Volume Visualization of High Resolution Data Using PC-Clusters

David H. Porter
dhp@lcse.umn.edu
Laboratory for Computational Science and Engineering, University of Minnesota

Abstract

There is a need to visualize multi-billion voxel data sets. Hardware accelerated rendering currently available on PCs provides a cost effective and scalable way to render these very large data sets. Very large voxel data can be rendered on systems with limited texture and system memory piecemeal by decomposing the data into a set of tiled sub-domains, which are sequentially read from disk. Hence, voxel resolution is only limited by disk space. To provide a trade off between image quality and overall rendering speed, and particularly for interactive modes of display, it is useful to generate (prior to rendering) a sequence of reduced resolution representations organized in a tree hierarchy. Such a "Block-Tree Hierarchy" may be thought of as a 3D mipmap. For rendering on clusters of PCs, it is useful to encapsulate the rendering application as a component object module, which may be invoked on remote hosts. Rendering parallelism is possible both across frames in a movie and within and individual frame by decomposing the rendering of an image into a tiling of sub-images, which divides the work of both casting voxels and filling pixels. Implementations of these ideas, and their use in visualizing billion voxel volumes on PCs, are described here.

1. Motivation

Rendering techniques presented here were originally motivated by numerical studies of turbulent fluid dynamics in astrophysical applications, such as stellar convection, which lead to studies of fluid turbulence and the analysis of sub-grid scale modeling of turbulent flows in their own right. We found that in order to directly model an adequate range of scales for fully developed fluid turbulence, meshes of a thousand zones in each of three dimensions, and larger, were required. There is no known general theory of fluid turbulence. In particular, there is no general way to reduce the size or resolution of the data without risking the loss of essential components of the flow. Hence we have found it useful to save all of the fluid dynamic fields, such as mass density, fluid pressure and components of velocity at the full resolution of the computational mesh, and at a sequence of times spaced close enough that the most rapidly varying features were captured. Over a span of time long enough to follow the full temporal evolution of a non-equilibrium system, or a representative span of time in an equilibrium system, this leads to hundreds or thousands of snapshots in time, each of which containing several variables sampled on billions of mesh points. While traditional statistical reductions are good for comparison with theory, observation, and lab experiments, they only represent a tiny fraction of the billions of degrees of freedom present in our numerical models. One of the most exciting uses of well resolved data is to find unexpected features in the flow.

Volume visualization proved to be an effective way of presenting a large fraction of the entire data set in an interpretable way and see such features.

Volume visualization techniques are well known, and we have been volume rendering our data for over 12 years now (references [1] and [2]). However, a new generation of larger simulations has forced us to deal with a new set of problems, including how to manage Terabytes of data, efficiently render fields at resolutions which range into billions of voxels, find the gold nuggets (i.e., surprises) that are buried in the complexity of such data, and produce visualizations which will demonstrate results and motivate further inquires. One challenge is that well resolved fluid turbulence possesses great complexity over a wide range of interacting spatial scales. We found that the simplest possible representations of the flow minimized confusion and lead to the most interpretable images. Hence, we have focused on volume visualizing scalar fields, and use a simple alpha-blending model. Another challenge is that these large data sets typically do not fit in system or texture memory, yet we may want to see the full data set. Hence we needed a way to render large volumes in pieces and page data form disk. The complexity of the data also motivated us to develop ways to interactively specify views, sub-regions, and transfer functions which map a 3D scalar quantity to color and opacity. We have developed a graphical user interface to control these inputs to the rendering. In order to iteratively see the results of varying these controls at interactive rates, even on the largest data sets, we use a sequence of reduced resolution representations of the data, which allow the user to choose the tradeoff between quality and speed. In order to see the time variation of the full detail in these systems, at a frame rate high enough to be perceived as continuous motion, we found it essential to pre-render frames and play them back as movies. Hence, we have developed way to define and generate movies of well resolved data as background processes. The shear size of these data sets made the overall speed of rendering important. The need for efficient disk and memory access lead us to organize volume data into blocks which tile the full spatial domain. We use hardware acceleration to get high rendering speeds one the data is in memory. In order to improve the throughput of large rendering jobs, such as high quality movies, we have developed ways to do parallel rendering across PC clusters.

These choices and how they were implemented are discussed in section 2. Our experience in using these techniques in the Laboratory for Computational Science and Engineering (LCSE) at the University of Minnesota, including interactive rendering of large data sets on PCs, background rendering of large movies on PC clusters, interactive rendering of data sets staged at remote computer centers, interactive visualization on tiled wall displays, along with examples of images, movies produced with these applications are described in section 3. Finally, current and future developments of these applications are discussed in section 4.

2. Method

This section begins with a review of the choices made which specifies the kind of rendering we focus on. Rational is given for these choices. This section continues with the way these choices are implemented. Then the way that rendering, both for individual images and for movies, can be specified is given. Finally, various configurations enabled by these techniques are presented.

2.1 Choices

The choices we made in designing these applications were driven by the volume nature of our data. Lower dimensional structures, such as points, lines, and surfaces, were not a fundamental part of our data representation. While well defined 2D structures, such as shock fronts, slip surfaces, and contact discontinuities, can occur naturally in Euler fluid dynamics, we wanted these rendering applications to discover such lower dimensional structure if they were present rather than impose them as a fundamental part of the algorithm. Hence, these techniques are not intended to be a general purpose solution to all 3D rendering applications, but rather are focused on visualizing 3D field variations in high resolution volume data.

