The PVR System

Cláudio T. Silva* Arie E. Kaufman* Constantine Pavlakos‡

*State University of New York at Stony Brook ‡Sandia National Laboratories

Abstract

PVR (Parallel Volume Rendering) is an object-oriented, client/server system, developed for high performance volume rendering of very large datasets. Among its important features are performance, scalability, transparency and extensibility. PVR is well suited for use in a supercomputing environment, where datasets are too large to be easily archived and visualized, and where computational steering capabilities are desired. For the scientist, PVR offers transparency from machine architecture details while achieving high performance visualization. For the tool builder it provides an easily extensible system architecture.

This paper introduces the system architecture of PVR, its implementation details, and some performance results. Our goal is to lead potential users in the computational science community into using direct volume rendering techniques, that before might have been too slow to be of practical value. With this goal in mind, we provide a short description of volume rendering and its parallelization. A secondary goal is to present the design of PVR to tool builders of such systems.

1 Introduction

Volume rendering [3] is a powerful computer graphics technique for the visualization of large quantities of 3D data. It is especially well suited for the visualization of 3D volumetric scalar and vector fields. Fundamentally, it works by modeling the volume as cloud-like cells composed of semi-transparent material. Each cell emits light, partially transmits light from other cells, and absorbs some incoming light (see *Volume Rendering* sidebar).

In order to allow researchers and engineers to make effective use of volume rendering in the study of complex physical and abstract structures, a coherent, powerful, easy-to-use visualization tool is needed. Furthermore, such a tool should allow for *interactive* visualization, ideally with support for user-defined "computational steering."

There are several issues and challenges in developing such a visualization tool. First, even with the latest volume-rendering acceleration techniques running on top-of-the-line workstations, it still takes a few seconds to a few minutes to volume render an image. This is clearly far from interactive. With the advent of larger parallel machines and better acquisition devices and instrumentation, larger and larger datasets are being generated (typically on the order of 32MB to 512MB, ranging to 16GB), some of which would not fit in memory of a workstation class machine. Second, even if rendering time is not a major concern, large datasets may be too expensive to hold in storage, and extremely slow to transfer to typical workstations over network links.

These issues lead to the question of whether the visualization should be performed directly on the parallel machine which is used to generate the simulation data or sent over to a high performance graphics workstation for post-processing. First, if the visualization software was integrated directly with the simulation software, there would be no need for extra storage, and visualization could be an active part of the simulation. Second, large parallel machines can render these large datasets faster than workstations can, possibly in real-time, or at least achieving interactive frame-rates (see the *Parallel Volume Rendering* sidebar). Finally, the integration of simulation and visualization in one tool, whenever possible, is highly desirable because it allows users to interactively "steer" the simulation. With steering, users are able to terminate or modify parameters in their simulations as the simulations progress, rather than have to wait for painfully long simulations on extremely expensive machines, with high storage and transmission costs, only to discover during post-processing that the simulations are wrong or uninteresting.

The Shastra project at Purdue has developed tools for distributed and collaborative visualization [1]. The system implements parallel volume visualization with a mix of image-space and object-space load balancing. Few details of the scheme are given, and they report using up to four processors for computation, which makes it hard to evaluate the systems usability in a massively parallel environment. Rowlan et al. [8] describe a distributed volume-rendering system implemented on the IBM SP-1. Their system seems to have several of the same characteristics as PVR. In particular, it runs on a massively parallel machine, provides object-space partitioning, uses separate rendering and compositing nodes and provides a front-end GUI. Unfortunately, their paper provides few details on the actual architectural design and implementation, and even the rendering is described very briefly. As far as we can detect, their system does not provide the flexibility, portability and performance that our system does. For instance, it does not provide support for multiple rendering or compositing clusters. Another similar system is DISCOVER [4], developed at National Cheng-Kung University (Taiwan). This system has been developed for custom medical

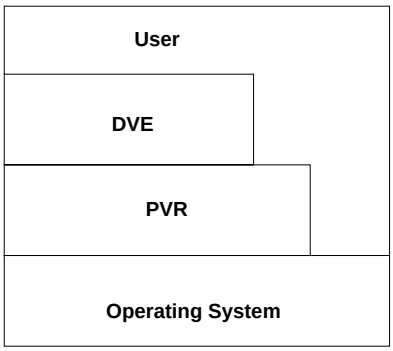


Figure 1: *The Relationship of a Distributed Visualization Environment (DVE) System and PVR.*

imaging applications and provides mechanisms for the use of remote processor pools. It provides a client/server architecture for a variety of clients, including support for Microsoft Windows.

In this paper, we introduce the PVR (Parallel Volume Rendering) system, developed under collaboration between the State University of New York at Stony Brook and Sandia National Laboratories. Unlike the other systems PVR is a component approach to building an interactive distributed volume visualization system. At its topmost level, it provides a flexible and high performance client/server volume rendering architecture with a unique load balancing scheme which provides a continuum of cost/performance parameters that can be used to optimize rendering speed.

The rest of the paper introduces the PVR client/server architecture and its components, with an emphasis on its support for volume rendering.

2 Overview of PVR

The original goals of PVR were to achieve a level of portability and performance for rendering beyond that of other available systems and to provide a platform that can be used for further development. In a certain way, PVR is more than a rendering system; its components have been specially designed to be user-extensible in order to allow for user-defined computational steering. That is, the user can easily add custom computational code to PVR and just link in the rendering library. Using PVR, it is much easier to build portable, high performance, complex, distributed visualization systems. Figure 1 displays the relationship between PVR and a distributed visualization environment.

It is well known that system complexity limits the reliability of large software systems.

Distributed systems exacerbate this problem with the introduction of asynchronous and non-local communication. With all of this in mind, we have used a component approach in developing our system. PVR attempts to provide just enough functionality in the basic system to allow for the development of large and complex visualization and computational steering applications. It is based on a client/server architecture, where there are, on one side, rendering/computing servers which are coupled, and, on the other side, the user acting as a client from his workstation.

The PVR client/server architecture is implemented in two main components: the *pvrsh*, which runs in the user's workstation, and the *PVR renderer*, which runs in the parallel machines. The renderer is implemented as a library and it allows for easy integration of user-defined code that can share the same processors as the rendering code. Communication across applications written with PVR are performed using the PVR protocol, and in our implementation communication is handled by separate UNIX processes (see Figure 2).

