dimensional scanning. In medical imaging, for example, computed tomography, magnetic resonance imaging, and laser surface imaging all produce volumetric data sets.[3] Synthetic volume data sets are generated by computer-based simulation and modeling, which may involve meteorological and thermodynamic simulations, finite-element stress analyses, computational fluid dynamics, or molecular modeling. Volume visualization provides mechanisms for expressing the information contained in these data sets as images; the challenge, of course, lies in handling these typically large data sets efficiently and in making the images easy to understand.

Volume rendering techniques can be classified as forward projection or backward projection. Forward projection techniques project the volume from object space into the screen space, similar to the way typical graphics hardware renders geometric primitives. Reverse projection techniques determine the final color of each pixel in the image by casting a ray through the viewing volume and intersecting the ray with the data volume and compositing the results. For both techniques, the cost scales with the size of the volume (N^3 for an $N \times N \times N$ data set). Volume data sets are common with $N = 128$ and are sometimes as large as $N = 512$, which clearly demonstrates the compute-intensive nature of the task.

The problem of visualizing such large volume data sets interactively has frequently been addressed by using multiprocessor parallel and vector parallel architectures.[4,5] The volume visualization system we describe operates on a network of low-cost workstations and provides several ways to view volumetric data: cross-sectional viewing, isosurface reconstruction, and direct volume rendering using ray casting. It also provides facilities for interactive control and specification of the visualization process.

Ray casting. Ray casting is a direct volume-rendering algorithm in which sight rays are cast from the viewing plane through the volume, accumulating the effects of sampled data encountered along their paths.[6-10] Certain termination criteria determine when the "tracing" stops, for example, when opaque or visible voxels are encountered, or when some accumulation threshold is achieved. The opacity accumulation is based on a classification of the material giving rise to the data values.[6,10]

The basic process is to compute the pixel color for each pixel in the final image in the manner described below. There exists a prespecified opacity value α with each voxel.

(1) Compute the world space line (ray) that maps to the pixel.
(2) Intersect this line with the volume to be rendered to compute the intersections of the line passing through the volume.
(3) Determine the intersections with voxels along the ray. This is performed quickly using an extension of the Bresenham line-drawing algorithm.
(4) Do one or the other of the following:

(a) Accumulate the values along the ray to determine the pixel color. The standard accumulation equation

$$rgb_{out} = \alpha * rgb_{in} + (1 - \alpha) * rgb_{out}$$

accumulates values from back to front. This is inefficient if highly opaque voxels near the eye eliminate the contribution of voxels farther from the eye. To allow for early ray termination, we rearrange the equation to accumulate opacity values from front to back so that rays may be terminated when the final pixel value is nearly opaque.

(b) Determine the first intersection with a threshold surface value and apply an illumination model to the surface at the intersection. An illumination model determines the light intensity, hue, and saturation on the basis of the surface's ambient and specular reflection properties.

Distributed volume rendering

There are two very good ways to parallelize volume-data rendering by ray casting. With the first method, object space distribution, we can partition the volume into subvolumes to be rendered independently by remote processors. With the second method, image space distribution, we can subdivide the final image space and assign image partitions to remote processors in a cyclic fashion until the entire image has been rendered. An examination of both methods reveals trade-offs based on data-set sizes, number of workstations, and network bandwidth.

Figure 1 shows example renderings of a $512 \times 512 \times 113$ volume of a skull and a $512 \times 512 \times 109$ volume of a head with cutaways (measured in voxels). The renderings were produced by the object space distribution algorithm.

Object space distribution. Subdivision of the volume into subvolumes for independent rendering is one method for distributing the computation of a volume rendering. Partitioning the volume is highly beneficial if there are enough workstations with sufficient total memory for the volume to be statically partitioned among the network's compute nodes. This lets us distribute the volume only once; subsequent communication consists solely of the rendered subimages and associated information. When using object space partitioning, we must choose a partitioning that allows the results from each server to be efficiently combined to form the final image.

Object space partitioning. In performing object space partitioning, we choose the common approach of subdividing the

Some commonly used volume visualization terms

Interpolant — A method for computing density values interior to the voxel by constructing a function based on the values at the corners of the voxel.

Isosurface — A surface of constant density in the sampled continuous interpolant.

Ray casting — A method for volume rendering based on ray tracing.

