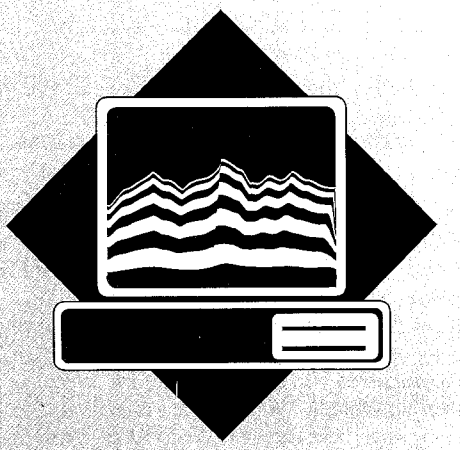

PVR: High-Performance Volume Rendering

Cláudio T. Silva and Arie E. Kaufman
State University of New York at Stony Brook

Constantine Pavlakos
Sandia National Laboratories

Traditional volume rendering methods are too slow to provide interactive visualization, especially for large 3D data sets. The PVR system implements parallel volume rendering techniques that speed up the visualization process. Moreover, it helps computational scientists, engineers, and physicians to more effectively apply volume rendering to visualization tasks.

Volume rendering is a powerful computer graphics technique for visualizing three-dimensional data.[1] While much visualization creates a rendering only of surfaces—though they may be surfaces of 3D objects—volume rendering lets us also see "inside," beneath the surface of the object being represented. This technique models a volume as cloudlike cells of semitransparent material. Each cell emits light, partially transmits light from other cells, and absorbs some incoming light (see "Volume Rendering" sidebar). For instance, while a surface rendering of the human body might show the skin, a complete volume rendering also shows the bones and internal organs, visible from any side in proper perspective.

Volume rendering began with medical visualization but has migrated to other fields, including visualization and graphics for nonscience uses. Objects of visualization need not be tangible; in fact, volume rendering is especially well suited for representing the 3D volumetric scalar and vector fields that frequently arise in computational science and engineering.

Volume rendering is a nontrivial technique and can be slow. To effectively use it in studying complex physical and abstract structures, researchers and engineers need a coherent, powerful, easy-to-use visualization tool. This tool should allow for *interactive* visualization, ideally with support for user-defined "computational steering," that is, the ability to change parameters during simulation.

But such a visualization tool presents development issues and challenges. First, even with the latest volume-rendering acceleration techniques running on top-of-the-line workstations, it still takes up to several minutes to volume-render an image—far from interactive! The large parallel computers that create the most detailed scientific simulations can generate data sets typically on the order of 32 to 512 megabytes and ranging up to 16 gigabytes. Second, even if rendering time is not a concern, large data sets may be too expensive to store and extremely slow to transfer over network links to typical workstations.

This raises the question of whether visualization should be performed directly on the parallel machine generating the simulation data, or sent to a high-performance graphics workstation for postprocessing in the traditional manner. If the visualization and simulation software were integrated, we would need no extra storage, and visual-

1070-9924/96/$5.00 © 1996 IEEE

Volume Rendering

Volume rendering accumulates information from voxels (volumetric "pixels") in a 3D data set to produce a 2D image, allowing structures in the data to be examined carefully. The technique models the volumetric data set as cloud-like material that scatters, emits, and absorbs light.[1] Several algorithms can be used. With the *ray casting* algorithm, a ray is cast in object (or volume) space for every pixel in the image. Roughly, for each ray the rendering equation

$$\int_0^x e^{-\int_0^t \sigma(s)ds} I(t)\,dt$$

is integrated, where $I(t)$ represents the light intensity emanating from a given portion of the volume and $\sigma(s)$ is the differential absorption of light (to calculate attenuation along the viewing direction). The integral is calculated by a simple numerical quadrature scheme from a set of uniform samples. $I(t)$ and $\sigma(t)$ are calculated by assigning *transfer functions*—table lookups based on the original volume data $f(x, y, z)$ computed by trilinearly interpolating the eight values defined at the volume's closest points. Each sample contains the color and opacity at a certain distance from the eye. With color and opacity known, we easily accumulate the final pixel, either back-to-front or front-to-back, in a process called *compositing*.

For instance, Figure A depicts back-to-front compositing. If the current voxel has color C, opacity α, and incoming intensity of color I, the outgoing intensity I' is given by what computer graphics people call the "over" operator, since it lays down one voxel over another:

$$I' = C + I(1-\alpha) = C \text{ over } I$$

The colors are saved premultiplied by the opacities (the actual color is C/α), which saves one multiplication per compositing operation. Compositing is associative—that is, (A over B) over C) gives the same result as (A over (B over C))—which is important for parallelization.

Transfer functions specify what portions of the volume are relevant for visualization. Like color maps, transfer functions specify color and opacity for each voxel. To locate interesting properties in data, researchers must often try different combinations of transfer functions (see Figure B) and viewing parameters. For instance, our eyes are well trained to see patterns in *moving* scenes (such as rotations). Thus, especially with complex data lacking visible hard edges, we would like to be able to animate the visualization. Unfortunately, volume rendering is typically slow, even for small data sets, especially when the volume is relatively transparent.

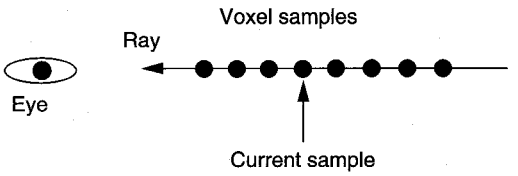

Figure A. Ray casting combines the color and intensity of voxels along each line of sight in 3D data to produce a pixel in the 2D visualization of the data. Black voxels have been composited, blue is being worked on, red voxels have not been done yet.