We choose to use perspective views to represent 3D data. Orthographic projection is possible as a limiting case. However, perspective provides a depth queue, is a natural way to generate interior views, as well as control the size and scope of sub-volumes. Further, a sequence of perspective views makes transitions of spatial scale and position visually obvious.

We have focused on representing complicated 3D data, such as from high resolution numerical simulations of fluid turbulence, in terms of scalar fields. We view one scalar field at a time. We found this to be an effective way to minimize confusion in the rendered image and leads to the interpretable images. Visualizing vector fields, such as velocity, and higher order tensor fields, such as velocity gradients, are important ways to understand a flow. We have found that visualizing scalar invariants of these multiple component fields to be effective. We compare different components of a flow by rendering separate images which share the same view and bounding box.

We use a color/opacity look-up tables (LUTs) to map a scalar field to red, green, blue, and alpha channels in each voxel. Rather than using a lighting model with an external source, voxels are rendered as being luminous with the associated color. Voxels block regions behind (relative to the view point) in proportion to their opacity size, providing another depth queue. The goal of using this simple and fairly standard alpha blending model is not photorealism, but to produce a simple and interpretable visualization of a highly detailed scalar field. Careful specification of the color LUT allows for effective mappings between value and visual impression, including highlighting special values. The alpha LUT is used to control the opacity threshold so that regions of containing certain ranges of values (usually high or low values) may be seen as 3D objects.

A scalar field is sampled on a regular rectangular array of voxels. The data is represented as one byte per voxel, which provides a good tradeoff between data compression and dynamic range. Hence, a map must be chosen which takes the original scalar field to integers in the range [0-255]. Simple maps are linear. A log scaling is effective for positive definite quantities with a large dynamic range, such as mass density in stellar convection. Hyperbolic tangent mappings work well for quantities distributed with wide tails, such a component of vorticity. Large volumes of data are tiled into blocks sized small enough to fit in memory and large enough to allow efficient disk IO. Optimal sizes of blocks tend to be in the Megabyte range. We typically choose block dimensions of either 64 or 128 voxels on a side.

We found that interactive specification of rendering parameters is an essential part of the investigation of high resolution data. One typically selects view points, sub-regions, and color/alpha LUTs iteratively in order to get an overall impression of the system and find features of interest. It is useful to vary the opacity threshold to see how robust a 3D feature is. Interactive visualization is also useful in specifying parameters for movies where the view point and sub-region follows the path through time of a feature of interest. In order to support interactive modes of use which are effective even on the largest data sets we needed to provide lower resolution representations of the data. For all of the reasons given above, the scalar field at each resolution is formatted in blocks of voxels which tile the entire domain.

2.2 Implementation

The visualization of a large volume of data is broken up into rendering a sequence of blocks which tile a 3D domain. Blocks are rendered in the order of farthest to nearest in a composite image. For each block, we implement a color/opacity model by casting 2D textures, taken as slices from the 3D blocks described above, in perspective onto a view port and alpha blending these images back to front according to the formula

$$\vec{C}_N = (1 - e^{-t})\vec{C}_C + e^{-t}\vec{C}_O$$
$$t = \frac{sA_c dx}{\cos(q)}$$

where $\vec{C}_i = (R_i, G_i, B_i)$ are the red, green, and blue color channels of the old (i=*O*), new (i=*N*), and current (i=*C*) rendering planes at each pixel. A_C is the alpha value of the current rendering plane at the associated pixel. The distance between rendering planes is *dx*, and theta is the angle between the normal to the rendering plane and the direction of view. Hence, $dx/\cos(q)$ is the length of the line segments which span one voxel width along the direction of view. The alpha value A_C may be thought of as an opacity line density, and t is an approximation to the optical depth of the voxel layer along the line of sight. Since perspective is used in these renderings, q should really be a function of position on the rendering plane. The RGBA values of the current rendering plane for each pixel on the image is derived via tri-linear interpolation from the perspective image of the 2D texture.

This standard alpha blending technique is well supported on most PC hardware graphics accelerators in terms of high level API calls. We use the OpenGL API routines to alpha blend 2D textures in perspective (reference [3]). In order to be able to run on as many kinds of PCs as possible, the rendering application uses only the OpenGL standard API, which limits us to using 2D textures, power of 2 dimensions for each texture, and non-paletted (i.e., RGBA) textures. Hence the default mode of rendering is to extract a mesh aligned 2D section from a block, expand the single byte value in each texel to a 4-byte RGBA texel value in software, and load the 4-byte texel via the standard OpenGL call. Paletted textures are implemented as an option in the rendering server, where the single byte texel values and color/alpha LUTs are loaded via the OpenGL API extensions. The use of 3D textures were implemented, and very effective, for versions of our rendering applications which we ran on SGI Infinite Reality pipes. However, 3D textures are neither uniformly nor well supported on PC graphics cards to date. Hence, we have not implemented 3D textures as an option for PC versions of this code. The constraint of texture dimensions being a power of 2 has not been problem. Since we tile large volumes into blocks, we are free to choose most of the block sizes to be powers of 2. Only edge blocks need be irregularly sized. We copy any non-power of 2 blocks into larger blocks, which are dimensioned in powers of 2, and cast only the relevant sub-regions.