Ray tracing — A general rendering method that renders an image by casting infinitesimally thin rays through each pixel in the image to determine which objects in the scene contribute to the pixel's color.

Rectilinear — Describes a volume data set in which all data lies on two-dimensional cross sections that are perpendicular to the coordinate axes.

Voxel — An abbreviation for "volume element" or "volume cell." It is the three-dimensional equivalent to the pixel. A voxel is a hexahedral element with data values at each of its eight corners.

volume into groups of slices parallel to the x-y plane. In this simple technique, determining the order for merging the subimages is trivial. However, the amount of work required to composite the results from each server is greater than would be necessary through other subdivision approaches, such as octree subdivision.

Merging results. Merging results from object space distribution is similar to the problem of ray-casting the initial volume. As described above, we accumulate opacities of voxels encountered along the ray from front to back. When compositing images, we perform the same accumulation from front to back using the computed color and opacity values for a given pixel from each subvolume. The front-to-back order is easily determined from the viewing direction because the subvolumes are all parallel in object space.

Image space distribution. Subdividing a volume's image space has several advantages. First, when performing the partitioning in image space, all computation for each pixel is local to a single processor; hence, there is no need for compositing the results from various processors. Second, the amount of data required to render a portion of the image is small and can be adjusted by modifying the partitioning scheme.

Image space partitioning. The first step in image space distribution is to partition the image space into regions to be distributed to the various servers. The goals in partitioning the image space are to separate the image space into regions requiring roughly equal computational time and to balance the amount of time needed to transmit data and render image cells such that both the servers and the client remain as busy as possible. Because the cost for transmission of data and ray-cast volume rendering of a subvolume is directly related to the volume's size, we consider the cell's "weight" to be the amount of volumetric data contributing to the rendering of the cell. Several subdivision methods, both adaptive and nonadaptive, have been explored.

In each case, we divide the image space into rectangles orthogonal to the screen coordinate system (see Figure 2). The easiest static method for defining the cells is to divide the space into squares of equal size. This method is fast but suffers from being insensitive to the image size and to the amount of data within the volume corresponding to the cell. Repeated subdivision is an adaptive method that takes both of these factors into consideration. Using this method, we begin with a region that contains the entire image to be rendered and subdivide in a quadtree manner on the basis of cell weights. This way we can guarantee an upper and a lower bound on the amount of data to be sent to a server and balance the network transmission time with the server rendering time.

Input volume partitioning. To determine the portion of the data volume contained within an image cell, we consider the world space within a cell to be the intersection of four half-spaces defined by four planes bounding the volume from top, bottom, left, and right. We map the corners of the cell to eight points in world space defining these planes. Each slice in the volume is a polygon in a constant Z plane. We clip this polygon by each of the planes bounding the cell to determine the portion of each slice contained in the cell. Since we send only rectangular portions of slices to the server processes, we take the bounding box of the clipped polygon as the required data to be distributed.

Figure 1. Renderings of a 512 × 512 × 113 volume of a human skull and a 512 × 512 × 109 volume of a head with cutaway.

Figure 3 shows distributed volume-rendered images of a 512 × 512 × 920 magnetic resonance imaging data set of a human cadaver. The image on the left shows a frontal view, with bone viewed as opaque and other tissues visualized as transparent, revealing the entire skeletal structure. The remaining two images show different views, with tissues assigned translucency values in relation to their density.

Data communication optimizations. The data sent from client to server consists of rectangular portions of input data slices containing the region of the slice required to render one of the cells of the image space. In certain applications, such as medical imaging, volume data frequently contains large regions of empty space. We take advantage of this fact when data is being transmitted between workstations by run-length encoding the zeroes in the stream of input data values sent to the server processes. In the worst case, there may be numerous isolated zeroes that would be doubled by the run-length encoding, but in practice we have realized compression ratios ranging from 4:1 to 10:1 with this method.

