A distributed-data implementation of the perspective shear-warp volume rendering algorithm for visualisation of large astronomical cubes

Brett Beeson, [1] David G. Barnes, [2] and Paul D. Bourke [1]

[1] Centre for Astrophysics and Supercomputing, Swinburne University of Technology, PO Box 218, Hawthorn, Australia, 3122

[2] School of Physics, The University of Melbourne, Parkville, Australia 3010
barnesd@unimelb.edu.au

Abstract

We describe the first distributed-data implementation of the perspective shear-warp volume rendering algorithm, and explore its applications to large astronomical data cubes and simulation realisations. Our system distributes sub-volumes of 3-dimensional images to leaf nodes of a Beowulf-class cluster, where the rendering takes place. Junction nodes composite the sub-volume renderings together and pass the combined images upwards for further compositing or display. We demonstrate that our system out-performs other software solutions, and can render a "worst-case" $512 \times 512 \times 512$ data volume in less than four seconds using 16 rendering and 15 compositing nodes. Our system also performs very well compared to much more expensive hardware systems. With appropriate commodity hardware, such as Swinburne's Virtual Reality Theatre or a 3Dlabs Wildcat graphics card, stereoscopic display is possible.

Keywords: methods: data analysis — techniques: image processing — surveys

1 Introduction

Astronomers, by virtue of the software provided to them for display and analysis, are ordinarily restricted to displaying two dimensional slices of data extracted parallel to one of the fundamental axes of their dataset. Some advanced applications exist, such as the `kpvslice` application in the Karma suite of visualisation tools (Gooch 1995), which provides the facility to display non-axial (and indeed non-planar) slices through volumetric data. Similar tasks are available in some radio astronomy reduction packages (*e.g.* `velplot` in Miriad). However, the dominant representations of volumetric data adopted for analysis or publication are two-dimensional axial slices (often with information along the non-displayed axes collapsed by some statistical operation — a *moment* map) and one-dimensional profiles (*e.g.* spectra).

Volume rendering (hereafter VR; Drebin et al. 1988) is an advanced technique for visualising volumetric datasets, wherein rays are cast through the data volume to generate a projected view. VR is useful for data with poorly defined surfaces such as astronomical data because in general it shows integrated properties of the data, and enables arbitary projections of the data into the displayed image plane. In many cases, such non-axial projections can enable the detection of new structure and relationships in complex multi-dimensional datasets, which are otherwise not visible in axial slices, or are concealed or washed out by statistical moment operations. For example, VR has been shown to be exceptionally useful for inspecting interferometric radio telescope images, especially as a tool to disentangle the complicated gas kinematics in disturbed galactic disks (Oosterloo 1995). Furthermore, interactive rendering where the volume can be manipulated (*e.g.* rotated, translated or magnified in

the viewing space) in near real-time, or where the transfer function controlling the mapping of data values to colours or opacities can be modified, can provide a substantially improved perception of structure in even quite noisy data.

Modern imaging systems, such as radio telescopes, can produce images having upwards of 100 million voxels. For smaller images, *e.g.* a single HIPASS cube (Barnes et al. 2001) covering $\sim 50\Box^\circ$ and having dimensions $170 \times 160 \times 1024$ voxels, a VR application such as KARMA's `xray` is satisfactory when running on a workstation with a few hundred Megabytes (MB) of memory. A present-day CPU can render of order 8 million voxels (Mvox) per second, so one can expect and indeed achieve, frame rates of order 0.3 frames per second (fps) using `xray` to render a 28 Mvox HIPASS cube. However, for many other cases, VR lies squarely in the domain of high performance computing (HPC). For example, the entire HIPASS dataset computed as a single data cube in the Zenithal Equal Area projection (ZEA; Calabretta & Greisen 2002) will comprise some 6×10^9 voxels! To volume render such an image requires of order six Gigabytes (GB) of memory (using only 8 bits per voxel). Even if such a machine were available, each frame would take at least 10 min to compute.

Direct images are not the only candidates for VR. Survey projects, for example the Sloan Digital Sky Survey (SDSS; York et al. 2000, Stoughton et al. 2002), now routinely collect tens of parameters for hundreds of millions of objects. For such databases, traditional visualisations such as two-dimensional scatter plots can and should be augmented with more sophisticated visualisation-aided data mining techniques. While plotting $\sim 10^9$ individual points in a three-dimensional phase space and then projecting to a particular point of view is a formidable task for single modern CPU — even one assisted with a geometry and transform hardware — if instead the data points are first gridded into a coarse volumetric dataset, having say 256 cells on each of three axes, the resultant data cube is of modest enough size for a VR algorithm to be applied and the data manipulated in real time. Some systems, especially those associated with virtual observatory endeavours, are pursuing this approach. For example, the DATOZ2K database system (Ortiz 2003) has a facility to generate simple VR visualisations of gridded catalogue data and present them to the user in a web browser.

VR is computationally expensive because the generation of a single projected view requires the consideration of all voxels in the data source; the display of two-dimensional slices generally involves less than one per cent of the voxel data. Fast and cheap hardware solutions do exist in the form of mass-market computer graphics cards. Their *texture memory* can be filled with slices extracted from the dataset, and then the geometry, transform and blending features of the card can be used to composite these textures into a projected view of the volume. However, this approach is severely limited by the memory available on present-day graphics cards (typically $\leq$ 128 MB), and an inflexible (hardware-coded) blending function.

Software algorithms on the other hand, are free to use main memory (typically $\geq$ 1 GB), and can give more extensive coverage of the domain of blending functions. Several algorithms have been developed for fast VR, and we have chosen the fast and efficient *shear-warp (S-W) factorisation*. A few parallel implementations of the S-W factorisation already exist (*e.g.* VOLPACK [Lacroute & Levoy 1994]; the National Center for Atmospheric Research's VOLSH; VIRVO [Schulze & Lang 2002]), but none distribute the data — all nodes "know" all of the data. In practice this limits these systems to rendering data volumes which, in their entirety, fit in a single node's memory space. As typical datasets from astronomical surveys now exceed one GB and are growing faster than the memory of commodity workstations, we explore the first implementation of the S-W algorithm for *distributed data*. By developing a VR application which runs on the nodes of a Beowulf-type cluster (Sterling et al. 1995), we benefit in two distinct ways, *viz.*

1. more processing resources are brought to bear on the problem, thereby improving minimum rendering time, and
2. more memory resources are made available, thereby enabling larger datasets to be rendered.

We have based our work on the VIRVO code,[1] described in Schulze & Lang (2002) and generously provided to us by Juergen Schulze. From VIRVO we use the rendering core to compute volume

[1]`http://www.hlrs.de/organization/vis/people/schulze/virvo/`

renderings of subsets of the data, modified by us to support the associative operator necessary for distributed data rendering. The remainder of the system — support for FITS-format data, the data division strategy and implementation, the correct compositing of individually rendered images, the design and implementation of the parallel, multiple-node distributed rendering tree, the selection and use of suitable compression techniques at different points in the system, and the front-end control and display software — is entirely new work.

We commence this paper with a brief review of volume rendering in Section 2. We describe the extension to distributed data rendering in Section 3, including some remarks on data division and optimisation strategies. In Section 4 we describe the essential features of the user interface to the software we have developed. We characterise its performance and scalability in Section 5, and finally we provide some sample applications in Section 6.

2 Volume Rendering

There are two distinct operations fundamental to VR which we now describe. Firstly, a VR operator is required which, given a set of voxels ordered back to front, will produce an integrated quantity representative of that set of voxels. Secondly, an efficient method of calculating lines of sight through the data volume, and therefore providing sets of voxels ordered back to front, is required.

2.1 The volume rendering and compositing operator

A volume is rendered by mapping each scalar voxel to a colour and opacity (see below) and accumulating integrated colour and opacity values along multiple conceptual viewing rays through the volume. To enable distributed-data VR, we need to ensure that sub-volumes of the data may be volume rendered independently and then *composited* together to produce the same result as if the entire volume had been rendered at once. We consider rendering to be the operation of producing a single output image from multiple input voxels, and compositing to the be operation of producing a single output image from multiple input images. The same operator is used for both, and must be *associative*. Note that not all voxel or pixel compositing operators are associative, for example the commonly used blending function of OpenGL[2] is not. In a now classic paper, Porter & Duff (1984) present a number of operators suitable for compositing separately rendered images, and derive the over operator, so named for the placement of a rendered foreground image over a rendered background image. We have chosen to use the over operator as it is associative, and suitable for use in both VR *and* compositing. We now briefly review this operator, and direct the reader to Blinn (1994) for further details.