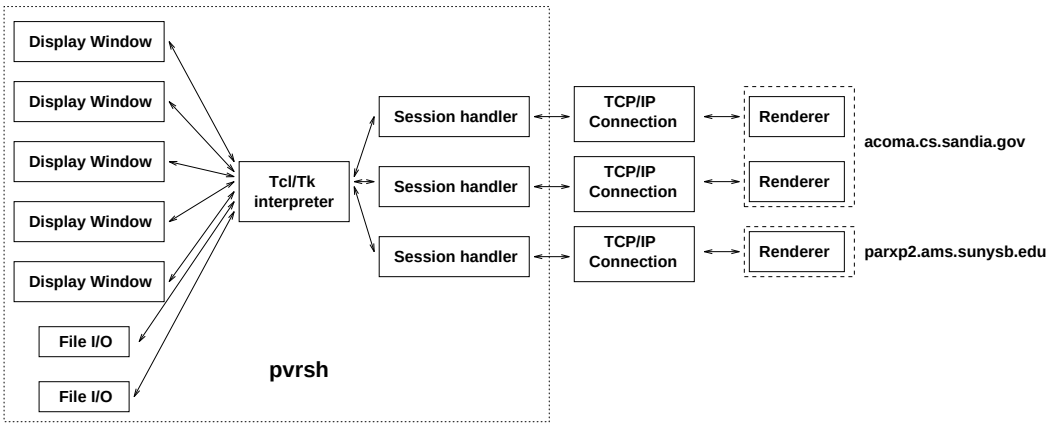


Figure 2: *PVR Architecture. The overall structure of the system is shown with an emphasis on the* pvrsh. *The Tcl/Tk core acts as glue for all the client components. Everything, with the exception of the* renderers*, runs on the user's workstation. The* renderers *run remotely on the parallel machines.*

3 The *pvrsh*

The *pvrsh* provides a single new object to the user, the *PVR session*. The *pvrsh* is an augmented Tcl/Tk shell. We chose to use Tcl/Tk as the system glue. Tcl, *Tool command language*, is a script language designed to be used as a generic language in application programs. It is easily extendible with new user commands, in C or Tcl, and coupled with the graphical environment Tk, it is a powerful graphical user-interface system. The use of the

Tcl/Tk, which is a well-designed, debugged application language and graphical environment has contributed to reducing the overall system complexity.

The *PVR session* is an object (such as the Tk objects). It contains attributes, and corresponding methods are used to change the attributes. One of the most important attributes is the one that *binds* a session to a particular parallel machine. Figure 2 contains an example of three sessions, two on acoma.cs.sandia.gov (a large Intel Paragon XP/S with over 1840 nodes running SUNMOS [5], installed at Sandia) and one on parxp2.ams.sunysb.edu (a small Intel Paragon with 110 nodes running Intel version of OSF/1, installed at Stony Brook). The system is designed to handle multiple sessions using the same protocol with machines running different operating systems.

As part of its attributes, a session specifies the number of nodes it needs and the parameters that are passed to those nodes. Several pieces of informations are *interactively* exchanged between the *pvrsh* and the *PVR renderer*, such as rendering configuration information, rendering commands, sequences of images, performance and debugging information.

There is a high amount of flexibility in the specification of the rendering. Not only can simple rendering elements, such as changing transformation matrices, transfer functions, image sizes and datasets be specified, but there are commands (see Table 1) to specify in a high level format the complete parallel rendering pipeline (see sidebar *Parallel Volume Rendering* for details). With these parameters in hand, the *pvrsh* can be used to specify almost arbitrary scalable rendering configurations (see Section 6).

The *pvrsh* is implemented as a single process (making ports easier) in about 5,000 lines of C code. We have augmented the Tcl/Tk interpreter with TCP/IP connection capabilities (some versions of Tcl/Tk have this built in). In order to support several concurrent sessions, all the communication is performed asynchronously. We use the `Tk_CreateFileHandler()` routine to arbitrate between input from the different sessions. A UNIX `select` call and polling could be used instead but would make the code harder to understand and, overall, more complex. Sessions work as interrupt-driven commands, responding to requests one at a time. Every session can receive events from two sources at the same time: the user keyboard and the remote machine. Locking and disabling interrupts are needed to ensure consistency inside critical sessions.

The overall structure of the code allows for user augmentation of a session functionality either by external or internal means. *External* augmentation can be performed without recompilation, such as that used by the user interface to show images as they are received asynchronously from the remote parallel server. *Internal* augmentation requires changes to the source code. The source code is structured to allow for simple addition of new

functionality. Only a single file needs to be changed to add a new session method. If it changes the *Resource Database* [9], two files need to be changed. New commands are added using Tcl conventions.

Every PVR message is sent either as a single fixed-length message, or as two messages (the first is used to specify the size of the second). This is used to make redirection easier and to achieve optimal performance under different configurations. Look-up tables are set up with actions to be taken up on the arrival of each message type. This setup makes additions to the PVR protocol very simple.

Command	Description
:s open *M:N*	*M* is an internet address; *N* is a port number.
:s close	Close the connection.
: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 list status	Show the state of the connection and the value of internal variables.
:s set *Option Val*	Change system status.
:s set -dataset *D*	Sets the dataset to be rendered.
:s set -cluster *C*	Sets the size of clusters.
:s set -group *G*	Used to group multiple clusters, for use in exploiting 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. The axis of rotation and initial, end, and incremental angles are specified.
:s performance memory cluster	Returns the amount of dataset memory in each cluster.
:s performance comp cluster latency	Estimates the latency time to composite images in the current cluster configuration.

Table 1: *A list of a few external PVR commands. These commands can be typed interactively, placed in execution files, or embedded in applications.*

4 The *PVR Renderer*

The *PVR renderer* is the piece of PVR that runs remotely on a parallel machine (see Figure 2). It is composed of several components, the most complex being the rendering code itself. In order to start up multiple parallel processes at the remote machine, we use *pvrd*, the PVR daemon. This daemon runs on the parallel machine. It waits on a well-known port for connection requests. Once a request for opening a new session is made, it *forks* a handling process that is responsible for allocating processors and communicating with the session on the client. On the remote machine, the handling process allocates the computing nodes and runs the renderer code on them. The connection process is illustrated in Figure 3. One *pvrd* can allocate several processes; once it is killed, it kills all its children before exiting.

The renderer is the code that actually runs on the parallel nodes. The overall structure of the code resembles a SIMD machine, where there are high-level commands and low-level commands. There is one *master* node, similar to the microcontroller on the CM-2 machines, and several *slave* nodes. The functions of the slaves are completely dependent on the master. The master receives commands from the *pvrsh*, translates them, and takes the necessary actions, including changing the state of the slaves and sending them a detailed set of instructions.

For flexibility and performance, the method of sending instructions to the nodes is through *action tables* (similar to SIMD microcode). In order to ask the nodes to perform some action, the master broadcasts the address of the function to be executed. Upon receiving that instruction, the slaves execute that particular function. With this method, it is very simple to add new functionality because any new added functionality can be performed locally, without the need to change global files. Also, every function can be optimized independently, with its own communication protocol. One shortcoming of this communication method (as in SIMD machines) is that one has to be careful with non-uniform 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 is impossible to select a group from the totality of the allocated nodes. Newer communication libraries, such as MPI [11], solve this shortcoming by introducing the idea of groups of nodes.