An important factor to consider is the classic issue of data packet size versus

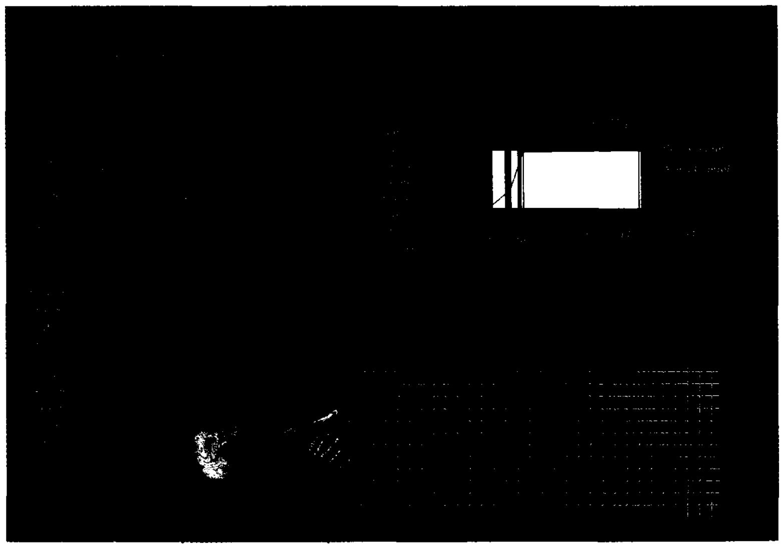


Figure 2. Image space partitioning for distributed rendering of a 512 × 512 × 920 human cadaver.

Figure 3. Distributed rendering of a 512 × 512 × 920 human cadaver with different levels of skin transparency.

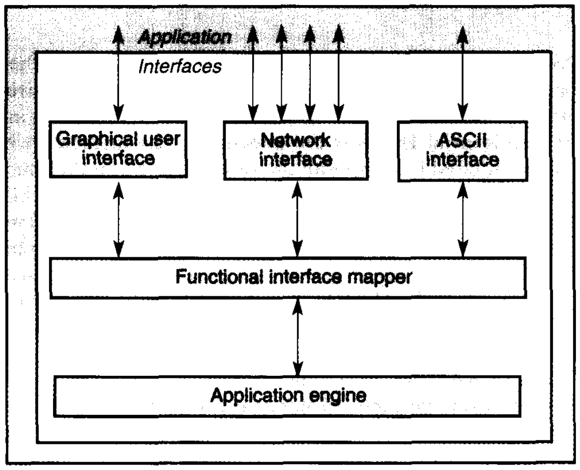


Figure 4. The architecture of applications in Shastra, a collaborative multimedia scientific manipulation environment.

number of packets. Since a large amount of data needs to be moved from the client to multiple servers, the local area network can easily get congested. However, if the data slices sent to servers are small, communication overhead dominates the total cost of distribution because of the very large number of data transfers. A balance, therefore, needs to be struck between the two. In practice, we have observed the balance to be sensitive to ambient network traffic. Our current system runs on an Ethernet (10 megabits per second) and would benefit greatly if we could use very high bandwidth network technologies such as ATM (asynchronous transfer mode).

Heterogeneity issues. In a heterogeneous computing environment, Shastra applications achieve hardware independence by building on top of high-level abstractions, above the greatest common denominator. We assume the availability of the X Window System (X11R5) for user interfaces. Multiplatform development is usually cumbersome because high-performance graphics platforms have different graphics models and application programming interfaces (APIs). The solution in Shastra is to achieve platform independence by building applications atop abstract libraries that hide hardware specifics. These abstract libraries can be easily extended to support standardized interfaces as they evolve. Platform heterogeneity problems in the realm of data representation are obviated by using the Shastra protocol for data transport, which uses XDR (external data representation) to encode data in a device-independent manner.

The XS graphics and windows library was developed to provide a machine-independent interface to routines for graphics. The use of this abstraction provides us with source-code-level portability across multiple platforms without compromising speed or graphics quality.[1] The current suite of libraries supports graphics using X11, SGI/GL, HP/Starbase, and Windows 3.1.

Shared workspaces

Shastra is a collaborative multimedia scientific manipulation environment in which experts in a cooperating group communicate and interact across a network to solve problems. The Shastra environment consists of a group of interoperating applications collectively called tools. Some tools are responsible for managing the distributed environment (the Kernel), and others are responsible for maintaining collaborative sessions (the Session Managers). Other tools provide specific communication services (the Services), while yet others provide scientific design and manipulation functionality (the Toolkits). Service applications are special-purpose tools for multimedia support; they provide mechanisms for textual, graphical, audio, and video rendition and communication.