We define opacity (α) in the interval $[0, 1]$ with $\alpha = 0$ and $\alpha = 1$ representing completely transparent and completely opaque voxels, respectively. Consider first the operation of combining a foreground pixel ($\mathcal{F}$ — a vector of red, green and blue colour components) with opacity $\alpha_{\mathcal{F}}$, with a background pixel ($\mathcal{B}$). The output pixel ($\mathcal{O}$) is simply

$$\mathcal{O} = \alpha_{\mathcal{F}}\mathcal{F} + (1 - \alpha_{\mathcal{F}})\mathcal{B}, \tag{1}$$

evaluated independently for the three colour components. Equation 1 — the *painter's equation* — is *not* associative. This is easily seen as there is no reference to the opacity of the background pixel.

We want a VR and compositing operator, &, such that for background, middle-distance ($\mathcal{M}$) and foreground voxels,

$$(\mathcal{B}\&\mathcal{M})\,\&\mathcal{F} = \mathcal{B}\&\,(\mathcal{M}\&\mathcal{F})\,. \tag{2}$$

To find the operator &, we set an intermediate image, $\mathcal{I}$, to be the composition of the middle-distance and foreground voxels, *i.e.*

$$\mathcal{I} = \mathcal{M}\&\mathcal{F}, \tag{3}$$

[2] `http://www.opengl.org/`

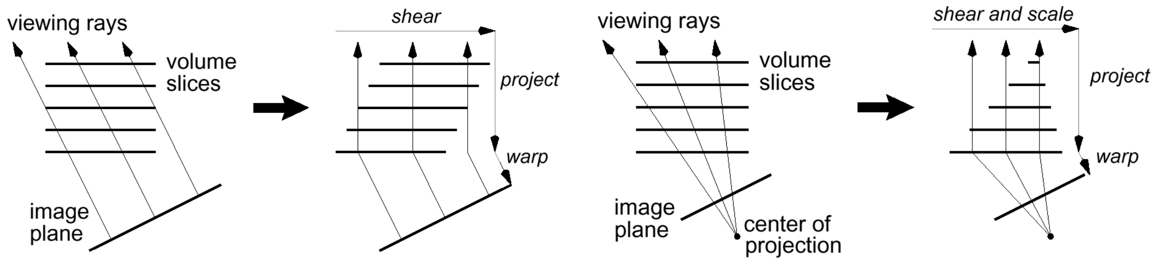

Figure 1: The shear-warp for parallel (left) and perspective (right) projections. Figure credit: P. Lacroute.

substitute in the painter's equation (it must still hold in the case of a completely opaque background voxel), and evaluate $\mathcal{I}$. We find that:

$$\alpha_{\mathcal{I}} = (1 - \alpha_{\mathcal{F}})\,\alpha_{\mathcal{M}} + \alpha_{\mathcal{F}} \quad (4)$$

$$\tilde{\mathcal{I}} = (1 - \alpha_{\mathcal{F}})\,\tilde{\mathcal{M}} + \tilde{\mathcal{F}}, \quad (5)$$

where the tilde above the voxels implies pre-multiplication by the opacity, *i.e.* $\tilde{\mathcal{X}} \equiv \alpha_{\mathcal{X}}\mathcal{X}$. Equations 4 and 5 define the over operator which we adopt for VR and compositing. Note that for $\alpha_{\mathcal{M}} = 1$, these equations reduce to the painter's equation. While the over operator is associative, it is *not* commutative, so we must preserve the ordering of voxels during VR and compositing.

2.2 The shear-warp and perspective shear-warp techniques

There are two distinct approaches to applying a VR operator to a volume of data:

1. A *pixel-order* renderer (also referred to as a *ray-caster*) loops over all of the pixels in the projected output image. For each pixel, a list of contributing voxels is compiled and sorted according to distance from the image plane, and then the VR operator is applied working from the back to the front of the list. Pixel-order rendering is suitable for associative but non-commutative VR operators, and is eminently suited to parallelisation by scan-line subdivision.

2. A *voxel-order* renderer (also referred to as a *splatter*) loops through the data volume, and projects each voxel onto the image plane. It lends itself well to commutative operators, such as *max* (maximum value), *sum* (summed value), etc., but in general will be an extremely inefficient procedure for any non-commutative VR operator (*e.g.* an opacity-dependent operator).

For a parallelised, distributed-data renderer, we note that pixel-order rendering is not suitable, because it would require all nodes of the rendering cluster to have access to all of the data. Somewhat paradoxically, voxel-order rendering is also not satisfactory, since it does not efficiently support non-commutative (*i.e.* ordered) operators which we have already established are required for piecewise rendering and compositing of a large data volume! Fortunately, an elegant and efficient technique — to some extent a halfway point between pixel- and voxel-order rendering — exists: the shear-warp factorisation.

The S-W factorisation was first applied to volume rendering by Lacroute & Levoy (1994). This algorithm *shears* the volume space and *warps* the image space, so that viewing rays are parallel to a fundamental axis of the data volume — see Figure 1 (left). In the transformed space, voxels and pixels align, and a VR system can traverse the volume *and* the image in order. Furthermore, the trajectories of individual viewing rays no longer need to be calculated, saving many costly transcendental calculations. The S-W is easily extended to provide perspective by including a distance-dependent scaling in the transform — see Figure 1 (right).

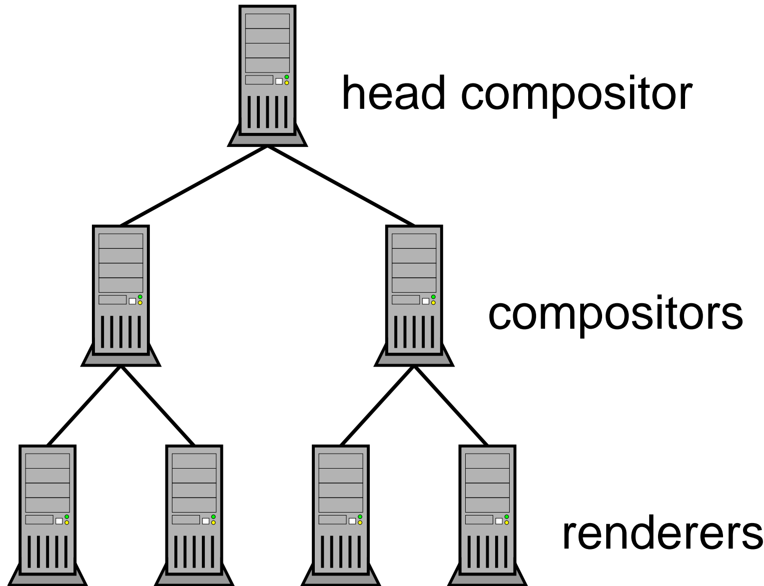

Figure 2: Example rendering tree with branching factor $b = 2$ and number of levels $n = 3$.

3 Distributed Data Volume Rendering

Our distributed data volume renderer is constructed using the S-W algorithm (with or without perspective) and the over operator:

1. The data volume is divided into two or more sub-volumes, each a three-dimensional array of voxels.
2. The S-W algorithm is used to render (with the over operator) each sub-volume independently with the same camera and projected onto the same image plane.
3. The over operator is then used again, this time to composite the rendered images, proceeding from back to front according to the position of the sub-volumes in the original volume.

The associativity of the over operator, its use for both rendering *and* compositing, and the correct sorting of the rendered images prior to compositing, ensure that the final composited image is identical to the output of a single-pass renderer.