The master intrinsically divides the nodes into *clusters*. Each cluster has a specialized computational task, and multiple clusters can cooperate in groups to achieve a larger task. All that is necessary for cluster configuration is that the basic functions be specified in user-defined libraries that are linked in a single binary. During runtime, the user can use

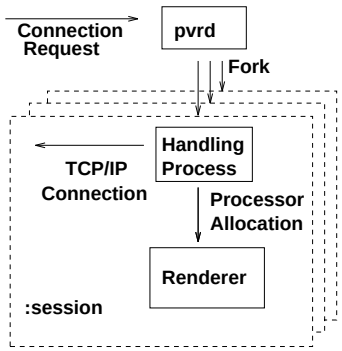


Figure 3: *In order to allocate nodes, the* pvrsh *sends a command to the* pvrd*, which in turn creates a special communication handling process and allocates a partition on the parallel machine.*

the master to reconfigure clusters according to immediate goals. The *pvrsh* can be used to *interactively* send such commands. As an example of the use of such a clustering scheme, see Figure 4, where the rendering configuration for PVR high performance volume renderer is depicted.

In order to achieve user-defined computational steering, one can use this clustering paradigm. It is usually necessary to add one's functionality to the action tables (e.g., linking the computational code with PVR dispatching code), and also add extra options to the *pvrsh* (usually through the `set` command) for modifying the relevant parameters interactively.

PVR volume rendering code was the inspiration for this overall code organization and is a very good application to demonstrate its features. Because in this paper our focus is on describing the PVR system, and not on the actual volume rendering code, we only sketch the implementation to give an insight as to how to add your own code to PVR and to give you enough information for effective use of the PVR rendering facilities.

5 Volume-Rendering Pipeline

The PVR rendering pipeline is composed of three types of nodes (besides the master). These are the *rendering nodes*, *compositing nodes*, and *collector nodes* (usually just one), (see Figure 4). This specialization is necessary for optimal rendering performance and flexibility. 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 15 and 16). The master configures (and re-configures) the overall dataflow using a set of user-defined and automatic load-balancing parameters.

At the top level are the rendering clusters. The nodes in a rendering cluster are responsible for resampling and shading of a given volume dataset. In general, the input is a view matrix, and the output is a set of sub-images, each of which is a 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 in order to speed up rendering. Once the sub-images are computed, 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 which corresponds to the decomposition of the volume dataset on the rendering nodes. The number of processors used to do compositing can actually be different than the number of nodes in the compositing tree, as we can use *virtualization* to fake more processors than allocated. Images are pipelined down the tree, with every iteration combining the results of compositing until finally all the pixels are a complete depth-ordered sequence. Those pixels are converted to RGB format and sent to the collector node(s) (at this time, we just use a single collector node).

The collector node receives RGB images from the compositing nodes and compresses them using a simple run-length encoding scheme (very fast compression is necessary). Finally, the images are either sent over to the *pvrsh* for user viewing (or saving), or locally cached on the disk. An additional option allows images to be trashed for performance analysis purposes.

The previous discussion is somewhat simplistic. There are several performance issues related to CPU speed, synchronization, and memory usage that have not been discussed. For more complete details, we refer the interested reader to [9].

6 Rendering with PVR

Figure 5 shows a simple PVR program. Several important features of PVR are demonstrated: in particular, 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 the images that are output by the system.

A *cluster* of multiple nodes and a *group* of clusters are the two basic components of the PVR load balancing scheme, and they are used together to specify flexible configurations of image-space, object-space and time-space parallelism. Rendering clusters can be assigned different scanlines of an image and each group of clusters is assigned a complete image at a time by the *master* node. The `-cluster` and `-group` options are used to specify this unique

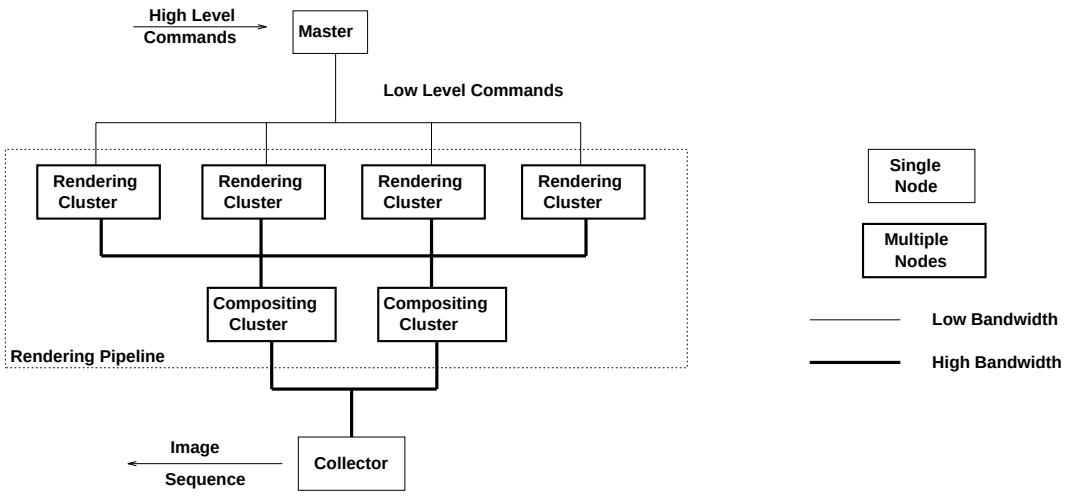


Figure 4: *The master receives high level commands that are translated into virtual* microcode *by the action tables. For rendering, the high level commands are for the generation of animations by rotations and translations, which are interpreted into simple transformation matrices commands. The rendering clusters perform rendering in parallel. The collector receives and groups images together and sends an ordered image sequence to the client application.*

capability of the PVR flexible load-balancing scheme. With both of these options, the relative sizes of the rendering and compositing clusters can be specified together with the image calculation allocation. Several scalability strategies can be used. A rendering cluster needs to be large enough to hold the entire dataset and at least a copy of the image. By increasing the size of the cluster (i.e., the number of nodes in the cluster), the amount of memory needed per node decreases. By grouping clusters (i.e., splitting the image computation across multiple clusters), the number of scanlines a given cluster is responsible for decreases, lowering both the image memory requirements and the computational cost, thus speeding up image calculation.