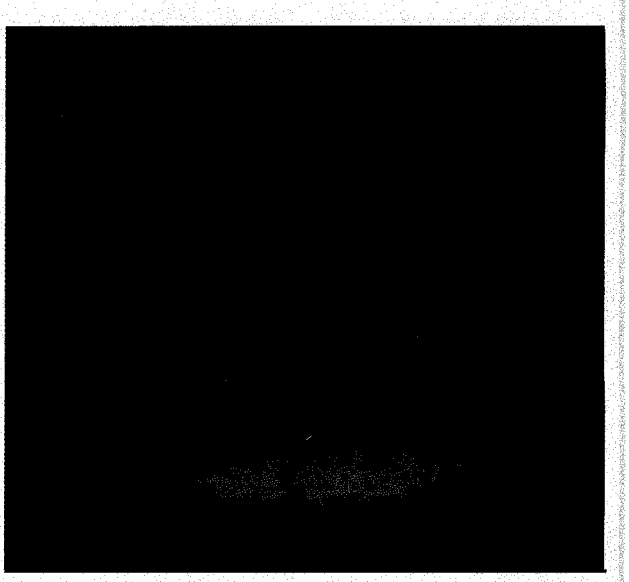

Figure B. Volume rendering of MRI data.

For example, using VolVis,[2] an advanced but nonparallel volume renderer, it takes hours to generate animations of the data sets shown in this article. With our PVR system, we can generate even the largest animations in a few seconds to a few minutes because the system scales easily.

References

1. N. Max, "Optical Models for Direct Volume Rendering," *IEEE Trans. Visualization and Computer Graphics*, Vol. 1, No. 2, June 1995, pp. 99–108.
2. R. Avila et al., "VolVis: A Diversified Volume Visualization System," *Proc. Visualization '94*, IEEE CS Press, Los Alamitos, Calif., 1994, pp. 31–38.

ization could be an active part of the simulation. Also, integrating simulation and visualization in one tool allows for the possibility of interactively "steering" the simulation. This developing methodology of computational steering lets a user observe and modify a simulation as it progresses, rather than wait for painfully long runs on expensive machines, only to discover during postprocessing that the simulations were wrong or uninteresting.

Parallel Volume Rendering

The need to render very large data sets faster, coupled with more widely available parallel and distributed machines, is the force behind parallel volume-rendering research. Good starting points in the literature are the recent survey by Tom Crockett[1] and the proceedings of the Parallel Rendering Symposia ('93, '95), the ACM Volume Visualization Symposia, and the IEEE Visualization conferences.

Parallel volume rendering can exploit three main types of parallelism:

- *Object-space parallelism*, where each rendering node gets a portion of the data set.
- *Image-space parallelism*, where different nodes compute disjoint parts of the image.
- *Time-space* or *temporal parallelism*, where different portions of the rendering pipeline are divided, pipeline fashion, among independent sets of nodes.

In addition to our group's efforts on the PVR system, other researchers have developed several different algorithms and systems based on these types of parallelism.

The Shastra project at Purdue has developed tools for distributed and collaborative visualization.[2] The system implements parallel volume visualization with a mix of image-space and object-space load balancing. These researchers report using up to four processors for computation but give few details, making it hard to evaluate the system's usability in a massively parallel environment.

John Rowlan[3] and his colleagues describe a distributed volume-rendering system implemented on the IBM SP1. The system apparently shares some characteristics with our PVR system. In particular, it runs on a massively parallel machine, provides object-space partitioning, uses separate rendering and compositing nodes, and provides a front-end graphical user interface. Unfortunately, Rowlan provides few details on the architectural design and implementation, and describes the rendering only briefly. As far as we know, their system does not provide the flexibility, portability, and performance of PVR. For instance, it does not support multiple rendering or compositing clusters.

Another similar system is Discover,[4] developed at National Cheng-Kung University, Taiwan. Researchers developed Discover, which can use remote processor pools, for custom medical imaging applications. It offers a client-server architecture, including support for Microsoft Windows.

Our group is particularly interested in ray-casting methods that run on distributed-memory machines, such as the Intel Paragon and the ASCI teraflops machine. In these machines, which limit each node's memory access to its local memory, we must divide the data set among computing nodes. This requires that we later group volume samples back together in an image.

All ray-casting parallel methods differ primarily in the way they handle this division and regrouping. Our PVR system's parallelization method is based on a combination of data set

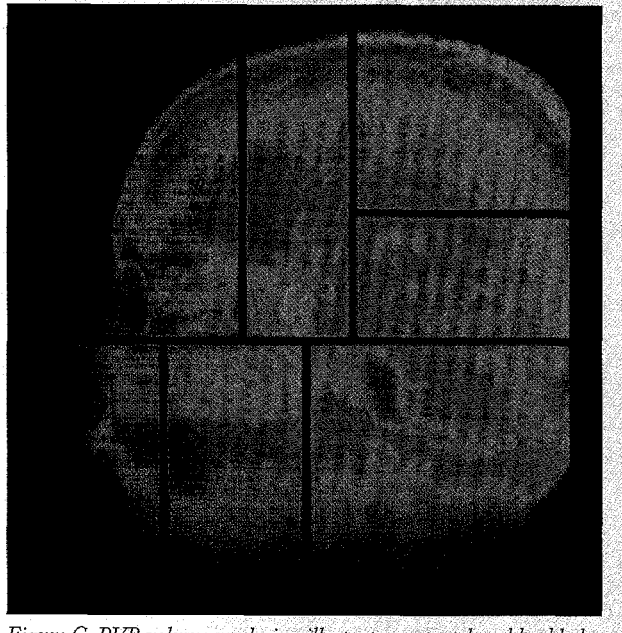

Figure C. PVR volume rendering illustrates content-based load balancing. A subdivision of the MRI head for eight processors is shown.

In this article, we describe the PVR (Parallel Volume Rendering) system that we have developed in a collaboration between the State University of New York at Stony Brook and Sandia National Laboratories. PVR is an attempt to provide an easy-to-use portable system for high-performance visualization with the speed required for interactivity and steering. The current version of PVR consists of about 25,000 lines of C and Tcl/Tk code. We've used it at Stony Brook, Sandia, and Brookhaven National Labs to visualize large data sets for over a year.