3.1 The rendering tree

We use a *rendering tree* with a configurable branching factor b, similar to the scheme used in VFleet,[3] except that VFleet is a parallel renderer requiring all rendering nodes have access to all of the data. The rendering tree contains *compositors* (branch nodes) and *renderers* (leaf nodes). For an n-level tree, there are b^{n-1} renderers and $1 + b + b^2 + \cdots + b^{n-2}$ compositors. A simple example rendering tree with $b = 2$ and $n = 3$ is shown in Figure 2. The connections between nodes represent *socket* connections.

The parameters of the rendering tree can be tuned to suit various configurations of processor speed, physical network topology and network bandwidth availability. For slow processors connected by a fast network, a low branching factor shares the compositing amongst many nodes, the extreme case being a binary tree with only one more renderer than compositors. Conversely, a tree of fast processors connected by a slower network will benefit from a higher branching factor which places more load on

[3] http://www.psc.edu/Packages/VFleet_Home/

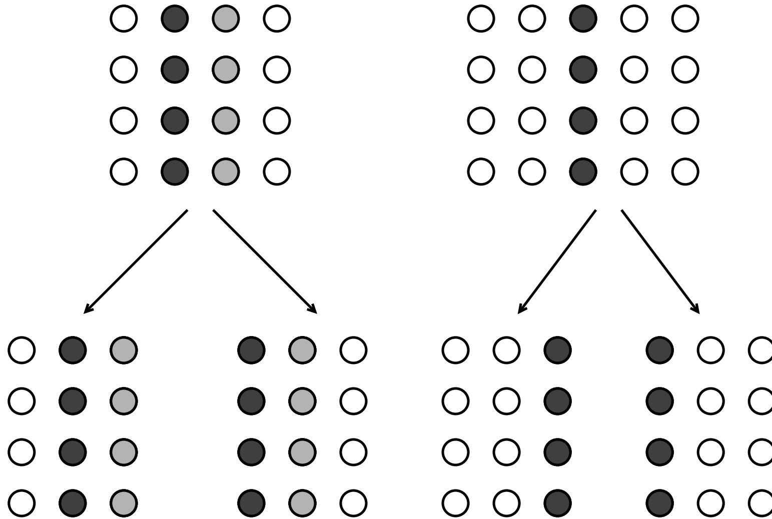

Figure 3: Volume division: dividing along an even-length axis (left) and an odd-length axis (right). The circles represent individual samples (*i.e.* voxels) in the data volume, which extends into the page.

fewer compositors, the extreme case here being a single compositor with b renderers. For a shared memory machine, where inter-node bandwidth can exceed one GByte/s with sub-μs latency, the best rendering tree will likely be one which utilises all available processors. We discuss performance further in Section 5.

3.2 Data division

The rendering tree structure determines to a large part how the volume data should be divided amongst the rendering nodes. We adopt an iterative division scheme which works in the following way. The head compositor node (the top of the tree in Figure 2), divides the entire data volume into b pieces which it passes to its b children. If the children are themselves compositors, then they further divide their own sub-volumes along the longest axis into b pieces for their b children. Note that the head compositor can — but need not necessarily — send "physical" arrays of data to the children. The data volume can be subdivided in advance if the rendering tree structure is known, and each sub-volume stored on network disk accessible to the rendering nodes, or even disk local to each node for even faster start-up. The division scheme produces convex, adjacent sub-volumes, thereby ensuring correct ordering is possible and yielding a balanced rendering tree. This strategy could be modified for use on a cluster with "fast" and "slow" nodes, but care would need to be taken to ensure that a unique back-to-front order remains. Note that the volume division must ensure sufficient information is available to each node to correctly reproduce edge values. To this end, we divide volumes as depicted in Figure 3, such that sub-volumes are always the same size, and share at least one plane of voxels.

3.3 Compositing

With the rendering tree installed and configured, VR can proceed. The requested viewing angle and image plane are parameterised and passed all the way down the rendering tree to the renderers. They apply an appropriate shear to their (sub-volume) of data, possibly applying a perspective scaling, use the over operator to generate a projected, volume rendered image and then warp this into the required image plane. The rendered images are then sent progressively up the tree where compositors use the same over operator to combine the b images of adjacent sub-volumes rendered (or composited) by their children, using ordering information from their positions. The head compositor node produces the final rendered image.

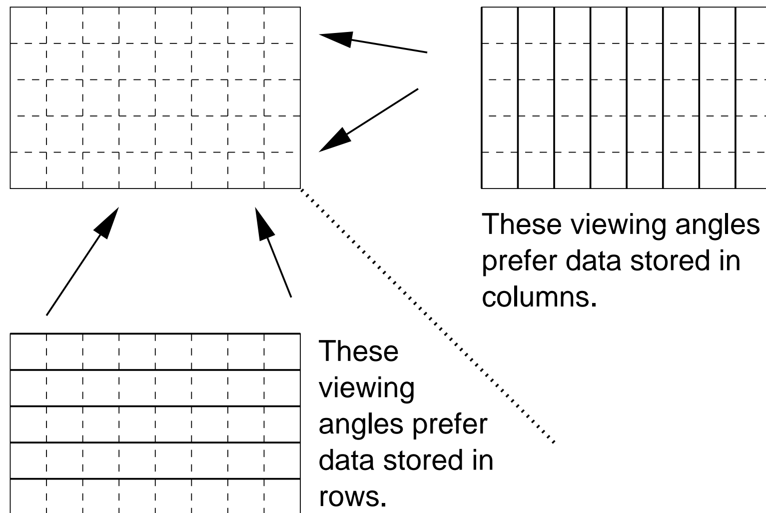

Figure 4: The effect of viewing angle on the preferred data storage scheme, illustrated for an axial slice through a volume.

3.4 Optimisations

Dynamic range compression. On 32-bit architectures, a single floating point value occupies 4 bytes. This provides a huge dynamic range (typically of order 10^{38}) which is rarely, if ever, required. This is especially true in the context of visualisation, where on a 24-bit display there are (nominally) 16 M colours available,[4] of which, under the very best conditions, the human eye can distinguish perhaps up to 1 M (Halsey & Chapanis 1951). To save a factor of four in memory requirements (and a similar factor in the number of processor cycles needed to shear data planes), it is straightforward to reduce the dynamic range to 65536 or even 256 by mapping the input floating point data to 16-bit or 8-bit integer values. Provided a careful choice of mapping is made, this measure will only infrequently compromise the output of VR.

Shear-warp projections. The efficiency of the shear-warp algorithm is mostly due to the traversal of the volume data in order. This depends on the volume data being stored such that the data for each sheared plane is stored in a single block of physical memory. For a three-dimensional volume of data, there are three orthogonal sets of planes which might be sheared, defined as the planes perpendicular to the first axis, the second and the third. In the non-perspective S-W, every viewing angle can be identified with one of these sets which is optimal for efficient rendering. Figure 4 should help clarify this: for viewing angles from the bottom (or top) of the figure, the S-W algorithm can be applied more efficiently with the data stored in rows, while for viewing angles from the right (or left) of the figure, the data is best stored in columns. This voxel set selection as described has the basic function of keeping the shear "rate" to less than one pixel per plane (at 45 degrees it is equal to one pixel per plane). In terms of efficiency this reduces memory requirements during the shear and reduces the total extent of the sheared axis (thereby reducing the intermediate rendered image size). However it also improves the correctness of the rendering by selecting against lines-of-sight which go through more than two pixels per sheared plane.

Most implementations of the S-W algorithm store the volume in one order, and re-order the data when the viewing angle demands a new storage order. As real-time re-ordering is not feasible for sub-volumes larger than a few Mvox, our implementation stores the three alternately-ordered copies of the sub-volume data on each rendering node. While this triples memory requirements, it can substantially

[4] Although many fewer than 16 M colours are produced in practice by computer display systems as they fail to produce fully saturated colours, and ambient light can substantially reduce contrast.

improve the interactive response of the system during rapid changes to the viewing angle.[5]

Window-encoding images. Images are sent from renderers to compositors and from compositors to compositors, quickly consuming network bandwidth. To improve transfer speed, each image can be window-encoded before it is sent upwards to a compositor. This involves computing the bounding box of non-blank pixels and only sending this sub-image. We do this by projecting each corner of the volume into the image plane. Very often the sub-volume rendered by a renderer or composited by a compositor will only project to a small part of the final image, so substantial savings can be made using window-encoding.