The same commands can be used to configure compositing clusters. Compositing clusters do not scale at the same rate as rendering clusters, because of the different nature of the task. Compositing is a relative light computation, high synchronization operation, as opposed to rendering. Compositing nodes need memory to hold two copies of the images, which can be quite large (our current parallel machine nodes have only between 16MB to 32MB RAM). The compositing latency increases as the number of nodes increase (the actual rate of increase depends on the height of the compositing tree). At this time, we use multiple compositing clusters in order to composite very large images (e.g., 1024×1024 or larger), not because of the computation bottleneck, but just for the lack of memory on our parallel nodes.

```
toplevel .rgb                          ; Tcl/Tk stuff – creates necessary windows
photo .rgb.p
pack .rgb.p
toplevel .c
canvas .c.c
pack .c.c
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 image dir ./                    ; where to place images
: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 dataset
: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 -cluster c -group 0,0       ; 2 compositing clusters of 15 nodes
                                       ; each, this allows for the calculation of very
                                       ; large images, as each cluster will handle half
                                       ; of pixels coming from the rendering nodes
: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. The commands can be put in a file and executed in batch, or can be typed interactively on the keyboard (or mixed). Tcl/Tk code (for example, "stat.tcl") can be written to take care of portions of the actions.*

7 Visualization Services

Our system architecture can be used to visualize time-varying data. When rendering time-varying data, we add a permanent *caching cluster* to the pipeline in Figure 4 which is responsible for distributing the volume data to the rendering nodes efficiently. The caching nodes are used only as *smart* memory and they are used to hide I/O latency from disk (or other sources), and in our content-based load balancing data distribution. This way, the user can visualize a dataset for as long as it takes a new version of the dataset to come along. Handling data that changes too rapidly (i.e., faster than we can move it and render it) is not possible, as it would require large amounts of buffering.

Another possible use of our parallel renderer is as a visualization server for large computational parallel jobs [7]. The basic idea is to pre-allocate a set of nodes that can be shared to a limited extent by multiple users for visualizing their data. Effective use as such a server would also make use of a caching cluster, as described above for time-varying data. The cluster, in this case, would be used to cache in alternate user datasets.

Distributed visualization environments can be developed by making use of the client/server metaphor. A DVE developed using Tcl/Tk is very portable, as Tcl/Tk has ports for almost all of the operating systems available, and TCP/IP (our communication protocol) is virtually universal. We give more details on the primitives from which DVEs can be built in Table 1.

Figure 13 shows a simple prototype GUI developed at Sandia. The complete interface is written in Tcl/Tk. The user is able to specify all the necessary rendering parameters in the right window (including image size, transfer function, etc.) and the load-balancing parameters in the left window. This simple interface uses only a single session at this time, but more functionality is currently being added to the system.

Using the prototype GUI, users are able to add their own functionality to the system as needed. This flexibility not only makes the system more usable, because redundant bells and whistles can be discarded, but also new functionality can be added straightforwardly. The use of a portable and well-documented windows interface (e.g., Tk) is imperative. Not only do users avoid having to learn yet another programming language and graphical toolkit, but the use of Tk saved us a lot of implementation and documentation cost (Tcl/Tk is widely used and well-documented). Another important feature of Tcl/Tk for the development of prototypes is that it is freely available, enabling us to do the same for PVR.

8 Results

The current version of PVR consists of about 25,000 lines of C and Tcl/Tk code. It has been used at Brookhaven National Labs, Sandia National Labs, and Stony Brook to visualize large datasets for over a year. Below we discuss a few of the current uses and performance of PVR. The biggest challenge we have faced so far is the limited amount of memory on our Paragon nodes. It is very hard from the software engineering point of view to have consistent and reliable treatment of memory allocation issues, specially when attempting to visualize very large datasets.

We have demonstrated the capability of rendering a 500MB dataset (the 512 $\times$ 512 $\times$ 1877 CT visible human dataset from the National Institute of Health – see Figure 14) using approximately 128 rendering nodes and 127 compositing nodes of the Intel Paragon at Sandia remotely displaying at Supercomputing '95 in San Diego. The rendering times for a 512 $\times$ 512 image are on the order of 5 seconds/frame. It is worth pointing out that the main bottleneck for this dataset is reading the 500MB of data from the Paragon disks. Currently, it takes around 15 minutes.