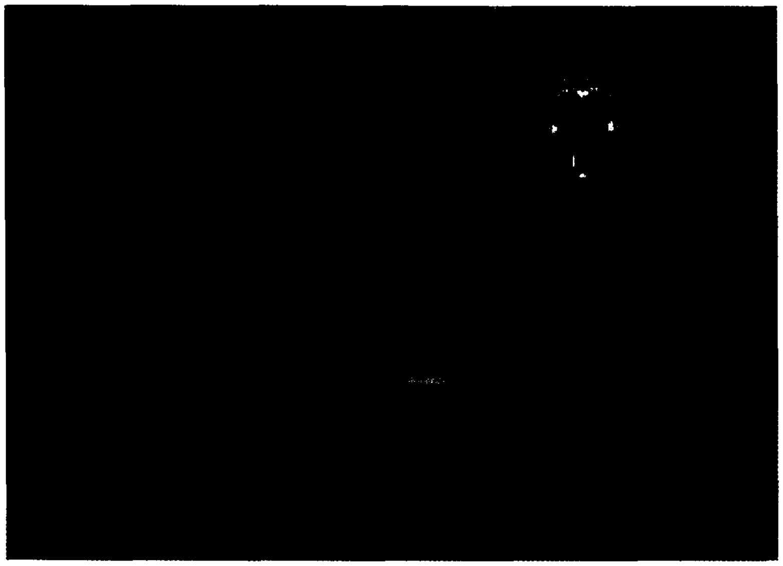


Figure 5. Using Poly, a Shastra application for collaborative visualization.

Different tools register with the environment at start-up, providing information about the kind of services they offer (directory) and how and where they are to be contacted for those services (location). The environment provides mechanisms to create remote instances of applications and connect to them in client-server or peer-peer mode (distribution). In addition, the environment supports many modes of synchronous multiuser interaction (collaboration). It offers facilities for starting and terminating collaborative sessions and for joining or leaving them. We have described the infrastructure in detail in the literature.[1]

The distribution infrastructure. All tools designed to run in the Shastra environment have certain architectural features that make them amenable to interoperation. A typical tool has an application-specific core — the application engine — that implements all the functionality offered by the toolkit or service. On top of the engine is an interface mapper that actually invokes functionality embedded in the engine in response to requests from the ASCII interface, the graphical user interface, and network interfaces. Users interact with a tool via the ASCII interface or the GUI. Intertool communication occurs via the network interfaces. Figure 4 shows a block diagram of this architecture.

The Shastra layer, comprising the network, session, and data communication substrates, joins the network interfaces of various tools. This layer provides connection setup and multiple-connection management facilities in the distributed environment. It also implements the Shastra communication protocol for peer-to-peer communication. The connected application-object interfaces of Shastra tools constitute a distributed virtual machine over which parallel algorithms are implemented and synchronous conferences are conducted.

A collaborative visualization tool. Klinker describes a telecollaborative data exploration environment[11] in which users share images and control their presentation. Gerald-Yamasaki describes an environment for cooperative visualization of computational fluid dynamics.[12] The Shastra environment goes a step further by providing facilities for media-rich interaction over the context of shared visualizations.

Poly is a rendering and visualization tool in the Shastra environment. New Shastra toolkits use Poly as their graphics interface, since it encapsulates graphical object manipulation, rendering, and visualization functionality. Poly provides a variety of mechanisms for visualizing multidimensional data. It understands numerous graphical object formats, which it converts to an internal form for efficient display and transport. At its network interfaces, Poly interoperates with other Shastra tools and provides a very high level abstraction for manipulating graphical data. A Motif-based GUI is used to manipulate visualized objects in multiple XS graphics windows.

Figure 5 shows the visualization system's user interface. The top image is a rendering of the head and upper torso of the cadaver. The skeletal structures are opaque and shaded, while the rest of the structures have been assigned different levels of transparency. The bottom image shows a surface rendering of a human head with a cutaway of the skull to show part of the brain surface.

Participants can leave a session at any time, and late arrivals can be brought up to date quickly.

The Shastra environment consists of a collection of Poly instances. A collaborative session is initiated by one of the Poly users in the environment. This user becomes the group leader and specifies to the local Kernel the list of Poly users who will be invited to participate in the session. The Kernel instantiates a Session Manager, which starts a session with the group leader as its sole participant and then invites the specified users of concurrently executing remote Poly instances to participate. Users who accept are incorporated into the session. Any Poly instance, or user, not in the conference can request admittance by using Shastra facilities to communicate with the group leader, who can admit that person to the session. A participant can leave an ongoing session at any time. Users can be invited dynamically to join and can be removed from conferences by the group leader or the leader's designees.