Minimal compositing. The brute force method of compositing is expensive since every pixel in the b input images must be considered. Performance can (obviously) be improved considerably by only compositing the non-blank image sections. This is accomplished by using the window-encoding information *already computed* for the network transfer of images, and choosing not to decode the full-size images. In this way the rendering tree composites only sub-images of sub-volumes.

4 Display and Control

4.1 The user interface

Volume rendering is often used to *explore* data — the user will modify the transfer function and move the viewpoint in order to identify and visualise different features of the data. A user interface is required, which we have chosen to de-couple from the rendering tree for the very important reason that the user may not be in the same physical location as the cluster that is available to render their data (see Figure 5). Additionally, the user's workstation may well be slow compared to available cluster nodes, and so computation on the workstation is kept to the minimum necessary to display the rendered image and to control rendering parameters such as the transfer function and viewing angle. We also note that separating function and interface allows future interfaces to re-use existing functional codes.

As the network connection between the user's workstation and the VR cluster may be slow compared to the cluster interconnect, it is prudent to run-length encode the image produced by the head compositor before it is sent to the user interface for display. Run-length encoding (RLE) entails replacing runs of repeated data values with the data value (or values) and a repeat count. For example, the sequence `kavababababyt` might be replaced by `kavab+3yt`. Since a final rendered image might have large but irregular patches of black (*i.e.* pixels whose lines-of-sight do not penetrate any of the data volume), RLE offers a good compromise between compression speed and compressed image size (*i.e.* network transfer time). The only other network traffic sent between the interface and the rendering tree is limited to a small set of instructions issued in response to user activity, such as `load data` and `rotate`.

The rendered image is displayed in the main window of the interface, which is written in TCL script using the TK widgets. TCL and TK were chosen for their availability on a wide range of systems (*e.g.* Unix, Microsoft Windows, Mac OS X), and also for the ability to use TCL scripting to produce movies following calculated "flight paths", or sequences of volume renderings of one dataset after another. TCL is actually quite fast for an interpreted language and our choice of TCL does not impact at all on rendering frame rates.

In the interface, the user can drag the mouse to rotate the volume, or rather, to move the camera around the volume. This system of direct volume movement is far more intuitive than setting camera angles manually as is required in KARMA's XRAY, but calls for framerates of a few frames per second to be useable. Our distributed system can meet this requirement for relatively large volumes (see Section 5). Our combined system of distributed data volume rendering and the TCL user interface is christened DVR, standing for "distributed volume renderer."

[5]For cases where memory resources are precious, this optimisation could in principle be switched off.

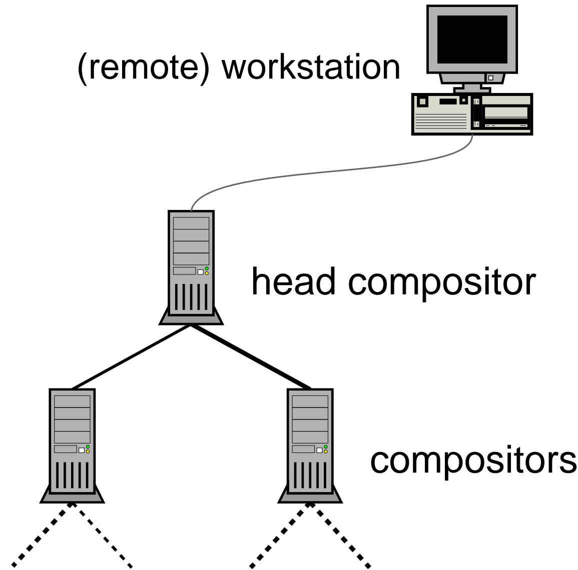

Figure 5: The top of a rendering tree running under the control of a (possibly remote) workstation. Typically the cluster nodes will interconnect via a fast, low-latency network, while the workstation will communicate (only) with the head compositor via standard ethernet. The workstation may be a different architecture to the cluster nodes, which themselves may be a heterogenous collection.

4.2 Perspective and stereo rendering

The camera control in the user interface allows the user to switch on perspective rendering. Perspective is generally not necessary for middle and distance views, but becomes essential for nearby views and views from within the volume itself. The perspective shear-warp projection (see Figure 1) could more accurately be called the scaled shear-warp projection, as the only substantial difference from the parallel shear-warp projection is in the application of a distance-dependent scale factor during the shear. In practice, the scaled shear-warp is slower than the parallel shear-warp because of the additional resampling of the volume data. However, the scaling operation produces an intermediate image, whose resolution can be chosen to provide an accurate rendering, or a faster, coarser rendering. Consequently, when a perspective render is selected, our system allows the image quality to be reduced while the volume or camera is in motion to provide higher frame rates. When the user stops manipulating the volume, a higher fidelity image can be rendered.

With appropriate hardware DVR can produce and display stereoscopic volume renderings. We use a non-symmetric camera frustrum for off-axis stereoscopic rendering, which produces coincident projection planes for both eyes.[6] The two views are rendered independently by DVR, one after the other, and the user interface combines the images for display, either on a 120 Hz frame-sequential stereo system with active LCD shutter glasses or a dual display passive stereo system viewed with polaroid glasses. We note that perspective rendering is essential for meaningful stereoscopic display.

4.3 The transfer function

The most important tool provided to the user is the transfer function editor, which controls the mapping from scalar voxel values (S) to colour ($\mathcal{F}$, a vector of red, green and blue colour components) and opacity ($\alpha_{\mathcal{F}}$):

$$S\left(x,y,z\right) \rightarrow \left\{\mathcal{F}\left(x,y,z\right), \alpha_{\mathcal{F}}\left(x,y,z\right)\right\} \tag{6}$$

[6]An introduction to the subtleties of stereographics can be found at http://astronomy.swin.edu.au/~pbourke/stereographics/stereorender/

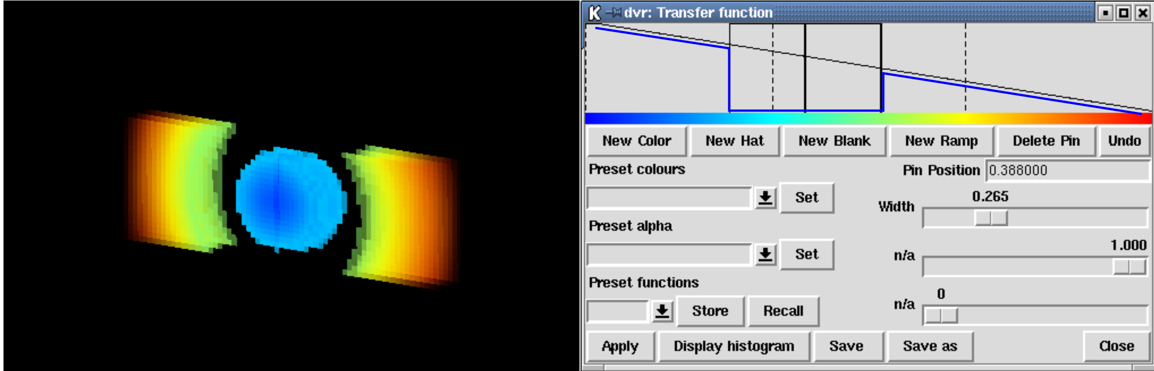

Figure 6: Example rendering (left) of a synthetic dataset, and the transfer function used (right), showing the combination of a ramp and blank. The thick blue line indicates the combined effect, with the blank taking precedence over the ramp.

Our implementation of a transfer function editor – shown in Figure 6 with a sample rendering – is now described. Certainly many other schemes are imaginable and could be implemented to replace the existing one. The X-axis of the transfer function graph extends over the the scalar data domain, which for 8-bit data is $[0, 255]$. A histogram of the voxel values can be displayed in the background of the transfer function graph to assist with interpretation and construction of the function.