Figure 6 shows the rendering times for each frame of a 72-frame animation sequence of the visible human dataset. 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 $\times$ 400, with three color channels. For rendering, the images are represented as an array of pixels, each of which is represented as four floating point numbers (what amounts to 16 bytes per pixel). At 400 $\times$ 400, each image is over 2.5MB. Images are transmitted from the rendering nodes to the compositing nodes, until they reach the root node of the compositing tree. There, images are converted to RGB format, with one byte per color channel, and transmitted to the collector node. The final images (with 480,000 bytes) are saved to disk by the collector. Computing the complete animation takes 129.23 seconds, or 1.79 seconds per frame, resulting in 32MB of data being saved to disk. There are noticeable sparks in the image generation rate and these deserve further study. We hypothesize the source of the stalls in the pipeline are due to load imbalance and also contention in writing the images to the disk (the collector node stalls the pipeline whenever an image is received before the previous image is saved). One can see 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 (14GB) with high temporal resolution (a 72-frame rotation uses 5 degree increments. Smaller increments are highly desirable, but a 0.5 degree increment would make the animation files huge, at

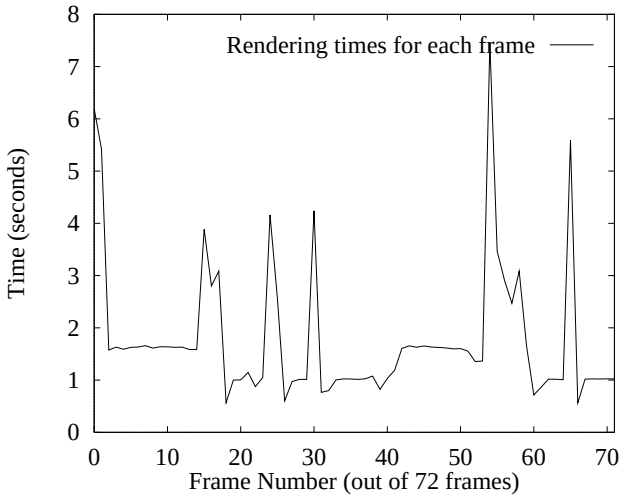
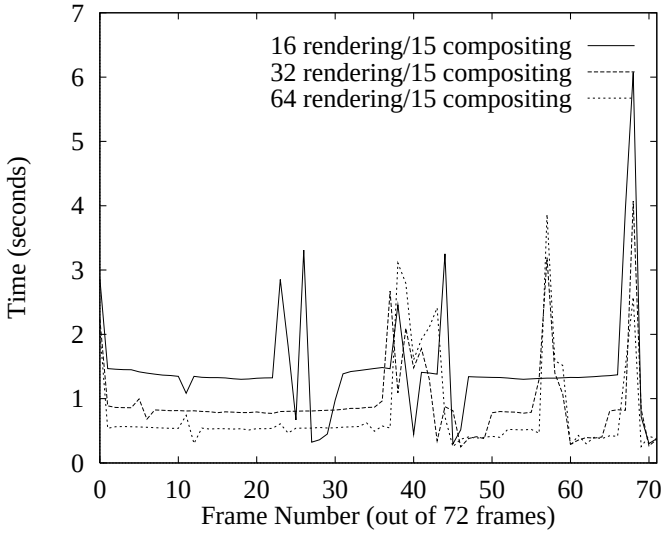


Figure 6: *Rendering times for a 72-frame animation sequence of the 512 × 512 × 1877 visible human dataset. Each image is 400 × 400.*

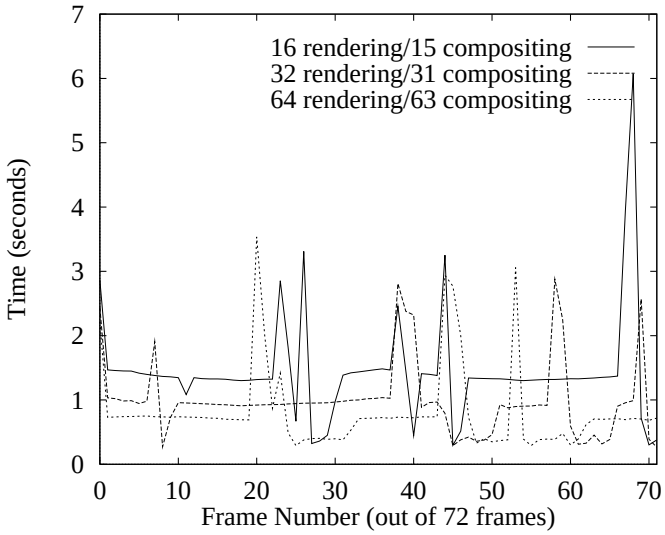
over 300MB). This requires the use of parallel I/O, a capability that currently we do not have, and dedicated use of a very large parallel machine, such as the entire 1840-node Intel Paragon at Sandia.

In order to show the scalability of PVR, we use a 256 × 256 × 937 version of the visible human dataset. Figure 7 shows the rendering times for 5 different configurations, varying the number of rendering and compositing nodes. The five configurations are: 16 rendering nodes and 15 compositing nodes (total rendering time (TRT) is 104.10 seconds, or 1.44 seconds per frame); 32 rendering nodes (2 clusters of 16) and 15 compositing nodes (TRT is 67.24 seconds, or .93 seconds per frame); 64 rendering nodes (4 clusters of 16) and 15 compositing nodes (TRT is 56.73 seconds, or .78 seconds per frame); 32 rendering nodes (1 cluster) and 31 compositing nodes (TRT is 71.42 seconds and .99 seconds per frame); 64 rendering nodes (1 cluster) and 63 compositing nodes (TRT is 58.79 seconds, or .81 seconds per frame). A simple conclusion that can be drawn from this data is that it is not cost effective to increase the size of the compositing cluster for relatively small images [9].

Figure 9 is a volume rendering of a 1024 × 1024 × 64 digital, 3D flourecent microscopy dataset showing T-cells in a thick section of thymic tissue. Figure 10 is a volume rendering of a 100 × 110 × 92 dataset showing T-cell receptor density on the surface of a T-cell/B-cell interaction. The datasets were generated by immunofluorescence microscopy at the National Jewish Center for Immunology and Respiratory and prepared for visualization by deconvolution on Sandia Intel Paragon. The volume rendering animations were generated at multiple frames per second using PVR [6].



(a) Scaling the number of clusters in a rendering group.



(b) Scaling both the rendering and compositing nodes.

Figure 7: *Rendering times for a 72-frame animation sequence of a 256 × 256 × 937 version of the visible human dataset. Each image is 250 × 250.*

9 Conclusions

In this article, we have introduced the PVR system. It has the following key features:

- *Transparency* - PVR hides most of the hardware dependencies from the DVEs and the user.
- *Performance* - PVR provides high speed pipelined ray casting with a unique load-balancing scheme and mechanisms to fine tune performance for any given machine configuration.
- *Scalability* - All the algorithms used in the system are gracefully scalable. Scalability is not only with respect to the machine size, but also allows for growth in dataset size and image size.
- *Extensibility* - The PVR architecture can be easily extended, making it easy for the DVE to add new functionality. Also, it is fairly easy for the user to add new functionality to the PVR shell and its corresponding kernel, allowing for user-defined "computational steering" coupled with visualization.

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

PVR introduces a simple way to create user interfaces. No longer does one have to 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. Tcl/Tk is becoming as popular as UNIX shell programming. Different sites should be easily able to create and/or customize their own versions of the systems.

Even though we have completed a *usable* and efficient system, much work remains to be done. We are working to make the system stable enough for general distribution, and are creating a more complete DVE (using VolVis [2] as a reference) on top of PVR.

Some functionality is missing from PVR and needs to be incorporated. The most important element is probably the support for multiple data sets in a session. The implementation of this capability may make the load-balancing scheme much more complicated, 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 each time a new volume is introduced into the picture. It is not yet clear how this could be done efficiently.

Research is currently underway to incorporate irregular grid rendering in PVR. We are currently looking to incorporate a recent algorithm [10] that is able to exploit a high level of locality, which should ultimately lead to more efficient communication schemes. We are currently porting PVR to use MPI as the communication layer, instead of NX.

Acknowledgments