To support rendering of large 3D volumes of data which are tiled in blocks, along with reduced resolution representations of the data which are also tiled in blocks, we organize this multiple resolution volume data into what we call a “block-tree hierarchy”. The sizes and locations of blocks are chosen so that sets of higher resolution blocks may always be found to tile regions covered by any block at lower resolution. Such a spatial decomposition is naturally organized into a tree structure, where a set of blocks at one mesh refinement which exactly cover the volume of space of a single block which is one level of mesh refinement courser, are children nodes of that block. A classic example of this kind of division of space is an oct-tree hierarchy, where a region of space is divided into octants, each of which may be divided in the same way. Our block-trees are not constrained to be such regular divisions of space. Instead, block sizes are chosen to optimize overall rendering speed, which is impacted by both disk IO and texture rendering speeds. Most rendering hardware works best when textures are a power of 2 in each dimension. And, at least on PCs, disk read performance tends to increase with the size of the IO request at least up to 256KB. On fiber-channel and striped disk systems IO speeds continue to increase substantially up through request sizes of several MB. Since we store one byte per voxel, 128-cubed is a good choice for block size. For both simplicity and optimal disk IO and rendering speeds, such a block of bytes is stored both on disk and in PC system memory in the memory order of a simple 3D array.

Since we use tri-linear interpolation, and want a seamless representation between blocks, the sub-region of space represented by each block needs to include a one voxel deep boundary. This boundary is included in the simple 3D array of each block. Hence, the spatial extents of adjacent blocks, which are at the same resolution, overlap by two voxels. The full block dimensions, including boundaries, should still be powers of 2, so the interior block dimension is typically of the form $2^n - 2$, where n is an integer. Most blocks span 126 voxels in each direction with a one voxel deep boundary. In general, not all of the blocks in such a tiling will be of the same size. If the layout of blocks starts at one corner of a regular 3D mesh then there will be smaller blocks along the three opposing faces of the volume, unless the mesh happens to be a multiple the common interior block size (e.g., 126). These smaller edge blocks are not a problem since the rendering algorithm copies them into larger meshes as described above. The time required for this extra memory copy is not a significant fraction of the overall rendering time since these edge blocks tend to be small in size and few in number.

Given a regular sampling of space at the finest resolution which is broken into blocks as described above, there are still many possible block-tree hierarchies. A simple and efficient way to construct block trees, as described above, is to choose a single integer reduction factor (Nrep) between levels of resolution, blend (or sub-sample) the entire domain by this factor, and then tile this reduced grid into blocks of the same voxel size as used on the finer mesh. By this construction, the interior of each reduced resolution block is always exactly covered by the interiors of some set of blocks at finer resolution.

The best way to generate voxel values at reduced resolutions is both data and application dependent. If we were to store full RGBA information in at each voxel, then the simplest and best reduced voxel values would be linear averages of each color and alpha channel over all of the finest resolution voxels covered by the lower resolution voxel in question. However, we use color/alpha LUTs for both flexibility and data compression. If these LUTs are linear, then linear averages still work well. We tend to use the alpha channel to apply non-linear thresholds on what is visible vs. transparent. If the quantity being rendered fluctuates rapidly over a few mesh points on the finest mesh, then the average value of a small contiguous set of voxels that may fall on the transparent side of an opacity threshold even though some of the original voxels would have been rendered visible. If it is important to find the locations of the extreme values of such a highly intermittent quantity, then the maximum or minimum value, rather than the average value, may work better. Other statistical measures, such as the median or the mode may be useful in some applications. For the most part, we tend to use the average value.

A small amount of meta-data is included in a block-tree structure. This meta-data includes file names and offsets into these files where the blocks reside, the 3D dimensions and spatial offset of each block, the replication factor of each block, and the parent (if any) of the block. The root of the tree, which contains the lowest resolution representation of the data, has not parent. In addition, each block is assigned an error value, which corresponds to the maximum deviation in 1-byte voxel value of any voxel in that block from the finest resolution representation voxel that it covers. All of the blocks at the finest resolution have zero error by construction. Where the data is smoothly varying, or constant, over an entire block the error will be small, or zero. This error information is useful in deciding whether a block at a coarsened resolution is adequate for a given rendering application, or whether it needs to be broken into its sub-blocks.

The extra information in all of the reduced resolution representations, overlapping boundaries of each block, and meta-data tree information increases the total data storage and IO by only a small factor. Even with a reduction factor of two in each direction, each successive coarsened representation of the data adds 1/8th of the previous. This limits the total size of all of the coarsened representations to no more that 1/7th of the original data. The one voxel layer deep boundaries on each 126-cube interior add about 5% to the total. The meta-data for each block is about 80 bytes, which is negligible. We impose the offsets of blocks within disk files to be multiples of common memory and disk page sizes (such as 16KB or 64KB) in order to facilitate high performance disk IO. Spacing blocks to be page aligned could add as much as an extra 3% to the total data. However, most blocks are exactly 2 MB in size and tend to start on page boundaries with no spacing. The smaller edge blocks are a small faction of the total. Hence, the total data size of a block-tree hierarchy tends to be between 14% and 18% larger than the original data.