To control opacity, or the "see-throughness" of the data, the user is able to select and place various *alpha pins* in the top panel of the transfer function editor. In this area, the Y-axis represents opacity in the range $[0, 1]$. The alpha pins include: straight lines ("ramps") whose slope and position can be controlled; trapezoidal functions ("hats") whose height, width and edge slope can be controlled and whose special cases include the tophat and triangle functions; and blanks whose width and position can be controlled. Where multiple pins are used and overlap, the maximum opacity is adopted, except that blanks, which make voxels totally transparent, have precedence over all other pins. In Figure 6, the effective opacity function as a result of combining a ramp and blank is marked in blue.

The coloured bar along the X-axis shows the mapping $S(x, y, z) \rightarrow \mathcal{F}(x, y, z)$, which is modified using *colour pins*, shown as vertical dashed lines. Each colour pin defines a colour using red, green and blue values in the range $[0, 255]$, and colours are linearly interpolated between the pins. The colour pins can be moved, effectively compressing or extending the gradient between adjacent pins. Colour pins can also be added or removed, and several popular colourmaps are provided with pre-configured pin colours and positions. A particularly effective way to use the colour and alpha pins is to provide strong colour gradients and moderate opacity over the "interesting" (signal) part of the voxel value domain, and gradual gradients and low or zero opacities over the remainder of the domain (typically the noise).

In Figures 6 and 7, we show the effect of two different transfer functions on a synthetic dataset. The data volume is a rectangular prism, with scalar value zero at its centre, increasing linearly with radius to 255 at the centre of the faces perpendicular to its longest axis. The transfer function of Figure 6 comprises a ramp which sets scalar values of zero to be completely opaque, scalar values of 255 to be completely transparent, and intermediate scalar values to be partially transparent. In addition, a blank is used which overrides the ramp and renders scalar values 64 to 128 to be completely transparent. The resultant volumetric render shows the highly opaque centre of the volume, and the transparent outer part of the volume. In Figure 7, the transfer function is a trapezoidal function centred on scalar value 164, with a narrow top and wide base. This has the effect that only scalar values in the range 128–200 have any appreciable opacity, and only a shell of the volume data is visible in the rendering.

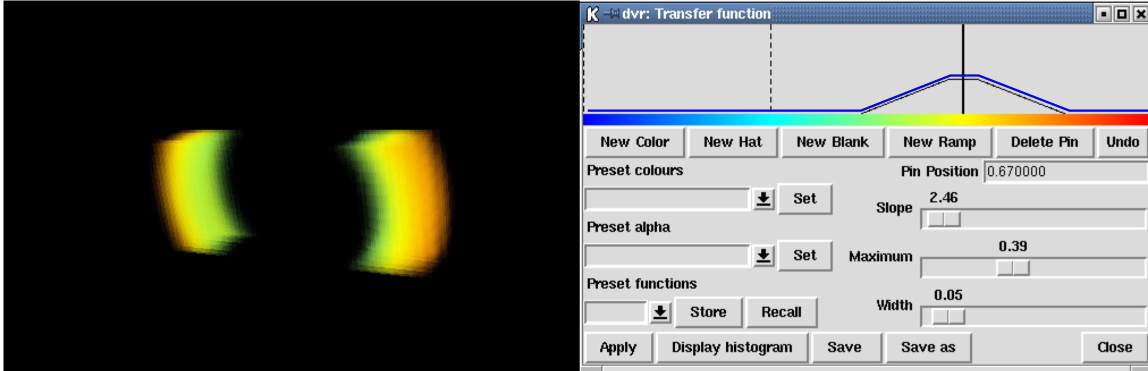

Figure 7: Example rendering (left) of a synthetic dataset, and the transfer function (right) which in this case is a simple trapezoid function. The thick blue line indicates the nett opacity transfer function.

5 Performance

We remind the reader that this project was motivated by a present and perceived future need to render volumetric datasets larger than typical workstation memories, at interactive frame rates. We have described a technique which enables us to break apart the data volume into a number of smaller pieces which are rendered independently as overlapping images then composited together to produce the final view. We now consider the performance of our system, which can broadly be broken down into the following areas: single-processor rendering performance, network transfer (and compositing), and scalability. Since the controlling interface of DVR is written in TCL script, we were able to acquire accurate and repeatable measurements of the performance of DVR by writing and running a short script to load a particular volume, configure the viewport, and submit frames for rendering.

Single-processor rendering performance. A single 2 GHz Pentium 4 CPU can render at $\sim$ 7 Mvox s^{-1}. This is measured using our rendering core (*i.e.* the over operator) applied to a dataset where every voxel contributes to the output image. A volume such as this, containing no fully opaque or completely transparent voxels, is suited to performance testing because the time to render the volume will generally be independent of the viewing angle. Practical applications of VR to noisy astronomy datasets however will usually entail a transfer function which arranges for many fully opaque or completely transparent voxels, in which case the core rendering speed may be substantially improved.

Network transfer and compositing. For distributed data rendering, image transfer time may contribute signficantly to the overall rendering time, and will depend on the network structure. For our tests we used the Swinburne Centre for Astrophysics and Supercomputing facility, which is a Beowulf-class cluster of Intel architecture machines running Linux. The cluster network is 1000 Mb s^{-1} ethernet ("Gigabit") and the cluster is connected to the front-end interface machine by standard 100 Mb s^{-1} ethernet ("100 Meg").

For our relatively fast network, Figure 8 shows that the application of window-length encoding to intermediate images (Section 3.4) yields unmeasurable image propagation times (*i.e.* less than one ms). Without compression, it would take $\sim$ 10 ms to send the intermediate images (500×500 pixels) over Gigabit. The window-length encoding and decoding operations take $\sim$ 1 ms each. Compositing, including the implicit handling of the window-length encoded images, takes around 10 ms per *input* intermediate image with our optimisations. Speeds are dataset and viewpoint dependent: views within the volume produce large images which cannot be window encoded while images with contiguous colour runs (*e.g.* distant views of the volume) are efficiently run-length encoded.

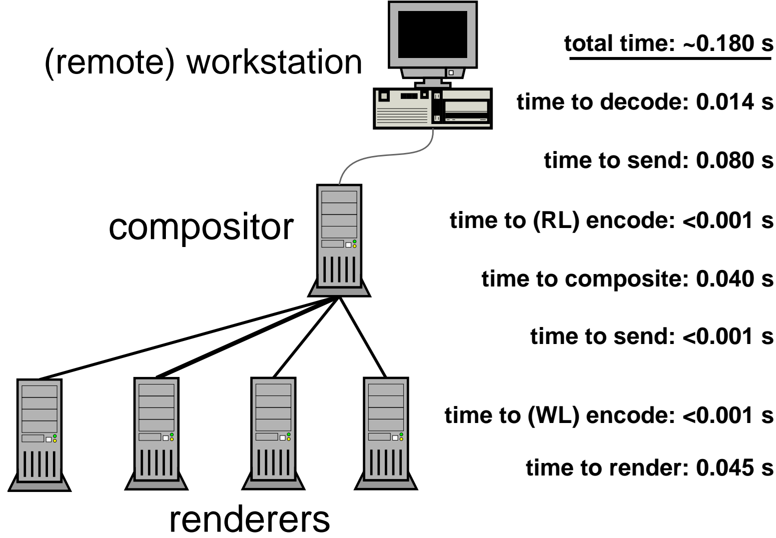

Figure 8: Breakdown of approximate time accrued in rendering, transferring and compositing a 500×500 image and delivering it to the display node. A fast cluster network is assumed, such that the main contributor to the rendering time is the transfer of the final rendered image to the display node over a standard network link.

Scalability. We can estimate the largest volume that can be rendered with N processors, given a base voxel rendering rate of $R_{\rm vox}$ in voxels per second. Ignoring parallelisation costs (*e.g.* the increase in network traffic and in the number of compositing processes with N), a cubic data volume of sidelength l can be rendered at a rate of r frames per second according to:

$$l^3 = \frac{R_{\rm vox} N}{r} \tag{7}$$

For our measured $R_{\rm vox} \simeq 7$ Mvox s^{-1}, a required rate of five frames per second, and 16 *rendering* processors, we deduce that a volume of dimensions $280 \times 280 \times 280$ can be rendered interactively. For a binary tree, a total of 31 processors would be required (16 renderers and 15 compositors) and we point out that this could easily be accommodated on the relatively commonplace 16-node dual processor cluster. Even with our distributed system, a Gvox volume (*i.e.* $1024 \times 1024 \times 1024$ voxels) is still expected to require ~ 150 rendering processors to produce frames at the rate of one per second. Real-life frame rates are likely to be much better than this though because often only a small fraction of voxels are unblanked and contribute to the VR transfer function.