We would like to thank Maurice Fan Lok who co-wrote the first version of PVR, Brian Wylie who supported the project and the development of the user interface, and Dirk Bartz, Tzi-cker Chiueh, Pat Crossno, Steve Dawson, Juliana Freire, Tong Lee, Ron Peierls, and Amitabh Varshney for useful discussions about the PVR system and this paper. The port of PVR to SUNMOS was only possible due to the help of Kevin McCurley, Rolf Riesen, Lance Shuler from Sandia, and Edward J. Barragy from Intel. The MRI head dataset is courtesy of Siemens. The visible human data is courtesy of the National Institute of Health. The cell and tissue data are courtesy of 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 Department of Energy Mathematics, Information and Computer Science Office, and by the National Science Foundation (NSF), grant CDA-9626370. A. Kaufman is partially supported by NSF under grants CCR-9205047, DCA 9303181, MIP-9527694 and by the Department of Energy under the PICS grant.

References

[1] V. Anupam, C. Bajaj, D. Schikore, and M. Schikore. Distributed and collaborative visualization. *IEEE Computer*, 27(7):37–43, 1994.

[2] R. Avila, T. He, L. Hong, A. Kaufman, H. Pfister, C. Silva, L. Sobierajski, and S. Wang. VolVis: A diversified volume visualization system. In *IEEE Visualization '94*, pages 31–38. IEEE CS Press, October 1994.

[3] A. E. Kaufman. *Volume Visualization*. IEEE Computer Society Press, Los Alamitos, CA, 1991.

[4] P.-W. Liu, L.-S. Chen, S.-C. Chen, J.-P. Chen, F.-Y. Lin, and S.-S. Hwang. Distributed computing: New power for scientific visualization. *IEEE Computer Graphics and Applications*, 16(3):42–51, 1996.

[5] A. Maccabe, K. McCurley, R. Riesen, and S. Wheat. Sunmos for the Intel Paragon - A Brief User's Guide. In *Proceedings of the Intel Supercomputer Users' Group 1993 Annual North America Users' Conference*, October 1993.

[6] C. Monks, P. Crossno, G. Davidson, C. Pavlakos, A. Kupfer, C. Silva and B. Wylie. Three Dimensional Visualization of Proteins in Cellular Interactions. In *IEEE Visualization '96*. (To appear)

[7] C. Pavlakos, L. Schoof, and J. Mareda. A visualization model for supercomputing environments. *IEEE Parallel & Distributed Technology*, 1(4):16–22, 1996.

[8] J. Rowlan, E. Lent, N. Gokhale, and S. Bradshaw. A distributed, parallel, interactive volume rendering package. In *IEEE Visualization '94*, pages 21–30. IEEE CS Press, October 1994.

[9] C. Silva. *Parallel Volume Rendering of Irregular Grids*. Ph.D. thesis, Department of Computer Science, State University of New York at Stony Brook, 1996.

[10] C. Silva, J. S. B. Mitchell, and A. E. Kaufman. Fast rendering of irregular grids. In *IEEE/ACM Volume Visualization Symposium '96*. (To appear)

[11] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, 1995.

Sidebar: Electronic Information

PVR Information, including related publications, images, animations, can be found in [1, 2]. The source code (Paragon version only) is currently available for users willing to provide feedback to our beta testing program. (Send mail to `csilva@cs.sunysb.edu`.) Information on the visible human project is available from [3], and on the Helper T-Cell/B-Cell interaction from [4, 5].

Several animations are available from our web sites. All these animation are best viewed on high quality MPEG viewers, such as the SGI movieplayer.

- `http://www.cs.sunysb.edu/~csilva/mpeg/Yhead-350.mpeg`
  This animation shows the effect of our content-based load balancing approach for a MRI head dataset using 8 processors. In particular, one can see that the amount of "volume" given in each subdivision is proportional to the amount of non-empty voxels in the subdivision.

- `http://www.cs.sunysb.edu/~csilva/mpeg/human.mpeg`
  This is an animation of a volume rendering of the frozen CT visible human dataset. Some registration artifacts can be seen in the animation.

- `http://www.cs.sandia.gov/VIS/mpeg/xcell.mpg`
  This is an animation of the volume rendering of a single cell.

- `http://www.cs.sandia.gov/VIS/mpeg/tissue.mpg`
  This is an animation of the volume rendering of the thymus tissue.

References

[1] `http://www.cs.sunysb.edu/~csilva/claudio-pvr.html`

[2] `http://www.cs.sandia.gov/VIS`

[3] `http://www.nlm.nih.gov/extramural_research.dir/visible_human.html`

[4] `http://www.cs.sandia.gov/VIS/colin.html`

[5] `http://www.njc.org/profinfohtml/SC95.html`

Sidebar: Volume Rendering

Volume rendering accumulates information from all voxels in the 3D dataset to produce a 2D image, enabling a comprehensive examination of the structures in a dataset. The technique works by modeling the volumetric dataset as a cloud-like material that scatters, emits and absorbs light [3]. Ray casting is one volume rendering algorithm. For every pixel in the image a ray is cast in object space (or volume space). Roughly speaking, 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 intensity of light emanating from a given portion of the volume and $\sigma(s)$ is the differential absorption of light (used to calculate attenuation along the viewing direction). The integral is usually calculated by a simple numerical quadrature scheme, most commonly a set of uniform samples are taken. $I(t)$ and $\sigma(t)$ are usually calculated by assigning *transfer functions*. Transfer functions are table lookups based on the original volume data $f(x, y, z)$, which is computed by trilinearly interpolating the eight values defined at the closest points in a volume. Sometimes a light dependent term is added to $I(t)$, usually by the use of Phong shading. Each sample contains the color and opacity at a certain distance from the eye (see Figure 8). With the color and opacity in hand, it is very simple to accumulate the final pixel, either back-to-front or front-to-back. This accumulation process is called "compositing" and can be defined using the well known **over** operator.