Given a way to render a sequence of blocks, and a block-tree representation of a quantity, an image may be rendered by scanning through the tree structure, starting with the root of the tree. The tree meta-data, which is stored all together at the beginning of one file, is read in. The node-parent meta-data associated with each block is used to generate a set of pointers to the children of each node. An ordered list of nodes to be rendered is generated iteratively. The 1st iteration of this list is the single node at the root of the tree. The iterative step is to consider each node in the list: if it is an accurate enough representation of the sub-domain it covers, then it will be used for rendering; if not, then it is replaced by all of the children of that node in the order from back to front relative to the eye point. This step is iterated until no nodes are broken up. If a node in the resulting list contains a region which is entirely outside the viewing frustum, then that node is removed from the list. Note that only the meta-data of the block tree is used to generate the final list of nodes to be rendered. Hence the entire list is known before any disk IO or rendering takes place. This allows disk access to run in parallel with rendering, with disk reads going into, and rendering data taken out of, a circular buffer. These two threads run asynchronously for the most part. The rendering thread need only wait for the block it is just going to render to be read in. The disk IO thread need only wait when the circular buffer fills: blocks of this memory are freed by the rendering thread as it completes rendering of each block.

Three criteria are used to decide if a node's representation is adequate: the largest angle subtended by any voxel in the block; the LUT index error associated with the block; and the replication factor of the block. If any of these three criteria are met, then the block is considered good enough. Typically, the angle subtended by a voxel should correspond to the angle subtended by one pixel in the viewing frustum. An appropriate threshold for the LUT index error is both data and color/alpha LUT dependent. However, data is frequently constant or slowly varying over large sub-regions and almost any small threshold on the error will work well. Limiting the tree scan by voxel size is a simple way to limit the rendering time for interactive rendering.

2.3 Encapsulation

To facilitate a variety of uses, the rendering algorithm described above is encapsulated as a component object module. This component, hvr_server, is invoked for interactive, background, remote, and PC cluster uses. It is controlled by, and rendered images are returned through, the method calls of the component's interface. True encapsulation is broken only for the heavy disk IO required for reading in block-tree data. Hence, this server relies on block-tree data being pre-staged on disks which are accessible to the host it runs on, as opposed to the host it is being controlled from. This minimizes both the frequency and total amount of communication between the rendering component and the process that drives it.

The interface for hvr_server includes method calls for setting the horizontal and vertical fields of view and pixel resolution of the viewing frustum, the eye point, look at, and up vectors, near and far clipping planes, six extra clipping planes for defining a sub-region bounding box, color/alpha LUTs, and rendering quality parameters. All of this information is passed via arguments passed into a few method calls, and correspond to a very small amount of data. The rendering server generates an OpenGL view port on the host it is running on and renders the image in that view port. For interactive modes of use, the view port is mapped to a window on a local display device. For background modes of use, hvr_server has a method call which passes back a rendered image, which is usually no more than a few MB in size. Returning such an image take a fraction of a second across a local area network, while typical rendering times for the large (GB) block-tree data are in the tens of seconds even with the fastest rendering hardware and disk drives.

So far, hvr_server has only been implemented for PCs running Microsoft Windows. In this environment the component framework is DCOM (reference [4]), which supports invocation, method calls, and deactivation of components, both on the local host and remotely.

3. Results

This section starts with examples of how we are using these applications at the LCSE. Examples of the final product of these rendering techniques, both still images and movies, are then provided. And finally, pointers to down loads of these applications are provided so that you can try out these techniques on your data using your own PC or cluster of PCs.

3.1 Configurations

Given that volume data has been put into block-tree format and staged on disk we usually start, we usually start with interactive visualization. This is where we start to get an overall impression of the data, identify features and surprises, and start to tweak color/alpha LUTs. Interactive visualization is also used for defining the key frames which may be interpolated between to make movies. We have written a graphical user interface, hvr_gui, to drive the rendering component interactively. In this mode of use, hvr_server and hvr_gui are run on the same host, see Figure 1.

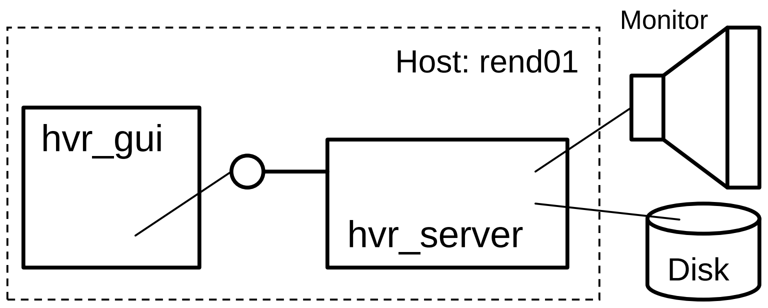

Figure 1

Here, hvr_gui is run on a local host, usually a desk top or laptop PC, it opens and manages a display window which is mapped to the hosts monitor. Hvr_gui instantiates a copy of hvr_server on the local host, and passes a handle for the display window to the server. The server then creates an OpenGL within the GUIs display window. The window handle is valid for the server because both server and GUI are running on the same machine.