The hybrid computation model for conferences in Shastra consists of a centralized Session Manager for each session, which regulates the activity of multiple instances of Poly across a network. Although this model suffers from problems of scale because of the centralized Session Manager, it performs well for typical group sizes of two to 10 participants. Replication of the Poly tool provides an important benefit in the realm of platform heterogeneity: Individual application instances are responsible for dealing with idiosyncrasies of the hardware and software platform they execute on. In addition, since the conference consists of cooperating applications, the notion of private and shared workspace, as well as private and shared interaction, is naturally supported. The centralization of the Session Manager for a collaborative session accords us the benefit of a centralized state. The Session Manager serves as a repository of shared objects, making it easy for late arrivals to sessions to be brought up to date quickly. It also eases the task of serialization of input actions for multipoint synchronous interaction, and constraint management for mutual consistency.

A permissions-based regulatory subsystem regulates dataflow and control flow at runtime, providing a variety of interaction modes. Collaboration in Shastra occurs in the regulated (turn-taking) mode or in the unregulated (free interaction) mode. In the regulated mode, users take turns by passing a baton. Shastra's collaboration infrastructure has a two-tiered permissions-based regulatory subsystem used to control interaction in the unregulated mode. Shastra permissions control "access" to a view of the conference, local viewing controls to "browse" a view, rights to "modify" conference state, and rights to "copy" shared objects.

The Session Manager allows only one user to manipulate "hot spots" (places where contention is possible) in the shared space at any one time. It uses the first-come, first-served paradigm to decide which user gets temporary exclusive control. The system's baton-passing facility can be used to take turns adjusting visualization parameters. Alternatively, for arbitration or to regulate access, designers can use the auxiliary communication channels — audio, video, and text — by initiating Phone, Video, or Talk ses-

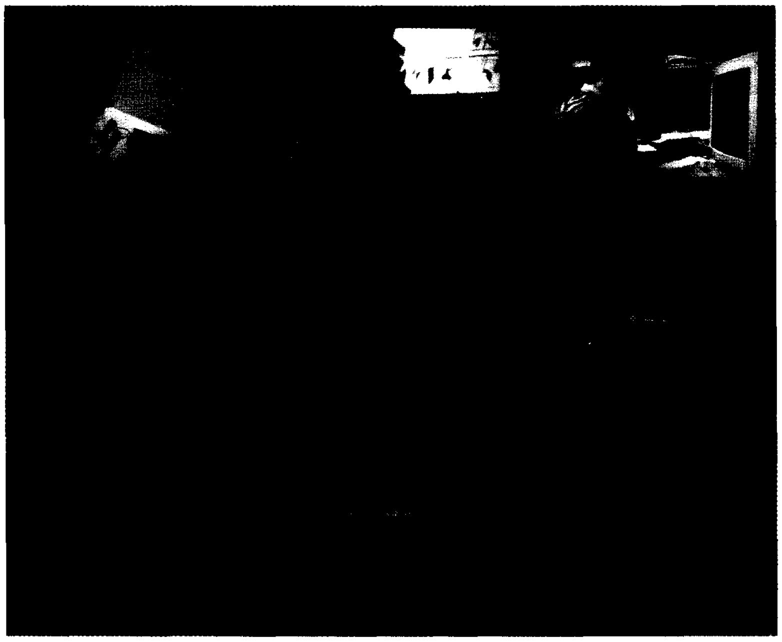


Figure 6. One site in a collaborative visualization. Collaborating users can adjust visualization modes and parameters and modify viewing modes and direction.

sions.[1] Operations are performed via the central Session Manager, which is responsible for keeping all sites up to date, so that the users have a dynamically changing and continuously updated view of the interaction in the shared windows.

Collaborative volume visualization. Every participating Poly instance creates a shared window in which all cooperative interaction occurs. Users introduce graphics objects into the session by selecting them and then placing them in the collaboration window. The Session Manager is responsible for providing access to the objects at all participating sites that have access permission and for permitting interaction relevant to the operation at sites that have modify permission for the collaboration. Collaborating users can adjust visualization modes and parameters and modify viewing modes and direction. In the shared windows, the system provides telepointers that each collaborating user can manipulate. It also indicates remote presence, revealing the viewing location of remote users in the collaborative session.