To verify the scalability of our system, we generated cubic data volumes between 64^3 and 1024^3 in size and rendered them using between three and 31 nodes. The data volumes were filled with random data, a flat transfer function was applied and a camera path was selected so that the rendered image completely filled the 512×512 pixel output image for all viewing angles. These conditions ensure consistent *worst case* performance because *all* voxels must be rendered, and no encoding or compositing savings are possible. Table 1 shows the resultant frame rendering times averaged over ten frames, as well as an indication of the *parallel efficiency* as a function of volume size. Parallel efficiency measures the rendering performance *per node* for the 31-node case as a percentage of that for the 3-node case. Table 1 shows that for small volumes, parallel rendering is very inefficient and not worthwhile, but for volumes upwards of 256^3 voxels parallel rendering offers an excellent performance gain with efficiencies of $\sim 50\%$. For a binary rendering tree, the maximum parallel efficiency is $\sim 75\%$ (rather than 100%) because for small trees two-thirds of the nodes are rendering nodes while for large trees only half of the available nodes will be rendering with the remainder attending to the generally less demanding task of compositing. The 1024^3 volume shows an unusually high efficiency simply because there is insufficient memory in the small rendering tree configurations to keep even the sub-

Volume size	Number of nodes				Parallel efficiency
	3	7	15	31	for 31 c.f. 3 nodes
64^3	0.20	0.18	0.28	0.41	5%
128^3	0.46	0.32	0.30	0.40	11%
256^3	2.8	1.5	0.81	0.56	48%
512^3	21	11	5.54	3.9	52%
1024^3	290	85	43	21	134%

Table 1: Measured total rendering time in seconds and parallel efficiency for fully-sampled, cubic volumes on the Swinburne facility. The rendering tree is a binary tree in all cases (*e.g.* 7 nodes comprises one head compositor, two compositors and four renderers).

Rendered image size	Frame rendering time (sec)
100×100	0.32
256×256	0.36
512×512	0.44

Table 2: Time to render output frames of different sizes for an input volume of 128^3 rendered with a binary tree spanning seven nodes.

divided volume data in physical memory and so the low-node configurations suffer from expensive swapping to disk.

In Table 2 we briefly show the effect of rendered image size on rendering rate. Internal timings showed the rendering time itself to be steady at ~ 0.25 s per frame independent of the rendered image size. The gains for smaller images therefore arise almost exclusively from the encoding and compositing optimisations already discussed.

Finally we tested the ability of our system to handle a very large volume. We generated a filled volume of dimensions $2048 \times 2048 \times 2048$, and rendered it using a tree comprising one head compositor and 32 rendering nodes. Rendering the same camera path as for the above tests, ensuring that the 512×512 output image was fully sampled, yielded an average frame rendering time of 85 s. While this is not an interactive frame rate, it lies within a factor of two of our predicted rendering rate (Equation 7) and to our knowledge is the largest dataset volumetrically rendered by an otherwise interactive system. This challenging rendering task consumed > 800 MByte of memory on each of the 32 nodes.

External comparisons. Here we compare our distributed data volume rendering rate with recent results from the high performance computing scene. Snavely et al. (1999) present timings for the SAMPLERAY rendering code (based on MPIRE[7]) running on two different supercomputer architectures, the Cray T3E and the Tera MTA. They rendered a 256^3 sub-volume of the Visible Male dataset, from the Visible Human Project[8], using between one and sixteen processors of a shared memory system rather than a cluster. They rendered to output images 400×400 pixels in size. In Figure 9 we plot their timings against our most similar tests from Table 1, *i.e.* a filled 256^3 volume rendered to fill a 512×512 pixel output image. The comparison is extremely favourable to our system, despite the fact that tests on our system were deliberately configured to give worst-case values, and the obvious advantage of their shared memory system for extremely low latency interprocess communication,

The TERAVOXEL project at the California Institute of Technology[9] has as its goal the capture and visualisation of fluid volumes at up to 1024^3 volume elements per second. Using specialised volume rendering hardware — eight VOLUMEPRO 500 systems built by TeraRecon interconnected

[7]MPIRE — Massively Parallel Interactive Rendering Environment — is a distributed VR system available on Cray and SGI platforms with specialised hardware; `http://mpire.sdsc.edu`.
[8]`http://www.nim.nih.gov/research/visible/visible_human.html`
[9]`http://www.cacr.celtech.edu/projects/teravoxel/`

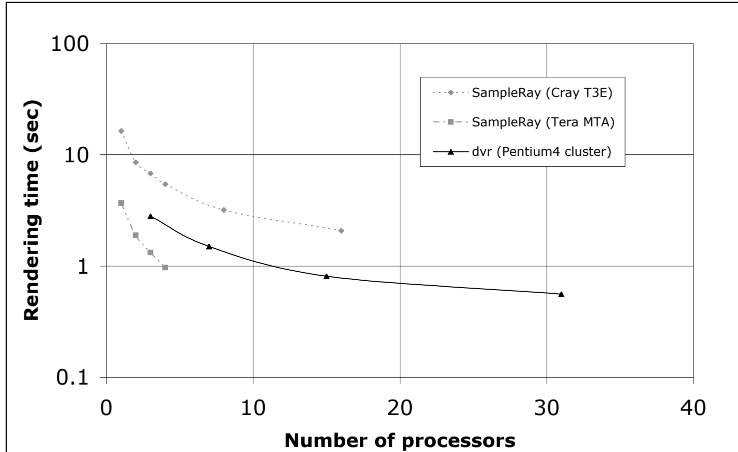

Figure 9: Rendering time comparison between the **SampleRay** renderer running on Cray T3E and Tera MTA systems, and our **dvr** renderer running on a cluster of Pentium 4 workstations. The **SampleRay** timings are from Snavely et al. (1999) for a 256^3 cutout of the Visible Male dataset (The Visible Human Project) rendered into a 400×400 output image; the **dvr** timings are taken from Table 1 for rendering a filled 256^3 volume into a 512×512 output image.

with HP-Compaq's SEPIA for hardware-based compositing — they have successfully rendered a 512^3 data volume at 24 frames per second — more than 100 times faster than our worst-case measurements for a 7-node rendering tree! This leaves no doubt about the merits of specialised hardware over general clusters for this kind of work, but other than being fiendishly expensive and singular in purpose, this hardware system, like most volume rendering systems that we know of including SAMPLERAY, is *not* a distributed data system — all nodes must store the entire dataset in memory. In practice, today's research groups seem eminently more likely to have access to Beowulf clusters than to facilities like the TERAVOXEL system, and so software implementations of VR such as ours which run on commodity hardware should remain useful for at least a few years.

6 Applications

The potential applications of DVR are many and varied. Here we give three examples drawn from theoretical and observational astronomy. Further to these examples, DVR has also been used to render multidimensional pulsar search and timing data collected at the Parkes 64 m radiotelescope and magnetic resonance images of the human brain. Any volumetric data can be visualised with DVR once it is converted to the appropriate, *simple* input format.