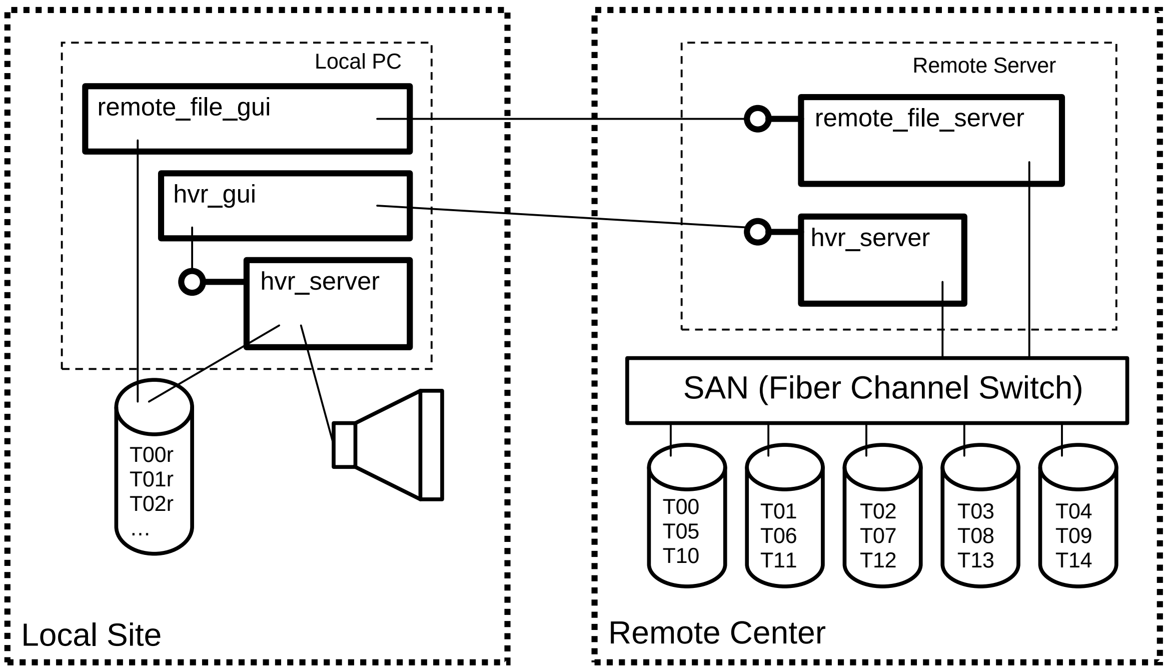

Figure 2

The block-tree format facilitates a practical way to mine very large volume data which is staged at a remote site. Suppose that a scalar 3D field is saved at a sequence of times, and is staged in block-tree format on a storage area network (SAN) at a remote center, see Figure 2. Here, T01, T02, … T14 represent disk files, each of which contains the block-tree volume data of the scalar field at one snapshot in time. At the LCSE, we cyclically distribute such time sequences across disk stripe groups for parallel disk read efficiency. The root of each block-tree is a relatively small (typically 2 MB) single brick of bytes which can be copied to a PC at your local site, which may be a center in another state, or simply your office. We have written a remote file server "remote_file_server" which is designed to extract root blocks from block-tree files and ship them to a client. The remote file server is written as a distributed component object module and may be invoked from a remote host. We have written a graphical user interface "remote_file_gui" to drive remote_file_server and write a set of root blocks to disks attached to the user's local PC. These single "root" blocks are written in block-tree format, and represented at T00r, T01r, T02r, … in Figure 2. The volume rendering GUI and server may then be run locally on these small block trees to define views, choose times, and set color/alpha LUTs. Once a view is defined, hvr_gui can invoke a copy of hvr_server at the remote center and use it to render the view at full resolution. The copy of hvr_server running at the remote center then ships the finished image back to the hvr_gui through a method call. Then hvr_gui ships this same image to the local copy of hvr_server, again through a method call, for display, see Figure 2.

We use these interactive sessions to specify movies and submit them to a volume rendering queue which runs on a cluster of PCs in parallel. Effective color/alpha LUTs, along with times, locations, views, and potentially time evolving structures of interest are identified through such interactive sessions. The graphical user interface, hvr_gui, can organize and save all of the information needed by the rendering server to render an image in a very small amount of information, provided that the large block-tree volume data is given by the path and name of the disk file and not the volume data itself. This information compactly identifies the field, time, sub-region of space, and view point from which an interesting feature may be seen. Several such views may be used as key frames to specify a movie. All that is needed is a way to interpolate between key frames. We have built into hvr_gui facilities for saving key frame information, selecting a sequence of key frames, and defining interpolation rules between consecutive pairs of key frames. Even though a movie specified in this way may consist of hundreds or even thousands of frames, the information needed by the rendering server to generate the movie is small since the large volume data is merely pointed to by its' disk file name. We call this information a "movie path" since is contain continuous "paths" through space and time of the eye point, look at vector, up vector, clipping planes, and color/alpha LUTs.

We have written a simple queuing system, hvr_queue, as a component object module which may be invoked from a remote host to receive movie path information generated by hvr_que, see Figure 3. This queue is designed to receive movie paths at any time, from multiple users, and immediately store this movie path information to its' local disk. Hvr_gui then detaches from, by releasing the pointer to, hvr_queue and does not wait for the movie to be rendered. Hvr_queue drives a parallel PC cluster to generate movies from the paths it receives. The time when a movie gets rendered, and which PCs are used in the rendering cluster, is usually best controlled locally. A graphical user interface, hvr_gueue_control, is used to invoke a copy of hvr_queue, specify which PCs will be used in the rendering cluster, and set the cluster going or shut it down. The queue invokes one copy of hvr_server on each PC in the cluster and sends the information to render individual frames to each rendering server. A fiber channel switch is used to allow any rendering PC to get at any block-tree data file. As each server finishes a frame, it sends the completed image back to the queue via a method call to the servers interface, the queue write the image data to a disk file and gives the server the next frame in the movie to render. Each rendering server is controlled asynchronously by a separate thread running in the queue so that parallel rendering is load balanced across frames. Some fault tolerance is designed into this queuing system. If a rendering PC hangs or dies for any reason, the queue will detect an error condition from method calls to the server running on that PC and remove the faulty PC from the rendering cluster.