Figure 6 depicts one site in a three-way collaborative visualization. The entire rendering of the cadaver is shared by all collaborating sites; they share the data set and the viewing location as well as visualization control parameters. In the other two renderings of MRI data sets of the human head, the collaborators share data sets and viewing location but use different cutaways to examine different parts of the data.

In its simplest form, the Shastra implementation for collaborative visualization is used by just one person to perform scientific visualizations, just as in a noncollaborative setting. Additional users can be allowed to join the session with only access and browse permissions, thus setting up the environment like an electronic blackboard to teach novices the basics of the process. An appropriate setting of collaboration permissions and turn-taking allows hands-on experience. In conjunction with Shastra's audio and video communication services, this becomes a powerful instructional environment. Collaborative sessions using Poly are a valuable tool for review and analysis of problem solutions. Multimedia communication facilities permit a rapid exchange of rationales for choices, interpretations of analyses, and iterative improvement.

Table 1. A comparison of the two distribution methods. The results, given in whole seconds of real time, show how long the user must wait before the final image is available.*

Data Set	Size	No. Processors	Object Space	Image Space
Voxelized function	274 Kvoxels,	1	130	123
	287 Kbytes	2	75	112
		4	42	91
Head data	28 Mvoxels,	1	377	330
	57 Mbytes	2	168	300
		4	92	260
Skull data	30 Mvoxels,	1	296	255
	62 Mbytes	2	157	195
		4	81	165
Cadaver	240 Mvoxels,	1	n/a	2,340
	482 Mbytes	2	n/a	1,640
		4	n/a	1,390

*This time includes network latency, process swapping, and network-file-system access time.

We have used the distributed and collaborative environment to render large data sets efficiently. Our computing environment for the distributed rendering tasks consisted of an Iris Indigo R4000 and Sun 4/50 (Sparc IPX) workstations with 32 megabytes of RAM, linked by a 10-megabit-per-second Ethernet. The data sets we used are stored on remote file systems and are accessed through a network file system. Preliminary measurements of total time taken to render different volumetric data sets (during normal network traffic conditions) are very encouraging: They reveal a close-to-linear speedup using multiple workstations to render large volume data sets by object space subdivision. As Table 1 shows, object space subdivision was superior in many cases; however, it was not possible to render the very large cadaver data set using static object-space partitioning because of the sheer size of the data set.

We are fine-tuning both distributed algorithms in terms of (1) memory access patterns to minimize swapping, (2) distributed data size to reduce network traffic, and (3) congestion and load balancing to get better performance. At present, multiple distinct views of the same volume data set are rendered separately. We are adding an optimization to be made when two views differ only in the cutaways. Then we will be able to identify the parts of the final image that are common to both views and render them only once. To further improve the speed of rendering, we are exploring other partitioning and distribution strategies. Specifically, we are integrating the distributed rendering algorithm with the brokering and load-balancing facilities of Shastra to make optimal use of network computational power.

Several additional collaborative applications have been developed within the Shastra environment. The client-server, collaborative multimedia tools and services, and libraries of the Shastra environment are available to both academic and commercial software developers. Detailed information on the software is available by querying shastra@cs.purdue.edu, via Mosaic http://www.cs.purdue.edu/research/shastra/shastra.html, or telephone (317) 494-6531, or fax (317) 494-0739. ■

Acknowledgments

We are grateful to the research centers at Johns Hopkins University (Elliot Fishman and Diane Hauser) for the human cadaver data set, the State University of New York at Stony Brook (Arie Kaufman) for one of the human head data sets, and the University of North Carolina for anonymous ftp access to the skull and the other human head data set. This research was supported in part by NSF Grants CCR 92-22467 and DMS 91-01424, Air Force Office of Scientific Research Grant F49620-93-10138, NASA Grant NAG-1-1473, and a gift from AT&T.

References

1. V. Anupam and C. Bajaj, "Shastra: Multimedia Collaborative Design Environment," *IEEE Multimedia*, Vol. 1, No. 2, Summer 1994, pp. 39-49.

2. A. Kaufman, *Volume Visualization*, IEEE CS Press, Los Alamitos, Calif., Order No. 2020, 1990.

3. B. Collins, "Data Visualization," in *Directions in Geometric Computing*, R. Martin, ed., Information Geometers Press, Winchester, UK, 1993, pp. 31-80.