Overview of PVR

Our original goals were to achieve portability and performance for rendering beyond that of available systems and to provide a platform for further development. In a way, PVR is more than a rendering system; its components have been specially designed to enable user-defined computational steering. With PVR, it is much easier to build portable, high-performance, complex, distributed visualization environments or DVEs.

Unlike several other approaches to parallel volume rendering (see "Parallel Volume Ren-

load balancing[5] and compositing[6] schemes proposed elsewhere.

In our volume-rendering implementation, we divide the processors into two distinct groups of nodes: rendering and compositing nodes. The rendering nodes get portions of the data set; the compositing nodes are responsible for turning a collection of subray images into a complete and correct image for viewing.

In PVR, every rendering node receives part of the data set with approximately the same number of nonempty voxels, as shown in Figure C. Other approaches, such as giving the same amount of volume to each node, are also feasible.[6] Dynamic load-balancing schemes have been tried[7] but are harder to implement.

The PVR rendering nodes sample and composite their part of a ray. To avoid global communication, each subvolume region assigned to a rendering node is convex and belongs to a global BSP-tree, which makes compositing simpler. The compositing nodes regroup all subrays together consistently to keep image correctness. This calculation is only possible because composition is associative, so if we have to subray samples where one ends and the other starts, we can combine their samples into one subray recursively until we have a value that constitutes the full ray contribution to a pixel.

Ma et al.[6] approached compositing differently, switching the rendering nodes between rendering and compositing. Our method is more efficient because we can use the special structure of the subray composition to yield a high-performance pipeline, where multiple nodes implement the complete pipeline (see Figure 4 in the main text). Also, the structure of compositing requires synchronization and light-weight computation, making it much less attractive for parallelization over many processors. (In a more recent paper Ma does divide the nodes into two classes.[8])

The PVR structure lets us exploit all three types of parallelism mentioned above. By using more than one rendering cluster to compute an image, we use "object-space parallelism" and "image-space[9] parallelism" (we can specify that each cluster in PVR compute disjoint scanlines of the same image). The clustering approach coupled with the inherent pipeline parallelism available in the recursive compositing process gives rise to "time-space parallelism." In the latter, we can exploit multiple clusters by calculating subrays for several images concurrently that are sent down the compositing pipeline concurrently. We perform each composition step in lockstep to avoid mixing of images.

References

1. T.W. Crockett, "Parallel Rendering," in *Encyclopedia of Computer Science and Technology*, A. Kent and J. G. Williams, eds., Vol. 34, Supp. 19, A., Marcel Dekker, 1996, pp. 335–371. (Also available as ICASE Report No. 95-31 (NASA CR-195080), April 1995.)
2. V. Anupam et al., "Distributed and Collaborative Visualization," *Computer*, Vol. 27, No. 7, July 1994, pp. 37–43.
3. J. Rowlan et al., "A Distributed, Parallel, Interactive Volume Rendering Package," *Proc. Visualization '94*, IEEE CS Press, Los Alamitos, Calif., 1994, pp. 21–30.
4. P.-W. Liu et al., "Distributed Computing: New Power for Scientific Visualization," *IEEE Computer Graphics and Applications*, Vol. 16, No. 3, May 1996, pp. 42–51.
5. C. Silva and A. Kaufman, "Parallel Performance Measures for Volume Ray Casting," *Proc. Visualization '94*, IEEE CS Press, Los Alamitos, Calif., 1994, pp. 196–203.
6. K. Ma et al., "Parallel Volume Rendering Using Binary-Swap Compositing," *IEEE Computer Graphics and Applications*, Vol. 14, No. 4, July 1994, pp. 59–68.
7. U. Neumann, "Parallel Volume-Rendering Algorithm Performance on Mesh-Connected Multicomputers," *Proc. 1993 Parallel Rendering Symp.*, ACM Press, New York, pp. 97–104.
8. K. Ma, "Parallel Volume Rendering for Unstructured-Grid Data on Distributed Memory Machines," *Proc. IEEE/ACM Parallel Rendering Symposium '95*, ACM Press, New York, pp. 23–30.
9. J.P. Singh, A. Gupta, and M. Levoy, "Parallel Visualization Algorithms: Performance and Architectural Implications," *Computer*, Vol. 27, No. 7, July 1994, pp. 45–55.

dering" sidebar), PVR uses a component approach to building an interactive, distributed system. At its topmost level, it has a flexible and high-performance client-server architecture for volume rendering. The PVR system has the following key features:

- *Transparency*—PVR hides most of the hardware dependencies from the distributed visualization environment and the user.
- *Performance*—PVR provides high-speed pipelined ray casting with a load-balancing scheme that enables performance fine-tuning for any given machine configuration.
- *Scalability*—All system algorithms are gracefully scalable. Scalability concerns machine size as well as growth in data set and image size.
- *Extensibility*—The PVR architecture can be easily extended, making it easy for the DVE to add new functionality. Also, new functionality can be easily added to the PVR shell and its corresponding kernel to accommodate user-defined computational steering coupled with visualization.

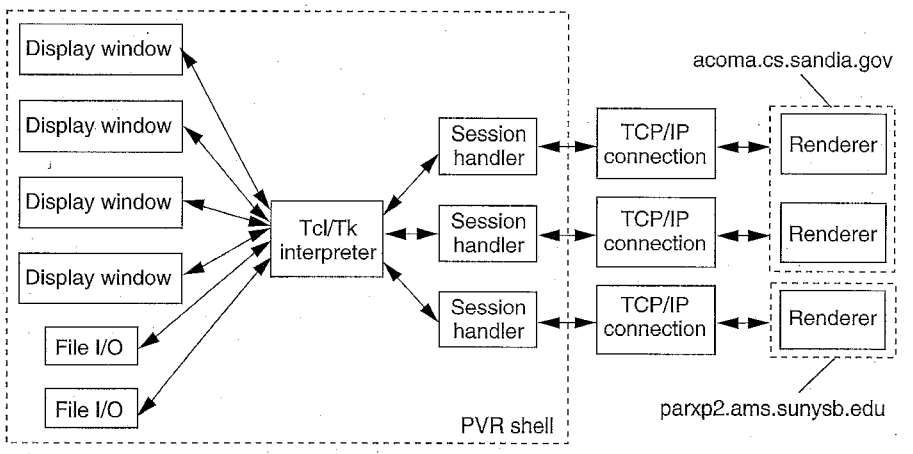

Figure 1. The PVR architecture, with an emphasis on the PVR shell. The Tcl/Tk core acts as glue for the client components. Everything except the renderers runs on the user's workstation. The renderers run remotely on parallel machines.