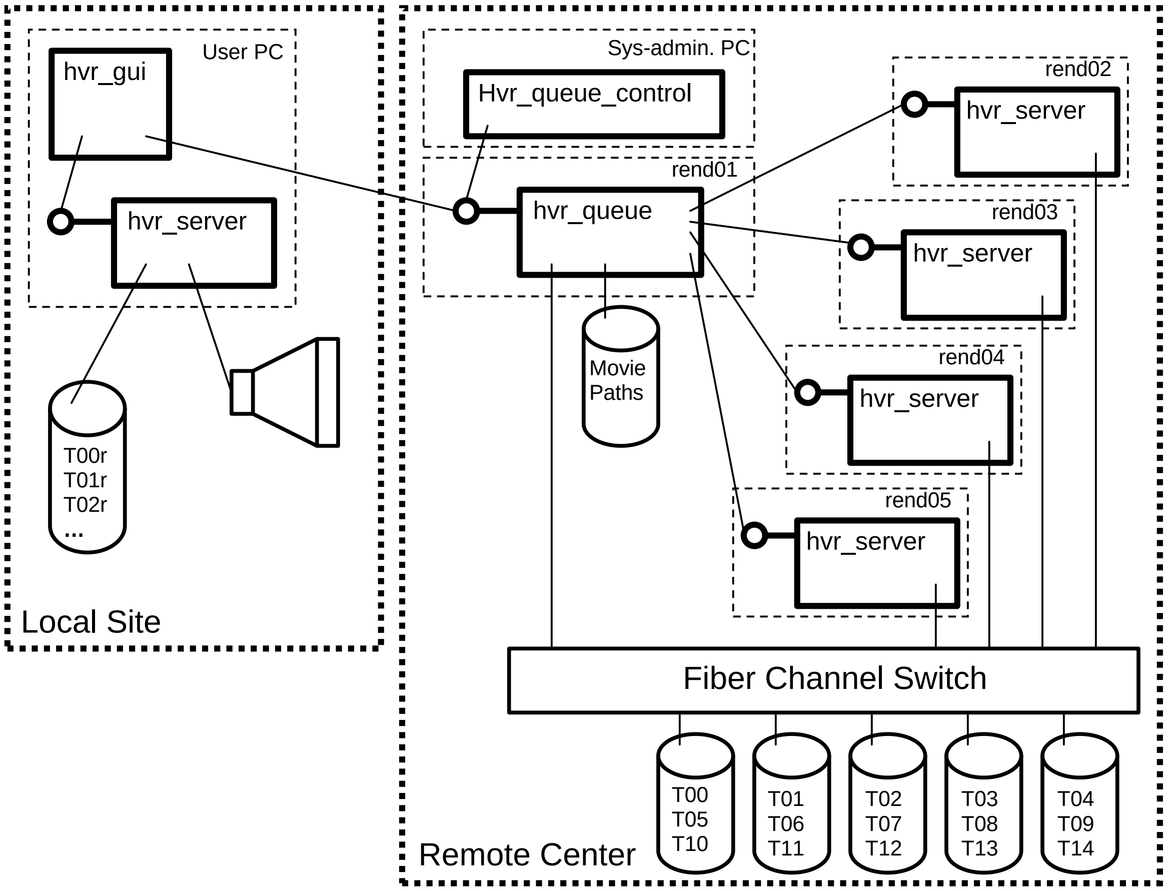

Figure 3

Hvr_gui is also used to specify the pixel resolution and geometry of the display on which these movies will be shown. The display is specified in terms of a number of panels.   A template view frustum is specified for each panel in terms the panel's pixel resolution, horizontal and vertical width, location, and unit vectors normal to the panel and in the panels vertical direction. These template panels are thought of as being attached to the eye point, with offsets and orientations rotated to correspond to the look-at and up vectors for each frame in a movie path. Hvr_qui send hvr_queue one set of template panel frusta for each movie submitted. As each frame of a movie is generated, hvr_queue sequentially sends hvr_server each template panel and hvr_server translates and rotates the template panel according to the eye, center and up vectors for current movie frame.