4. W.M. Hsu, "Segmented Ray Casting for Data Parallel Volume Rendering," *Proc. 1993 Parallel Rendering Symp.*, IEEE CS Press, Los Alamitos, Calif., Order No. 4920, 1993, pp. 7-14.

5. K-L Ma et al., "A Data-Distributed, Parallel Algorithm for Ray-Traced Volume Rendering," *Proc. 1993 Parallel Rendering Symp.*, IEEE CS Press, Los Alamitos, Calif., Order No. 4920, 1993, pp. 15-22.

6. R. Drebin, L. Carpenter, and P. Hanrahan, "Volume Rendering," *Computer Graphics* (Proc. Siggraph 88), Vol. 22, No. 4, 1988, pp. 65-74.

7. L. Harris et al., "Noninvasive Numerical Dissection and Display of Anatomic Features," *Recent and Future Developments in Medical Imaging*, Vol. 152, Soc. Photooptical Instrumentation Engineers, Bellingham, Wash., 1978, pp. 10-18.

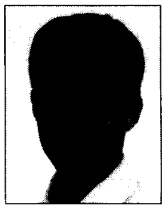
8. J. Kajiya and B. Von Herzen, "Ray Tracing Volume Densities," *Computer Graphics* (Proc. Siggraph 84), Vol. 18, No. 3, 1988, pp. 165-174.

9. M. Levoy, "Efficient Ray Tracing of Volume Data," *ACM Trans. Graphics*, Vol. 9, No. 3, ACM, New York, 1990, pp. 245-261.

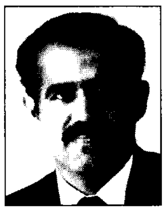
10. P. Sabella, "A Rendering Algorithm for Visualizing 3D Scalar Fields," *Computer Graphics* (Proc. Siggraph 88), Vol. 22, No. 4, 1988, pp. 51-58.

11. G.J. Klinker, "An Environment for Telecollaborative Data Exploration," *Proc. IEEE Visualization 93*, IEEE CS Press, Los Alamitos, Calif., Order No. 3940, 1993, pp. 110-117.

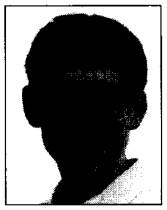
12. M. Gerald-Yamasaki, "Cooperative Visualization of Computational Fluid Dynamics," *Computer Graphics Forum*, Vol. 12, No. 3, Aug. 1993, pp. 497-508.



Vinod Anupam is a PhD candidate in the Department of Computer Sciences at Purdue University. His research interests include computer-supported cooperative work and groupware, networking and distributed systems, geometric modeling, graphics and visualization, hypermedia, and graphical user interfaces. He received a bachelor's degree in computer science from Birla Institute of Technology and Science, Pilani, India, in 1988. He is a member of Upsilon Pi Epsilon.



Chandrajit Bajaj, a professor in the Department of Computer Sciences at Purdue University, directs the Collaborative Modeling Laboratory. His research areas are computational science, geometric modeling, computer graphics, scientific visualization, and distributed and collaborative multimedia systems. He graduated from the Indian Institute of Technology, Delhi, in 1980 with a bachelor's degree in electrical engineering. He subsequently received MS and PhD degrees in computer science from Cornell University in 1983 and 1984, respectively. Bajaj is a member of ACM, the Society for Industrial and Applied Mathematics, and the IEEE Computer Society.



Daniel Schikore is a doctoral candidate in the Department of Computer Sciences at Purdue University. His research interests include scientific visualization, rendering techniques, volume visualization, and distributed systems. He received a BS in computer science and mathematics in 1992 and an MS in computer science in 1993, both from Purdue University. He is a student member of ACM, Siggraph, IEEE, and the IEEE Computer Society.

Matthew Schikore is an undergraduate student in the Department of Computer Sciences at the University of Iowa. The work described in this article began while he was participating in a National Science Foundation program at Purdue University. His research interests include volume visualization and physically based modeling. He is a student member of IEEE and the IEEE Computer Society.

The authors can be contacted at the Department of Computer Science, Purdue University, West Lafayette, IN 47907-1398; e-mail, {anupam, bajaj, drs, mcs}@cs.purdue.edu.