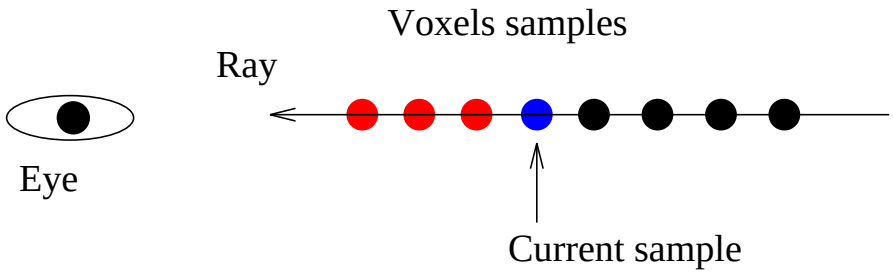


Figure 8: *A typical ray is shown, together with its samples.*

For instance, Figure 8 depicts a back-to-front compositing. If C is the color of the current voxel, α its opacity, and I the incoming intensity of color at the current voxel. The outgoing intensity, I' is given by the **over** operator:

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

In these equations the colors are saved pre-multiplied by the opacities (i.e., the actual color is C/α), this saves one multiplication per compositing operation. It is easy to see that

compositing is associative, that is, ((A **over** B) **over** C) produces the same result as (A **over** (B **over** C)). This property is important in the parallelization of volume rendering.

Transfer functions are used to specify what portions of the volume are relevant to be visualized. Transfer functions acts just like color maps, they specify color (RGB) and opacity for each voxel in the volume. In looking for interesting properties in the data, it is imperative to be able to try different combinations of transfer functions and viewing parameters (see Figure 11). In cases where the data is very complex without visible hard edges, one effective way to see patterns in the data is by using motion. Our eyes are very well trained to extract 3-dimensional information from animations (such as rotations). Unfortunately, volume rendering is typically very slow, even for small datasets, and especially when the volume is relatively transparent. For instance, using VolVis [1], a state-of-the-art volume visualization system developed at Stony Brook, it takes hours to generate animations of all the datasets shown in this paper. Performing visualization in such a manner is of limited use as it is counter-productive to have to wait hours for animations that might not contain useful information. By using the new Parallel Volume Rendering system (PVR), we are able to generate even the largest animations in a few seconds to a few minutes because the system easily scales to provide the desirable performance (e.g., the larger the dataset, the more nodes we use).

Further information on obtaining VolVis (free with full source code) is available by sending e-mail to `volvis@cs.sunysb.edu` or at `http://www.cs.sunysb.edu/~volvis`. The survey by Kaufman [2] provides papers and further pointers on volume rendering.

References

[1] R. Avila, T. He, L. Hong, A. Kaufman, H. Pfister, C. Silva, L. Sobierajski, and S. Wang. VolVis: A diversified volume visualization system. In *IEEE Visualization '94*, pages 31–38. IEEE CS Press, October 1994.

[2] A. E. Kaufman. *Volume Visualization*. IEEE Computer Society Press, Los Alamitos, CA, 1991.

[3] N. Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, June 1995.

Sidebar: Parallel Volume Rendering

The need for faster rendering of very large datasets coupled with the wider availability of parallel and distributed machines are the main push behind parallel volume rendering research.

In parallel volume rendering, three main types of parallelism can be exploited:

- *Object-space parallelism*, where each rendering node gets a portion of the dataset.
- *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 in a pipeline fashion among independent set of nodes.

There are several different parallel algorithms that have been developed based on these types of parallelism. A recent survey on parallel rendering by Tom Crockett [1] is a good starting point for the interested reader. The Parallel Rendering Symposia ('93, '95), the ACM Volume Visualization Symposia and the IEEE Visualization Conferences are further sources for current information on parallel volume rendering. In a nutshell, large datasets are usually divided up among nodes, or group of nodes, where pieces of images are rendered independently and later composited together.

Our particular interest is in ray casting methods that run on distributed memory machines, such as the Intel Paragon and the ASCI Teraflop machine. In ray casting, rays are cast into the volume for each pixel position and samples are calculated along the ray, at equally spaced positions. Each sample represents a shaded color and opacity. After all these calculations, they are composited into a single image (see the Volume Rendering sidebar).

In a distributed memory machine, where each node has memory access limited to its local memory, it is necessary to divide the dataset among computing nodes. In turn, this subdivision requires that the volume samples be grouped back together in an image. All the ray casting parallel methods differ primarily in the way they handle these two primitives. Here, because of space constraints, we only provided an overview of the parallelization method used in PVR, which is based on a combination of the dataset load balancing proposed in [5], and the compositing method proposed in [2].

In PVR, we implement a parallel volume rendering pipeline in the canonical way (see Figures 15 and 16), the rendering nodes receive portions of the dataset, divided in such a way as to optimize the global load balancing. Every rendering node receives a portion of

the dataset that has approximately the same number of non-empty voxels (see Figure 12). Other approaches, such as giving the exact same amount of volume to each node is also feasible (and is used in [2], among others). Dynamic load balancing schemes have been tried [4], but are harder to implement, specially for extremely large datasets, or in machines with very limited memory resources.