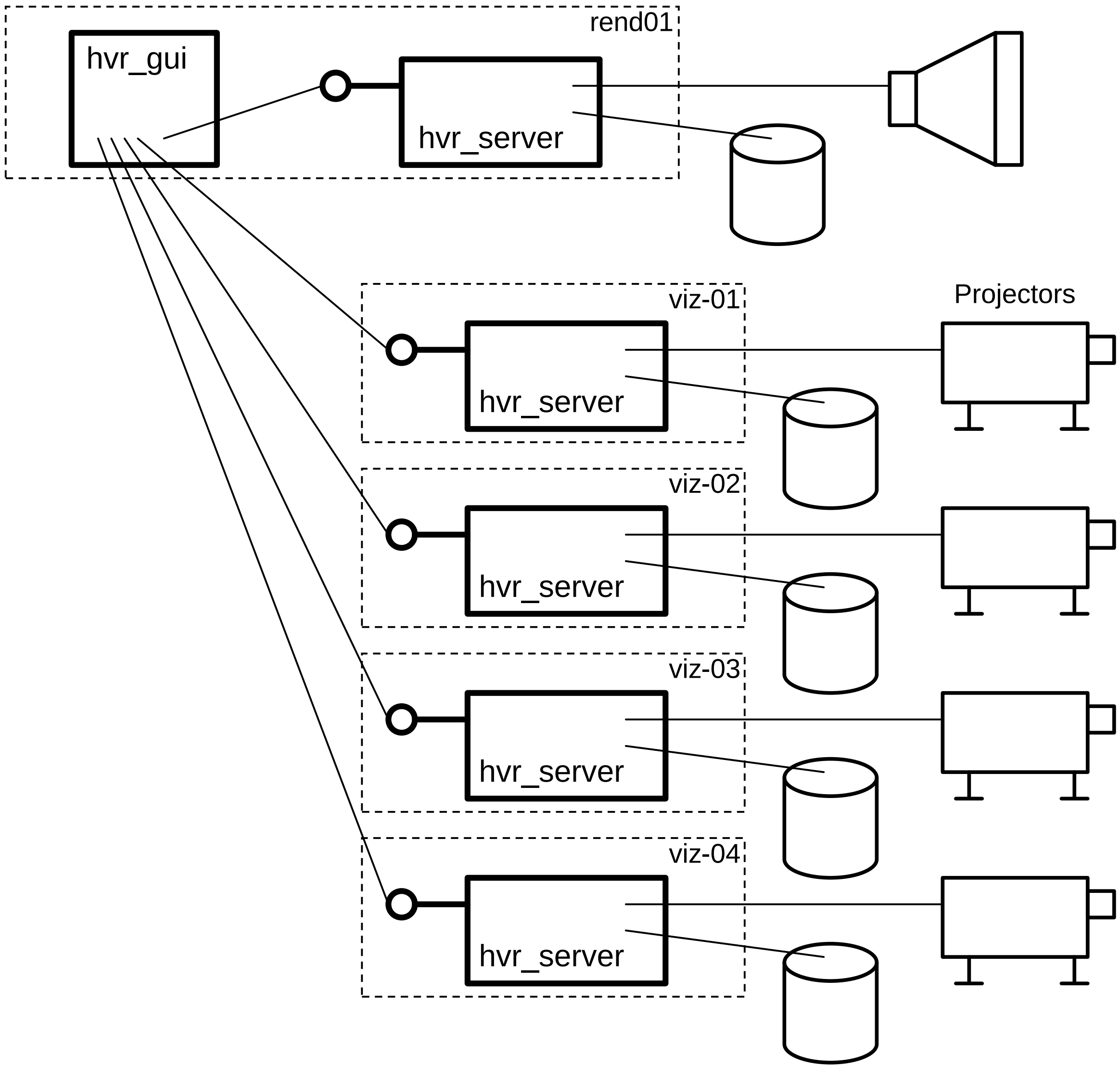

Figure 4

We also use hvr_server and hvr_gui to interactively render large volume data on tiled wall displays. At the LCSE, we have a tiled wall display (PowerWall) composed of ten rear projected tiles. Each projector displays the monitor of one PC. Hvr_gui instantiates one copy of hvr_server on each of these PCs, see Figure 4. Template viewing frusta for the panels of the display wall are generated by hvr_gui and the appropriate template frustum is sent to each copy of hvr_server. The GUI also instantiates a local copy of hvr_server, which is used interactively in the normal way. However, as each frame is rendered on the local copy of hvr_server, the GUI retrieves the rendering information from the local server and sends it to each copy of hvr_server on the PC cluster. Also, the local server is set to render only the root block of the block tree, since the high quality display rendered in parallel on the tiled wall. In order to get maximum interactive speeds for this mode of use, block-tree data is copied to locally attached fiber channel drives on each PC in the cluster, as shown in Figure 4. This mode provides the fastest way to render very large volumes, since each PC only reads in and renders the blocks that are in its' view frustum. Also, the extra screen resolution really does help in untangling the clutter of detail present in, for example, a billion voxel representation of fluid turbulence. This has proven to be an effective mode of presentation to small groups.

3.2 Visualization Examples

Fully developed 3D fluid turbulence is characterized by vortex tubes which can be very long, spanning the largest scales in a problem, and yet have diameters near the dissipation scale, which is usually much smaller and near the scale of the mesh in a numerical computation. The magnitude of vorticity is a scalar field which works well for visualizing vortex tubes. We shall use the vorticity field from a one billion zone calculation of decaying fluid turbulence (references [5], and [6]) to illustrate trade offs between image quality and rendering speed afforded by using different levels of a block-tree hierarchy.

The top three rows of panels on the color page show the vorticity field at a late time when the turbulence is fully developed (top two rows) and at an early time before the vorticty has broken into tubes (3rd row) from a 1024-cubed simulation of decaying turbulence. All twelve panels show the same section of the full volume, which spans 1024x1024x256 voxels. In each row, the four panels show the same field, rendered using the same color/alpha LUTs. The only difference between panels in a given row is the reduction factor, in the block-tree hierarchy, of the blocks used. From left to right the reduction factors are 8, 4, 2, and 1. Hence images at the far left were rendered using just one 128-cubed block in about 0.1 sec, which included the time to read in the block. The images at the far right were rendered at full resolution using 192 blocks of size 128-cube plus 51 smaller blocks in 12.1 sec. These times are dominated by disk IO, which includes a significant start up time. If blocks are cached in memory, then IO times are insignificant, and a 128-cube can be rendered in 0.015 sec. on a PC with an nVidia GeForce 3 graphics card.