System complexity limits the reliability of large software systems. Distributed systems exacerbate this problem with asynchronous and nonlocal communication. PVR attempts to provide just enough functionality in the basic system, through a component approach, to allow development of large, complex visualization and steering applications. Our client-server architecture has coupled rendering/computing servers on one side and the client user workstation on the other.

We implemented the PVR client-server architecture in two main components:

- the *PVR shell* (often abbreviated *pvrsh*), which runs on the user's workstation, and
- the *PVR renderer* (*pvrren*), which runs on large parallel machines.

The PVR shell, *pvrsh*

The PVR shell, an augmented Tcl/Tk shell, gives you a single new object: the *PVR session*. Tcl/Tk, which is a well-designed, debugged script application language and powerful graphical environment, has helped reduce the system complexity.

The PVR session is an object (in the Tk sense) that contains attributes. A key attribute is the one that *binds* a session to a particular parallel machine. Figure 1 shows some of the PVR shell internal architecture and its multiple sessions capabilities. It shows three sessions, two on an Intel Paragon XP/S with over 1,840 nodes running Sunmos (Sandia–University of New Mexico Operating System), installed at Sandia, and one on an Intel Paragon with 110 nodes running an Intel version of OSF/1, installed at Stony Brook. The system uses a single protocol to handle multiple sessions on machines running different operating systems.

A session specifies the number of nodes it needs and the parameters passed to those nodes. The pvrsh and the pvrren *interactively* exchange, for example, rendering configuration information, rendering commands, image sequences, and performance and debugging information.

The flexible rendering specification means you can specify simple rendering elements, such as changing transformation matrices, transfer functions, image sizes and data sets. Moreover, with commands (see Table 1) in a high-level format, you can specify the complete parallel rendering pipeline. With these parameters, you can use the pvrsh to specify almost arbitrary scalable rendering configurations.

We implemented the PVR shell as a single process (which simplifies porting to other operating systems) in about 5,000 lines of C code. We augmented our version of the Tcl/Tk interpreter with TCP/IP connection capabilities. To support several concurrent sessions, the system performs all communication asynchronously. We use the `Tk_CreateFileHandler()` routine to arbitrate between the different sessions' input. (We could have used a Unix `select` call and polling instead, but that makes the code more complex.) Sessions work as interrupt-driven commands, responding to requests one at a time. Every session can receive events from two sources at once: the user keyboard and the remote machine. The system needs locking and disabling interrupts to ensure consistency inside critical sessions.

Our code structure lets the user augment session functionality either externally or internally. *External* augmentation occurs without recompilation, such as that performed by the user interface to show images as they are received asynchronously from the remote parallel server. *Internal* augmentation requires source code changes. The source code structure allows easy additions of functionality.

The PVR renderer, *pvrren*

The PVR renderer runs remotely on a parallel machine (see Figure 1) and has several components, the most complex being the rendering code itself. To start up multiple parallel processes at the remote machine, we use the PVR daemon, *pvrd*. On the remote machine, the handling process allocates the computing nodes and runs the renderer code on them. One PVR daemon can allocate several processes.

The renderer is the code that actually runs on the parallel nodes. The overall code structure resembles a SIMD (single instruction, multiple data) machine with high-level and low-level commands. There is one *master* node, similar to the microcontroller on a Thinking Machines CM-2, and several *slave* nodes. Slave functions depend completely on the master. The master receives commands from the PVR shell, translates them, and takes actions such as changing the slaves' states and sending them detailed instructions.

For flexibility and performance, instructions are sent to the nodes through *action tables* (similar to SIMD microcode). To ask the nodes to perform some action, the master broadcasts the address of the function to be executed. On receiving that instruction, the slaves execute the function. With this method, it's simple to add new functionality because the added functionality can be performed locally, without changing global files. Also, every function can be optimized independently, with its own communication protocol. One shortcoming of this communication method, as with SIMD machines, is that you must be careful with nonuniform execution, in particular because the Intel NX communication library (both OSF and Sunmos have support for NX) has limited functionality for handling nodes as groups. For example, in setting up barriers with NX, it's impossible to select a group from all allocated nodes. Newer communication libraries such as MPI[2] solve this shortcoming by introducing groups of nodes.

The master node divides other nodes into clusters. Each cluster has a specialized computational task; multiple clusters can cooperate in groups to perform larger tasks. Cluster configurations require only that the basic functions be specified in user-defined libraries linked in a single binary. During runtime, you can use the master to reconfigure clusters according to immediate goals and use the PVR shell to interactively send such commands. Figure 2 shows how

Table 1. Some external PVR commands. They can be typed interactively, placed in execution files, or embedded in applications.

Command	Description
:s open *M:N*	*M* is an Internet address; *N* is a port number.
:s image window *W*	*W* is a Tk photo widget.
:s image callback *F*	*F* is a procedure to be called every time a new image is received.
:s image file *F*	*F* is the name of the local file name where the video stream is saved.
:s set *Option Val*	Changes system status.
:s set -cluster *C*	Sets cluster size.
:s set -group *G*	Groups multiple clusters, to exploit image-based parallelism.
:s set -imagesz *X,Y*	Sets the desired image resolution.
:s render rotate *X,Y,Z S,E:N*	Sends a rendering request. Specifies the axis of rotation and initial, end, and incremental angles.
:s performance memory cluster	Returns the amount of data set memory in each cluster.
:s performance comp cluster latency	Estimates how long it will take to composite images in the current cluster configuration.