Spectral line data cubes. The work described in this paper was motivated by the need to visualise spectroscopic data acquired using the Multibeam facility at the CSIRO's Parkes radiotelescope. However, it might equally have been prompted by the need to display spectral line data from synthesis radiotelescopes such as the Australia Telescope Compact Array, or from integral field unit multi-object spectrographs which are becoming commonplace on the world's major optical telescopes. In each case, an intermediate product of the data reduction process is the *spectral line data cube*, a 3-dimensional volume of data whose axes are (normally) latitude and longitude on the sky, and one of frequency, wavelength or derived radial velocity. Spectral line data cubes typically comprise 10^7–10^9 voxels, and

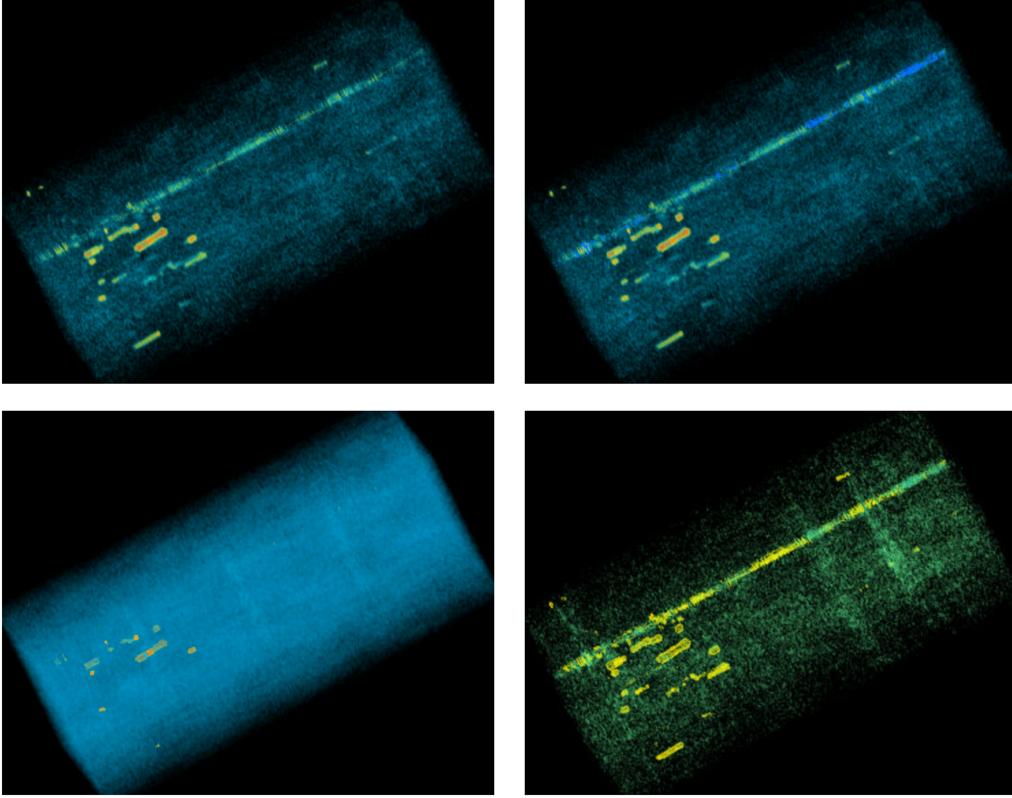

Figure 10: Volume renderings of the deep HI Fornax galaxy cluster cube showing the capacity of different transfer functions to emphasize different features. **Top-left**: simple ramp-function which applies close to zero opacity to the noise and complete opacity to the highest values reveals most of the spectral line sources in the data plus the strong continuum source Fornax A. **Top-right**: a top-hat of moderate opacity placed over strong negative values brings out the negative values present in the baseline ripple induced by Fornax A. **Bottom-left**: a top-hat of very low opacity placed over the noise illuminates the entire data volume and is complemented by a top-hat of very high opacity placed over only the highest voxel values, revealing the neutral hydrogen bright members of the cluster. **Bottom-right**: two top-hats, but with reduced opacity in the noise regime and a wider top-hat covering the source emission regime; the colourmap has also been modified.

so lend themselves well to distributed data volume rendering.

As an example, we present in Figure 10 a new volume rendering of a deep neutral Hydrogen (HI) emission image of the Fornax cluster of galaxies. The $148^2 \times 380$ voxel data set has been kindly provided to us by M. Waugh in advance of its publication. We show only one projection of the data but with different transfer functions to highlight different components of the data. In Figure 10, galaxies appear as "blobs" extending diagonally bottom-left to top-right which corresponds to the frequency or line-of-sight velocity axis of the data cube. The feature extending all the way along this axis is the radio continuum source Fornax A which induces baseline ripple in spectra taken in its vicinity. The two angular coordinates on the sky lie at right angles to this axis, *i.e.* diagonally bottom-right to top-left and into the page.

HI emission images of galaxy clusters such as Fornax are extremely sparse. That is, the overwhelming bulk of the voxels in the data cube are noise, and only a tiny fraction of the volume data contains astronomically interesting values. This is in stark contrast to HI emission images of our own Galaxy, its satellite Magellanic Clouds, and its population of discrete, high velocity clouds which can be found over most of the sky. Images of these features can be beautiful, but very complicated and difficult to interpret without advanced visualisation software. In Figure 11, we present a volume rendering of an HI emission image of the galaxy NGC 3109, including Galactic and high velocity gas. The data were

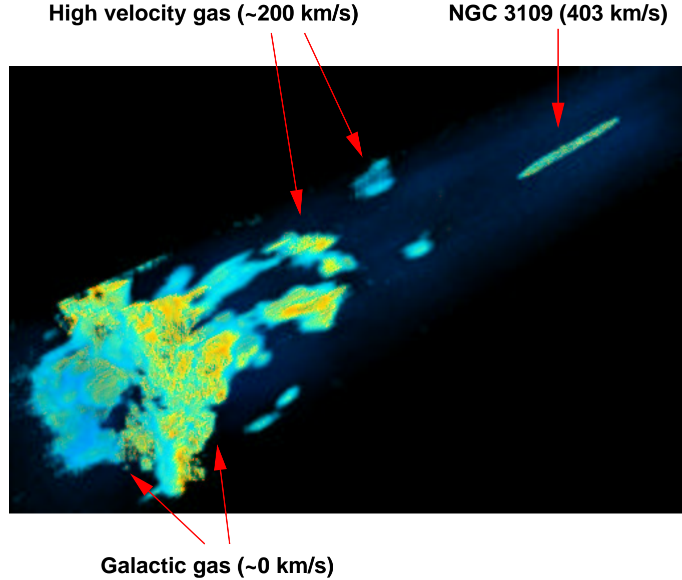

Figure 11: Perspective volume rendering of an HI emission image of the galaxy NGC 3109, Galactic gas, and the intervening (in velocity space) high velocity gas.

taken from Barnes and de Blok (2001), and the complex nature of the field is immediately evident in the volume rendering.

N-body cosmology. Traditionally, N-body data is visualised by individually projecting the N points to the 2-dimensional plane of the screen, and colouring the points according to some property other than position, *e.g.* mass or line-of-sight velocity. In such displays, foreground particles generally obscure background particles and integrated line-of-sight quantities are not easily assessed. To display the true volumetric nature of N-body realisations, a VR system such as DVR is needed. To this end, we obtained a single time-step realisation of the Universe generated by the multi-level adaptive particle mesh (MLAPM, Knebe et al. 2001), comprising some two million particles, each tagged with a measure of the local density. We gridded the sampled densities into a 256^3 volume and submitted the data to DVR for rendering. A script was used to control the camera movement, and the resultant movie (composited from 130 frames) is available in QuickTime format from `http://www.aus-vo.org/software.html`. Four frames from the animation are shown in Figure 12.

The application of VR, and specifically of DVR, to cosmological studies presents interesting possibilities for future work. For example, the periodic boundary conditions which constrain most N-body simulations allow the data to be translated within the bounding box of the simulation and wrapped from one edge to the opposite edge, to provide a different but equally valid realisation. With some careful thought, the shear-warp algorithm may lend itself to a modification whereby the shear is replaced with a shear-and-wrap (thence the "shear-wrap-warp" algorithm), such that in addition to controlling the view direction, the user is able to choose different translations of the simulation realisation within a VR environment. One possible implementation of this scheme within a *distributed-data* system like DVR would be to divide the data only along the axis nearest the view direction such that each node has a set of data spanning two axes of the volume.