The PVR rendering nodes are responsible for sampling and compositing their part of a ray. In order to avoid global communication, each sub-volume region assigned to a rendering node is convex, and belongs to a global BSP-tree, which makes compositing simpler (see Figure 15). The compositing nodes are responsible for regrouping all the sub-rays back together in a consistent manner, in order to keep image correctness. This calculation is only possible because composition is an associative operation, so if we have to sub-ray samples where one ends and the other starts, it is possible to combine their samples into one sub-ray recursively until we have a value that constitutes the full ray contribution to a pixel.

Ma et al. [2] use a different approach to compositing, where instead of having separate compositing nodes, the rendering nodes switch between rendering and compositing. Our method is more efficient (in his latest paper, Ma [3] adopts a similar decomposition of the nodes into two classes) because we can use the special structure of the sub-ray composition to yield a high performance pipeline, where multiple nodes are used to implement the complete pipeline (see Figure 16). Also, the structure of compositing requires synchronized operation (e.g., there is an explicit structure to the composition, that needs to be guaranteed for correctness purposes), and light weight computation, making it much less attractive for parallelization over a large number of processors, specially on machines with slow communication compared to CPU speeds (almost all current machines).

In summary, our implementation of volume rendering divides the processors into two distinct groups of nodes: rendering and compositing nodes. This fits well with our clustering scheme explained before. The rendering nodes get portions of the dataset, the compositing nodes are responsible for turning a collection of sub-ray images into a complete and correct image for viewing.

The structure of the PVR rendering pipeline makes it possible to exploit all the three types of parallelism. For instance, by using more than a single rendering cluster to compute an image, we are making use of "image-space paralellism" (in PVR, it is possible to specify that each cluster compute disjoint scanlines of the same image; see [6] for the issues related to image-space parallelism). The clustering approach coupled with the inherent pipeline parallelism available in the compositing process (because of its recursive structure) gives rise to "time-space paralellism". In the latter, we can exploit multiple clusters by concurrently

calculating sub-rays for several images at once, that can be sent down the compositing pipeline concurrently. Here, it is important for the correctness of the images, that each composition step be done in lockstep, in order to avoid mixing of images.

References

[1] T. W. Crockett. Parallel rendering. In A. Kent and J. G. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 34, Supp. 19, A., pages 335–371. Marcel Dekker, 1996. (Also available as ICASE Report No. 95-31 (NASA CR-195080), April 1995.).

[2] K. Ma, J. Painter, C. Hansen, and M. Krogh. Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics and Applications*, 14(4):59–68, 1994.

[3] K. Ma. Parallel volume rendering for unstructured-grid data on distributed memory machines. In *IEEE/ACM Parallel Rendering Symposium '95*, pages 23–30, 1995.

[4] U. Neumann. Parallel volume-rendering algorithm performance on mesh-connected multicomputers. In *1993 Parallel Rendering Symposium Proceedings*, pages 97–104. ACM Press, October 1993.

[5] C. Silva and A. Kaufman. Parallel performance measures for volume ray casting. In *IEEE Visualization '94*, pages 196–203. IEEE CS Press, October 1994.

[6] J. P. Singh, A. Gupta, and M. Levoy. Parallel visualization algorithms: Performance and architectural implications. *IEEE Computer*, 27(7):45–55, 1994.

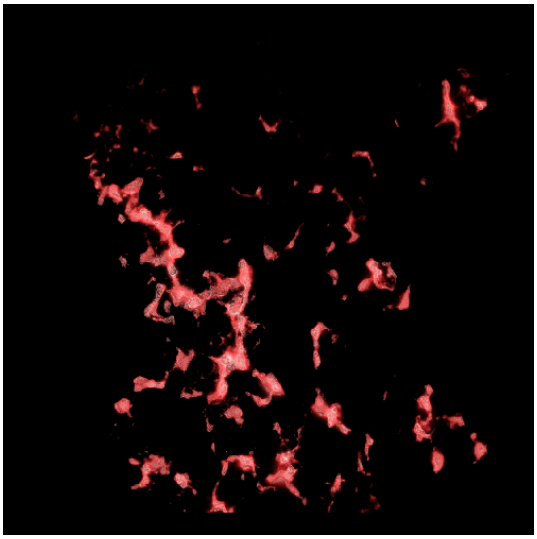


Figure 9: Volume rendering of the Thymus tissue. An (512 × 512, 72 frame) animation was produced in just about 3 minutes using PVR on the Paragon.

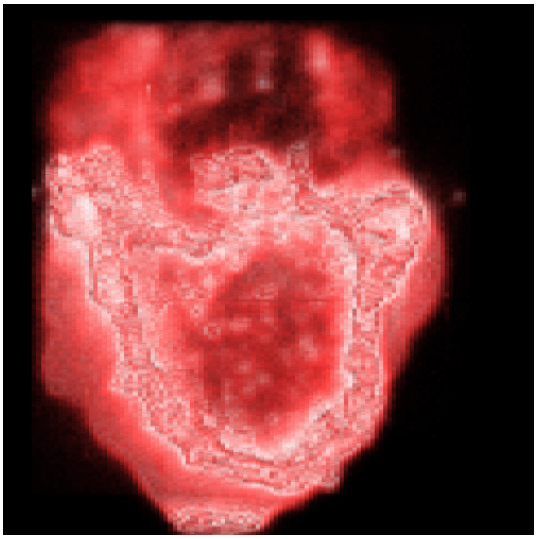


Figure 10: A volume rendering showing T-cell receptors on an immuno-flourecent microscopy dataset.

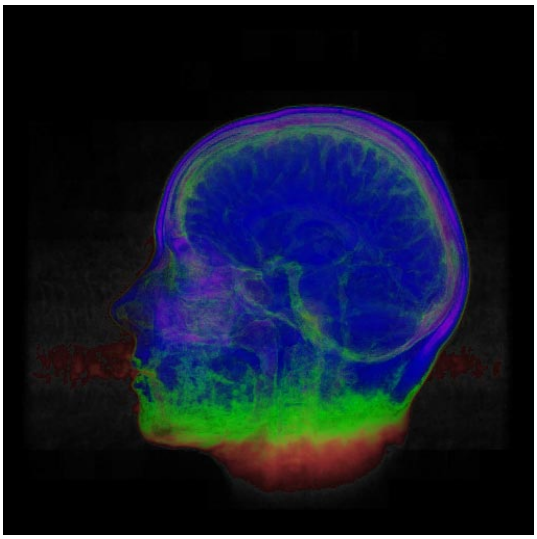


Figure 11: Volume rendering of MR data from a human head using an unconventional transfer function in order to illustrate the flexibility of volume rendering.

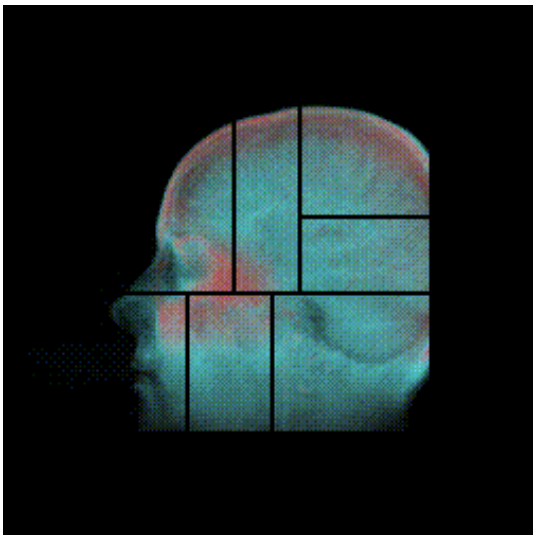


Figure 12: A subdivision of the MR data for 8 processors is shown, illustrating our content-based load balancing.

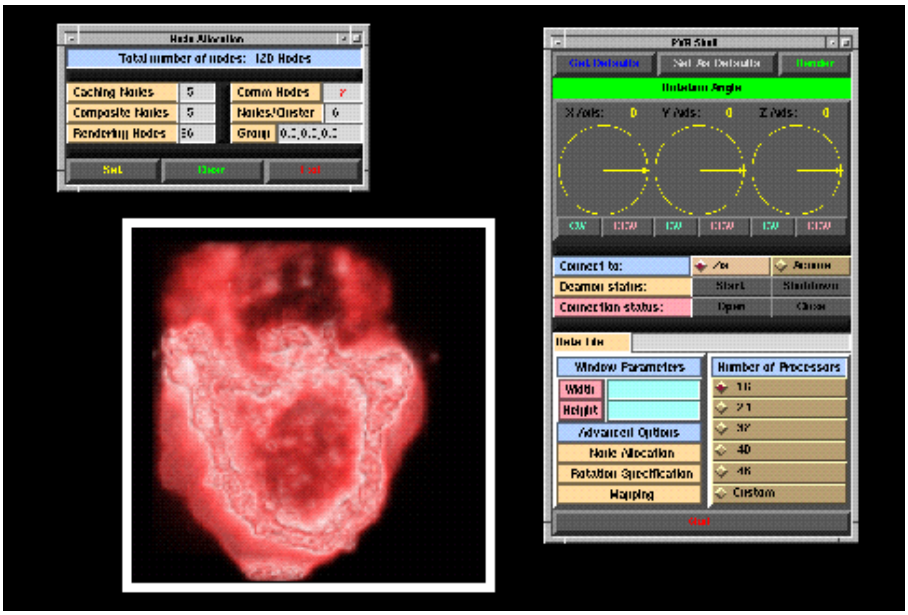


Figure 13: A snapshot of the simple PVR GUI, with three windows. The main interface window on the right, where the user can specify general rotations. The cluster configuration window, on the left. The third window is the image of a cell calculated with PVR.

Figure 14: Volume rendering of the $512 \times 512 \times 1877$ visible human.

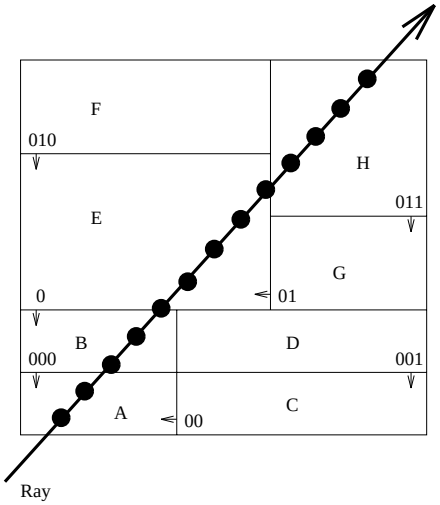


Figure 15: *Data partitioning shown in two dimensions. The dataset is partitioned into 8 pieces (marked A ...H) in a canonical hierarchical manner by the 7 lines (planes in 3D) represented by binary numbers. Once such a decomposition is performed, it is relatively easy to see how the samples get composited back into a single value.*

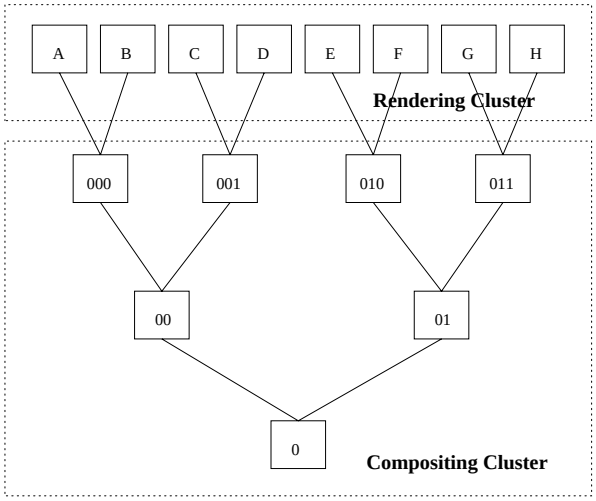


Figure 16: *The internal structure of one compositing cluster, one rendering cluster and their interconnection is shown. In PVR, the communication between the compositing and the rendering clusters is very flexible, with several rendering clusters being able to work together in the same image. This is accomplished by using a set of tokens that are handled by the first level of the compositing tree in order to guarantee consistency. 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 is relative to the decomposition shown in Figure 15.*