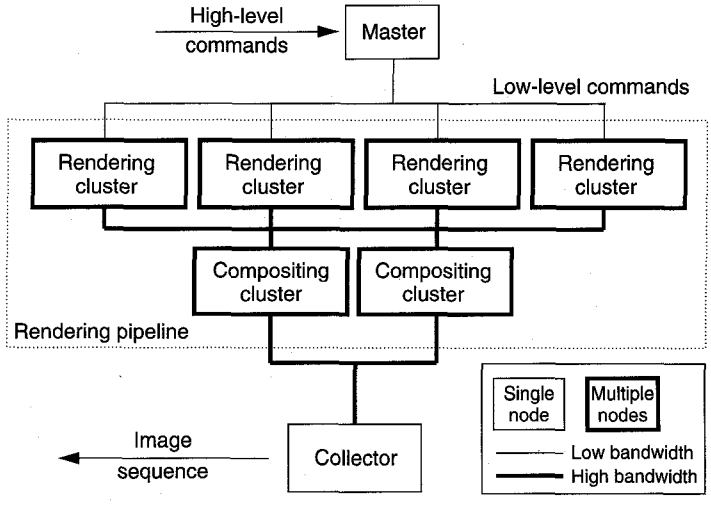

Figure 2. In PVR, the master node receives high-level commands that are converted into virtual microcode by action tables. When rendering is the task at hand, the high-level commands are for generating animations by rotations and translations. The rendering clusters work in parallel. The collector groups images together and sends an ordered image sequence to the client.

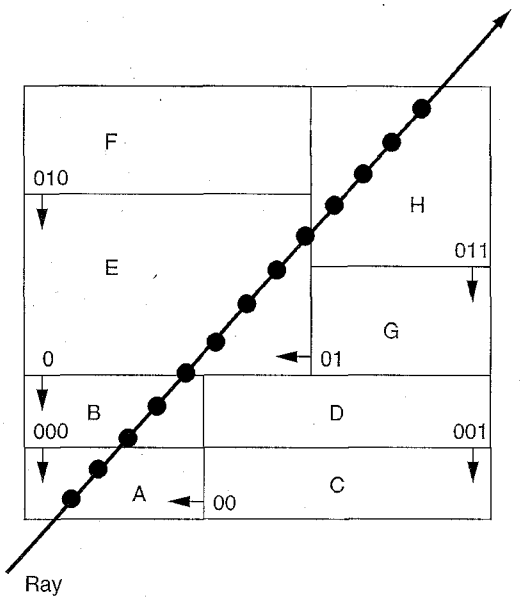

Figure 3. Data partitioning of a volume (shown in 2D cross section). The seven lines (planes in 3D) marked with binary numbers partition the volume into eight pieces A to H in a canonical hierarchical manner. A line-of-sight ray, with discrete samples shown as dots, passes through the volume. The ray's samples get composited back into a single value as shown in Figure 4.

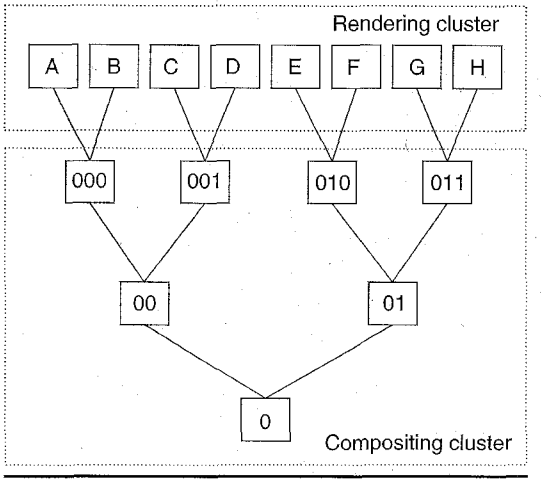

Figure 4. The internal structure of one compositing cluster, one rendering cluster, and their interconnection. In PVR, communication between the compositing and rendering clusters is flexible; since the first level of the compositing tree handles a set of tokens to guarantee consistency, several rendering clusters can work together in the same image. Because of its tree structure, one properly synchronized compositing cluster can work on several images at once, depending on its depth. The compositing cluster shown relates to the decomposition in Figure 3.

the configuration for the PVR system's high-performance volume renderer makes use of such a clustering scheme.

This clustering paradigm could help in implementing user-defined computational steering. This would usually be done by adding the functionality to the action tables (for example, linking the computational code with PVR dispatching code), and also adding extra options to the PVR shell to interactively modify the relevant parameters.

PVR volume-rendering code was the inspiration for this overall code organization and is a very good application to demonstrate its features. However, because this article focuses on describing the PVR system, not on the volume-rendering code, in the next section we only sketch the implementation.

Volume-rendering pipeline

Besides the master node, the PVR rendering pipeline is composed of *rendering nodes*, *compositing nodes*, and a *collector node* (usually just one), as in Figure 2. Optimal rendering performance and flexibility require this specialization. All the clusters work in a simple dataflow mode, where data moves from top to bottom in a pipeline fashion. Every cluster has its own fan-in and fan-out number and type of messages (see Figures 3 and 4). The master configures (and reconfigures) the overall dataflow with user-defined and automatic load-balancing parameters.

Rendering clusters reside at the top level. The clusters' nodes resample and shade a given volume data set. Generally, the input is a view matrix, and the output is a set of subimages, each related to a node in the compositing binary tree. The master can use multiple rendering clusters working on the same image (but on disjoint scanlines) to speed up rendering. Once PVR computes the subimages, they are passed down the pipeline to the compositing clusters.