N-body galaxy formation and evolution. As a second example of using VR to visualise the results of N-body simulations, we present the final time step in an interaction between the Milky Way galaxy and a satellite galaxy. A parallel tree smoothed particle hydrodynamic code (Kawata 1999) was used to simulate a point-source satellite galaxy inducing a high-latitude warp in the disk of the Milky Way galaxy (Kawata et al., *in prep.*). The simulation included 200000 halo particles,

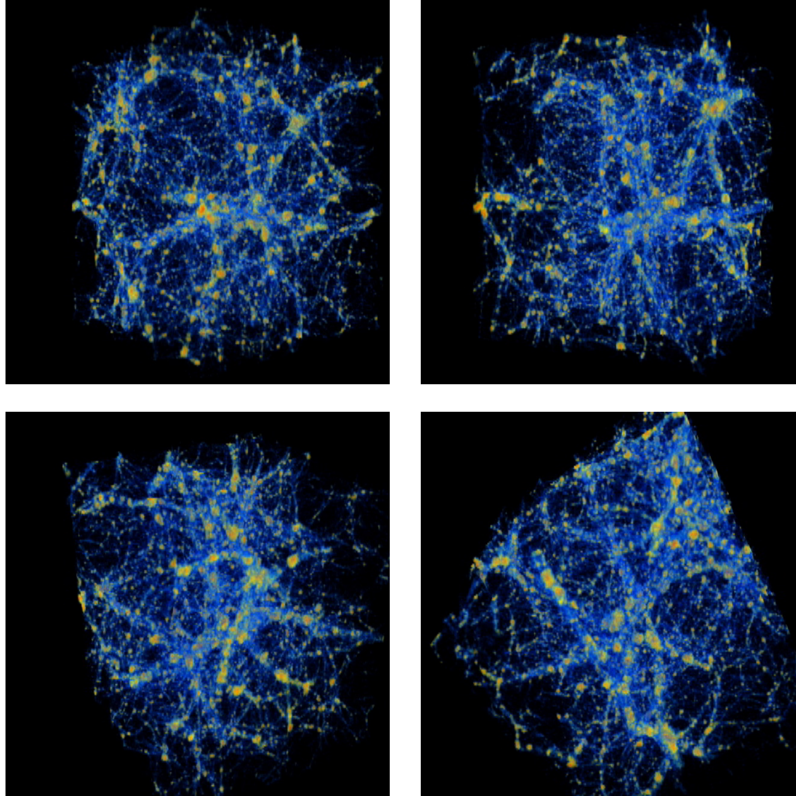

Figure 12: Four views of a single time-step realisation of the Universe generated by the MLAPM code (Knebe et al. 2001). Two million samples of the matter density in the Universe were smoothed into a 256^3 volume and rendered using DVR.

80000 disk particles and 20000 bulge particles. The bulge and disk particles were gridded into a 128^3 volume which was rendered using DVR. A 130-frame movie is available in QuickTime format from `http://www.aus-vo.org/software.html`, and four frames from the animation are shown in Figure 13.

7 Conclusion

We have described the extension of the shear-warp volume rendering algorithm with perspective to a distributed data volume rendering system. Sub-volumes of the data are distributed to rendering nodes which produce intermediate images for compositing. Rendering and compositing uses the associative over operator to yield a valid final image. Our software, DVR, performs exceedingly well compared to other state-of-the-art systems including shared memory supercomputers, and we have reported the first successful volumetric rendering of an 8 Gvox volume with non-specialised hardware. DVR is available for download from the software section of the Australian Virtual Observatory website, `http://www.aus-vo.org`.

Acknowledgments

We acknowledge the Victorian Partnership for Advanced Computing for supporting this project through a 2002 Expertise Grant. We express our gratitude to Juergen P. Schulze for sharing his rendering core with us and allowing us to redistribute it. We also thank P. Lacroute and M. Levoy

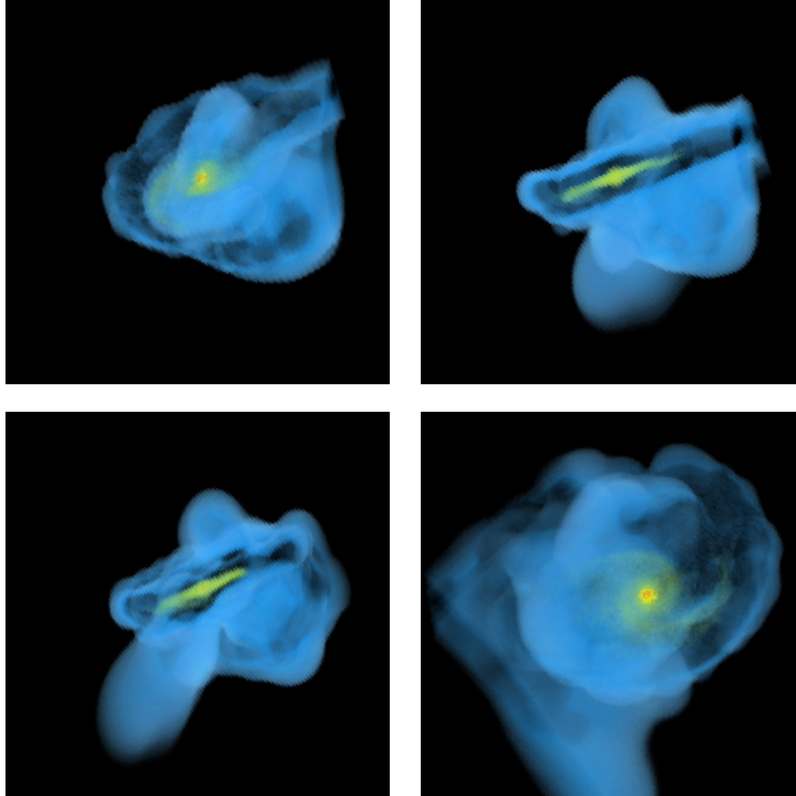

Figure 13: Four views of the final time-step of a simulation of the perturbation of the Milky Way disk by an intruder dwarf galaxy, generated by a smoothed particle hydrodynamic code (Kawata, Thom & Gibson, *in prep.*). 100000 particles were smoothed into a 128^3 volume and rendered using DVR.

for kindly giving us permission to reproduce figures 1 and 2 from Lacroute & Levoy (1994), and D. Kawata and A. Knebe for allowing us to use their new N-body simulations as example data sets. Finally we thank the referee for valuable comments on the manuscript and for pointing out the possible use of boundary conditions in simulation realisations.

References

Barnes, D.G. et al. 2001, MNRAS, 322, 486
Barnes, D.G. and de Blok, W.J.G. 2001, AJ, 122, 825
Blinn, J. 1994, IEEE Computer Graphics and Applications, September 1994, 83
Calabretta, M.R. and Greisen, E.W. 2002, A&A, 375, 1077
Drebin, R.A., Carpenter, L., and Hanrahan, P. 1988, Computer Graphics, 22, 65
Gooch, R.E. 1995, in Astronomical Data Analysis Software and Systems V, ASP Conf. Series vol. 101, eds. G. H. Jacoby and J. Barnes, (San Francisco: ASP), 80
Halsey, R. and Chapanis, A. 1951, J. Optical Soc. of America, 41, 1057.
Kawata, D. 1999, PASJ, 51, 931
Knebe, A. Green, A. and Binney, J. 2001, MNRAS, 325, 845
Lacroute, P. and Levoy, M. 1994, in SIGGRAPH '94: Conference Proceedings, ed. S. Cunningham, (New York: ACM), 451
Oosterloo, T. 1995, PASA, 12, 215
Ortiz, P.F. 2003, `http://barbara.star.le.ac.uk/datoz-bin/datoz2k`
Porter, T. and Duff, T. 1984, in SIGGRAPH '84: Conference Proceedings, ed. H. Christiansen, (New

York: ACM), 253
Schulze, J.P. and Lang, U. 2002, in Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization, eds. D. Bartz, X. Pueyo and E. Reinhard, (Aire-la-Ville: Eurographics Organization), 61
Snavely, A., Johnson, G. and Genetti, J. 1999, in Proceedings of the High Performance Computing Symposium - HPC '99, ed. A. Tentner, (SCS), 59
Sterling, T.L., Savarese, D.F., Becker, D.J., Dorband, J.E., Ranawake, U.A. and Packer, C.V. 1995, in Proceedings of the 1995 International Conference on Parallel Processing, ed. P. Banerjee, (Boca Raton: CRC Press), I:11
Stoughton, C. et al. 2002, AJ, 123, 485
York, D.G. et al. 2000, AJ, 120, 1579