The compositing clusters are organized in a binary tree structure, matching that of the compositing tree that corresponds to the decomposition of the volume on the rendering nodes. The number of processors doing compositing can differ from the number of nodes in the compositing tree, as we can apply *virtualization* to fake more processors than allocated. We pipeline images down the tree, with every iteration combining the compositing results until all the pixels

are a complete depth-ordered sequence. At the root of the compositing tree, pixels are converted to red-green-blue (RGB) format and sent to the collector node(s).

The collector node receives RGB images from the compositing nodes and compresses them with a simple, fast run-length encoding scheme. Finally, the system either sends the images to the PVR shell for user viewing (or saving), or locally caches them on the disk.

Details more completely describing our system and performance issues related to CPU speed, synchronization, and memory usage appear elsewhere.[3,4]

```
source stat.tcl                          ; External command specified in stat.tcl, it will
                                         ; place images that get to the session handler
                                         ; in the specified window, and draw a small
                                         ; performance graph
pvr_session :brain                       ; creates a session called "brain"
:brain image window .rgb.p               ; specifies the window that receives the images
:brain image callback imgCallback        ; specifies the external command
:brain open acoma.cs.sandia.gov          ; opens a connection with acoma using the
                                         ; default number of nodes (100); the defaults
                                         ; are in .pvrsh; if this command succeeds, we
                                         ; are connected
:brain set -dataset brain.slc            ; specifies the data set
:brain set -cluster r,16 -group 0,0,1,1  ; 4 rendering clusters of 16 nodes divided into
                                         ; 2 groups, nodes in a group share the same
                                         ; image calculation
:brain set -imagesz 512,512              ; specifies the image size
:brain render rotation 0,1,0 15,59:60    ; specifies the rendering of 45 images, starting
                                         ; from one quarter rotation along the y axis
```

Figure 5. A simple PVR program with a set of PVR rendering commands. This program renders images of a human brain. The commands can be put in a file and executed in batch, or can be typed interactively on the keyboard (or mixed). You can write Tcl/Tk code (for example, "stat.tcl") to take care of portions of the actions.

Rendering with PVR

Figure 5 shows a simple PVR program, which demonstrates the seamless integration with Tcl/Tk, the flexible load-balancing scheme, and the interactive specification of parameters. The `set` command can have several options (in Figure 5, options are usually specified in multiple lines but could be specified in a single line). For instance, `-imagesz` specifies the size of images output by the system.

A *cluster* of multiple nodes and a *group* of clusters are the two basic components of the PVR system's load-balancing scheme. Together they specify flexible configurations of image-space, object-space, and time-space parallelism. The master node assigns different image scanlines to rendering clusters, and assigns each group of clusters a complete image. The `-cluster` and `-group` options are used to specify this unique capability of the PVR system's load-balancing scheme. With both options, you can specify the relative sizes of the rendering and compositing clusters together with the image calculation allocation.

Several scalability strategies are possible. A rendering cluster must be large enough to hold the entire data set and at least a copy of the image. By increasing the cluster size (its number of nodes), the memory needed per node decreases. By grouping clusters (splitting the image computation across multiple clusters), the number of scanlines per given cluster decreases, lowering both the image memory requirements and the computational cost, thus speeding up image calculation.

You can use the same commands to configure compositing clusters. These don't scale at the same rate as rendering clusters, because compositing is a relatively light-computation, high-synchronization operation, unlike rendering. Compositing nodes need memory to hold two copies of the images, which can be quite large (our parallel machine nodes only have between 16 and 32 Mbytes of RAM). The compositing latency increases as the number of nodes increases (the actual rate of increase depends on the height of the compositing tree).

How PVR can be used

PVR is a flexible system that can be used for visualization in many ways. For example, the PVR system architecture facilitates the visualization of time-varying data, such as the time-step volumes computed during a computational fluid dynamics simulation. When rendering time-varying data, we add a permanent *caching cluster* to the pipeline in Figure 2 that efficiently distributes the volume data to the rendering nodes. We use the caching nodes only as *smart* memory; they hide I/O latency from disk (or other sources) and are used as buffer nodes to optimize the computation during our content-based load-balancing data distribution. You can thus visualize a data set for as long as it takes to receive updated data. Handling data that changes too rapidly (that is, faster than

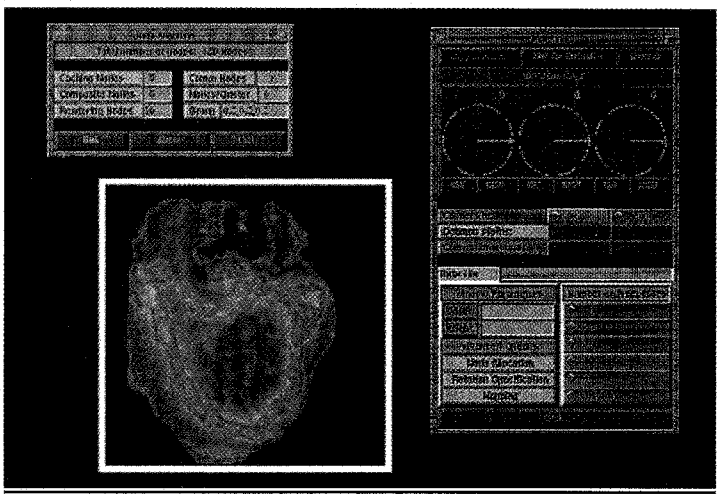

Figure 6. The simple PVR GUI. The user specifies general rotations in the main window (right). Clusters are configured at upper left. At lower left is a volume rendering of a 100 × 110 × 92 data set showing T-cell receptor density on the surface of a T-cell/B-cell interface. This lets biologists clearly check that chemical interactions are actually happening. The data sets were generated by immunofluorescence microscopy, and prepared for visualization by deconvolution on Sandia's Intel Paragon. Volume-rendering animations were generated at multiple frames per second using PVR.[6]

we can move and render it) is impossible because it would require excessive buffering.

Another use for our parallel renderer is as a visualization server for large computational parallel jobs.[5] For this, you would preallocate nodes that can be shared somewhat by multiple users for visualizing their data. To implement this server effectively, you'd also need a caching cluster, as described above. The cluster, in this case, would cache alternate user data sets.

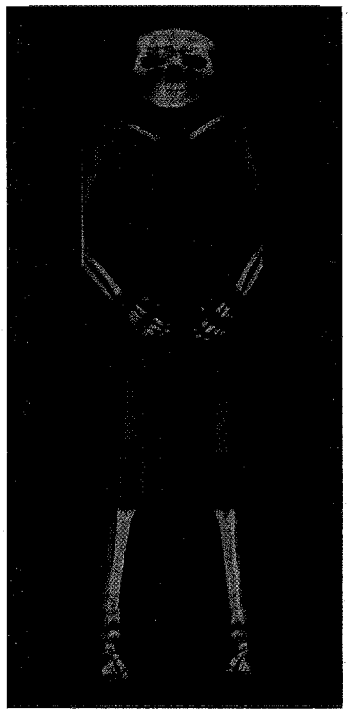

Figure 7. PVR volume rendering of the 512 × 512 × 1,877–voxel Visible Human data set.

PVR can be used to develop distributed visualization environments by means of the client-server metaphor. A DVE developed with Tcl/Tk is very portable, as Tcl/Tk versions exist for almost all of the operating systems available, and TCP/IP, which underlies our communication PVR protocol, is virtually universal. Table 1 listed more details on some of the primitives from which DVEs can be built.

Figure 6 shows a simple prototype GUI, written in Tcl/Tk, developed at Sandia. Necessary rendering parameters (such as image size and transfer function) are specified in the right window, and the load-balancing parameters in the left window. This simple interface uses only a single session, but we will be adding more functionality. With the prototype GUI written in a well-documented interface language, users can straightforwardly add functionality to the PVR GUI as needed.

PVR performance results

PVR has let us visualize numerous scientific data sets, giving us useful performance information. Our biggest challenge thus far is the limited memory on our Intel Paragon nodes. It's difficult, from the software engineering point of view, to consistently and reliably allocate memory, especially for visualization of very large data sets.

Visible Human

At the Supercomputing '95 conference in San Diego, we demonstrated PVR's ability to volume-render a 500-Mbyte data set, the 512 × 512 × 1,877–voxel Visible Human from the National Institutes of Health (see Figure 7). (This is only a subset of the full Visible Human data set.) We did this with approximately 128 rendering nodes and 127 compositing nodes of the Intel Paragon at Sandia, remotely displaying in San Diego. Rendering a 512 × 512 image takes about 5 seconds per frame. The main bottleneck is reading the 500 Mbytes of data from the Paragon disks, which currently takes around 15 minutes.

Figure 8 shows rendering times for each frame of a 72-frame animation sequence of the Visible Human data set. This is a full 360-degree rotation along the y-axis. The times are wall-clock times calculated at the collector node as it receives the images and saves them to a local disk. Each image is 400 × 400, with three color channels. For rendering, PVR represents the images as an array of pixels, each represented as four floating-point numbers (amounting to 16 bytes per pixel). At 400 × 400, each image is over 2.5 Mbytes.

The system transmits images from the rendering nodes to the compositing nodes, until they reach the root node of the compositing tree. There, we convert images to RGB format, with one byte per color channel, and transmit them to the collector node. The collector saves the final images (each 480,000 bytes) to disk. Computing the complete animation takes 129.23 seconds, or

1.79 seconds per frame, resulting in 32 Mbytes of data being saved to disk.

The noticeable peaks in the image generation time deserve further study. We believe the source of the pipeline stalls is load imbalance and also contention in writing the images to disk (the collector node stalls the pipeline whenever an image is received before the previous image is saved). The first image takes considerable longer than the others; this is the pipeline initialization cost.

Our next step is to extend the system to render the *full* RGB Visible Human data set (14 Gbytes) with high temporal resolution; that is, many frames for the rotational animation. (A 72-frame rotation uses 5-degree increments. Smaller increments are highly desirable, but a 0.5-degree increment would expand the animation files to more than 300 Mbytes.) This project would require the use of parallel I/O, a capability that we currently lack, and dedicated use of a very large parallel machine, such as the entire 1,840-node Intel Paragon at Sandia.

Scaling experiments

To show PVR's scalability, we used a 256 × 256 × 937 version of the Visible Human data set. Table 2 shows the rendering times for five different configurations, varying the number of rendering and compositing nodes. While rendering scales reasonably well, a comparison of rows 2 and 4 and rows 3 and 5 in the table indicates that it is apparently not cost-effective to increase the size of the compositing cluster for relatively small images.[3]

PVR introduces a new level of interactivity to high-performance visualization. Larger distributed visualization environments can be built on top of PVR and yet be portable across several architectures. These DVEs that use PVR have the opportunity to effectively use available processing power (up to a few hundred processors), giving a range of cost/performance to end users. PVR is a strong foundation for building cost-effective DVEs.

PVR also introduces a simple way to create user interfaces. No longer must users spend time coding in X/Motif (or Windows) to create the desired user interface. The Tcl/Tk combination is much simpler, gives more flexibility, and is nearly as powerful as the other alternatives.

Even though we have completed a usable, efficient system, much work remains. We are, for example, making the system stable enough for general distribution, and we are creating a more complete DVE (using the VolVis system[7] as a model) on top of PVR.

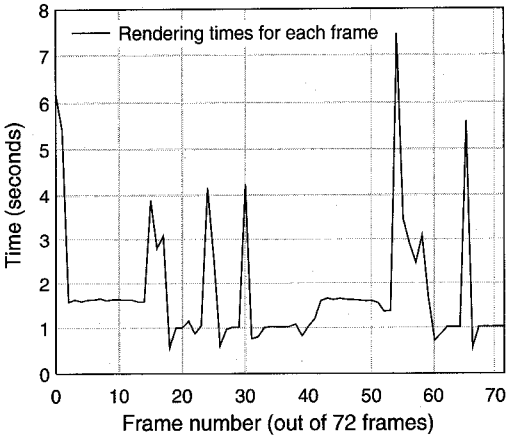

Figure 8. PVR rendering times for a 72-frame animation sequence of the 512 × 512 × 1,877 Visible Human data set. Each image is 400 × 400.

Functionality now missing from PVR must be incorporated. The most important element is probably the support for multiple data sets in a session. Implementing this capability may complicate the load-balancing scheme, and simple heuristics might not generate well-balanced decomposition schemes. If the volumes were allowed to overlap (as in VolVis), the problem would be even harder, and the solution would require heavier processing on the compositing end. It might be necessary to have a reconfiguration phase whenever a new volume is introduced, although how to do so efficiently is unclear.

Research is ongoing to incorporate irregular grid rendering in PVR. Moreover, we are considering adding a recent algorithm[8] that exploits a high level of locality, which should ultimately lead to more efficient communication schemes. Finally, we are porting PVR to use MPI as the communication layer, instead of NX. ♦

Table 2. Scalability of PVR rendering times on a 72-frame animation sequence of a 256 × 256 × 937 version of the Visible Human. Images were 250 × 250.

Rendering nodes (and clusters)	Compositing nodes	Total rendering time (s)	Mean time per frame (s)
16 (1 cluster)	15	104.10	1.44
32 (2 clusters of 16)	15	67.24	0.93
64 (4 clusters of 16)	15	56.73	0.78
32 (1 cluster)	31	71.42	0.99
64 (1 cluster)	63	58.79	0.81

For More Information

More PVR-related information, including publications, images, and several animations of the data described in this article, can be found at

- http://www.cs.sunysb.edu/~vislab
- http://cg.ams.sunysb.edu/~pvr
- http://www.cs.sandia.gov/VIS

PVR source code (Intel Paragon version) is available to users willing to provide feedback to our beta testing program. (Contact csilva@ams.sunysb.edu.)

Acknowledgments

We thank Maurice Fan Lok for co-writing the first version of PVR, Brian Wylie for project support and user interface development, and Dirk Bartz, Tzi-cker Chiueh, Pat Crossno, Steve Dawson, Juliana Freire, Tong Lee, Ron Peierls, and Amitabh Varshney for helpful discussions. For the PVR-to-Sunmos port assistance, thanks to Kevin McCurley, Rolf Riesen, Lance Shuler from Sandia, and Edward J. Barragy from Intel. Image data is courtesy of the following: MRI head, Siemens; Visible Human, the National Institutes of Health; cell, Colin Monks from the National Jewish Center for Immunology and Respiratory Medicine and George Davidson from Sandia. C. Silva is partially supported by CNPq-Brazil under a PhD fellowship, Sandia National Labs, and the Dept. of Energy Mathematics, Information, and Computer Science Office, and by the National Science Foundation, grant CDA-9626370. A. Kaufman is partially supported by the NSF under grants CCR-9205047, DCA 9303181, MIP-9527694 and by the Dept. of Energy under the PICS grant.

References

1. A.E. Kaufman, ed., *Volume Visualization*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1991.
2. M. Snir et al., *MPI: The Complete Reference*, MIT Press, Cambridge, Mass., 1995.
3. C. Silva, *Parallel Volume Rendering of Irregular Grids*, doctoral dissertation, State University of New York at Stony Brook, Dept. of Computer Sci., 1996.
4. C. Silva, A.E. Kaufman, and C. Pavlakos, *The PVR System*, Tech. Report TR96.06.10, Dept. of Computer Sci., State University of New York at Stony Brook, 1996.
5. C. Pavlakos, L. Schoof, and J. Mareda, "A Visualization Model for Supercomputing Environments," *IEEE Parallel & Distributed Technology*, Vol. 1, No. 4, Nov. 1993, pp. 16–22.
6. C. Monks et al., "Three Dimensional Visualization of Proteins in Cellular Interactions," *Proc. Visualization '96*, ACM Press, New York, pp. 363–366.
7. R. Avila et al., "VolVis: A Diversified Volume Visualization System," *Proc. Visualization '94*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1994, pp. 31–38.
8. C. Silva, J.S.B. Mitchell, and A.E. Kaufman, "Fast Rendering of Irregular Grids," in *1996 Symp. Volume Visualization*, ACM Press, New York, pp. 15–22.

Cláudio T. Silva *is a research associate in the Department of Applied Mathematics and Statistics, State University of New York at Stony Brook. His research interests are in computer graphics, scientific visualization, and high-performance computing. He received a BS in mathematics from the Federal University of Ceará, Brazil, and an MS and PhD in computer science from SUNY at Stony Brook. He is a member of ACM, IEEE Computer Society, and the Society for Industrial and Applied Mathematics.*

Arie E. Kaufman *is director of the Center for Visual Computing and Leading Professor of computer science and radiology at the State University of New York at Stony Brook. He received a BS in mathematics and physics from the Hebrew University of Jerusalem, an MS in computer science from the Weizmann Institute of Science, Rehovot, and a PhD in computer science from the Ben-Gurion University, Israel. He is editor-in-chief of* IEEE Transactions on Visualization and Computer Graphics *and is a director of the IEEE Computer Society Technical Committee on Computer Graphics. Kaufman received a 1995 IEEE Outstanding Contribution Award and a 1996 IEEE Computer Society Golden Core Member recognition.*

Constantine Pavlakos *is a senior member of the technical staff at Sandia National Laboratories. His interests include distributed visualization architectures for high-performance computing environments, volume visualization, parallel visualization algorithms, visualization of very large data sets, hierarchical data techniques, virtual reality, and multimedia. He received an MS in computer science and a BS in mathematics, both from the University of New Mexico. He is a member of ACM.*

Corresponding author: Cláudio T. Silva, Dept. of Applied Math and Statistics, State Univ. of New York at Stony Brook, Stony Brook, NY 11794; e-mail, csilva@ams.sunysb.edu; WWW, http://cg.ams.sunysb.edu/~csilva.