The relative quality of the images can be seen to depend on both the color map and the nature of the data. Reduced representations were generated by taking linear averages of the 1-byte voxel index values. The grey scale used in the 1st row, is a linear map between index value and RGBA values and works well for this highly intermittent field. The second row shows the same field which was rendered using nonlinear color/alpha LUTs: the alpha LUT was 0 for over the lower 2/3rds of the scale, and linearly ramped up over the last third; the color LUT ramped from green through blue to white over the top third of the scale. In the full resolution image, far right, the vorticity is either lower than the opacity threshold, or well into the blue to white region of the color map. There is very little green. As the reduction factor is increased, a higher percentage of voxel with values mapped to green show up, due to the high values of vorticity in very thin vortex tubes averaging with the small values surrounding it. The vorticity at earlier times is organized in larger coherent regions, third row of panels, and there is little color shift due to blending as the reduction factor is increased.

The next example illustrates how block-threes may be used to efficiently render irregularly shaped regions, even if the original volume data uniformly samples space in a rectangular volume. Convection zones in red giant stars span nearly the entire volume of the star and are roughly spherical in shape. These stars are also observed to pulsate in radius. Numerical simulations of the convection zones of red giants (references [7] and [8]) have been carried out on rectangular meshes in which only the nearly spherical interior of the star is followed and everything outside the nearly spherical surface (photosphere) is approximated as a vacuum. The surface is free to move though the mesh as the star pulsates. Due to large convective velocity the surface can be bumpy and deform significantly from spherical. Hence, the region we wish to visualize in these models is irregular and time varying. The panel in the lower left hand corner shows temperature fluctuations from a numerical simulation of a red giant star which was carried out on a 512-cube mesh. The near clipping plane is pushed into the volume to show the interior. Gas pressure turns out to be a reasonable depth coordinate for calculating temperature fluctuations. Relatively cool temperatures (compared to the average temperature for the local pressure) are blue and ramp up to aqua for very cold. Relatively hot temperatures are assigned red color and ramp to yellow for extremely hot. Neutral gas is transparent in this rendering. The principle block size used here is 64 to take advantage of the geometry. Roughly half of the full resolution blocks (replication factor of 1) are outside the surface, have zero temperature fluctuation, and are not used, there-by cutting rendering time in half.

The final example, shown in the lower left corner of the color page, illustrates how visualization of very large data sets can reveal a wealth of surprises. The mixing layer induced by the passage of a shock wave across a fluid interface was simulated on an 8 billion zone mesh (2048x2048x1920). The entropy was constant in each of the unmixed fluids, but the two fluids had different entropy. Entropy is used here to visualize the degree and nature of fluid mixing. Opacity ramps from completely transparent for the entropy of the gas on top, to completely opaque for the gas below. Color ramps from black for the gas above, through blue, green, and red to white for the gas below. Hence, solid white represents one gas, transparent the other, while colors represent degrees of mixing. A thin section of part of the mixing layer is shown here. This section is about 100 zones deep and spans roughly 1200 zones in the horizontal. Mixing is seen to be anything but uniform. Near the left hand side of the panel the interface between the two fluids is still thin and smooth. Near the right hand side mixing has proceeded down to the mesh scale. A plume of one gas has penetrated into the other (top left), with fluted columns of gas near the base. Variously sized, shredded, and yet unmixed fragments of the white gas are seen all over the mixing region. The varying degrees and kinds of mixing that such visualization shows to be simultaneously present in this mixing layer have motivated new analysis in sub-grid scale turbulence modeling (references [9] and [10]).

3.3 Movies

Two movies are provided with this paper. The first movie, which is available at http://www.lcse.umn.edu/hvr/h13diag.avi, shows the mixing of a passively advected quantity (dye) in a system of decaying fluid turbulence similar to those described in reference [6]. The fluid system is in a periodic box. Random, long wavelength, and smoothly varying velocity, gas pressure, and density perturbations are imposed initially. These perturbations fluid dynamically tear them selves apart to form smaller and smaller perturbations in the classic 3D fluid turbulence cascade. This system is numerically simulated on a 1024-cube mesh and 150 snapshots are saved over 3 flow times. Each of the 150 block-tree hierarchy files, representing dye density, is just over 1 GB in size.

The density of the passively advected dye is initially distributed in a linear ramp in the Z (vertical) coordinate, starting with 0 density in the middle of the cube, increasing to 0.5 at the top, continuing to increase from 0.5 at the bottom of the periodic cubical volume, and reaching 1.0 at the middle of the cube where it discontinuously jumps back to 0. The opacity LUT was zero for density of the dye less than 0.5, the opacity linearly ramps from 0 to 1 for values of dye density from 0.5 to 1. Over this last range, colors ramp from black through pink and yellow to white. The movie shows the dye mixing with time from a static viewpoint. Then the viewpoint is continuously adjusted and the near and far clipping planes are brought closely together to show a thin vertical section. Detail of the time dependent mixing is then shown again in this thin section.

Then second movie, available at http://www.lcse.umn.edu/hvr/shock-tube.avi, shows, in some detail, the structure at the end, and the temporal evolution of the entropy from an 8 billion zone numerical simulation of a shock-tube mixing layer: the same simulation and visualization field as described in section 3.2 and shown at the bottom right of the color page. A total of 274 snapshots of the entropy were saved. The data in each block-tree hierarchies was just over 9.1 GB, for a total of 2.5 TB. This movie was originally generated for the LCSE 10 panel, 13.1 million pixel display wall.

3.4 Web Links and Downloads

Movies generated by several generations of this rendering software may be found on the web site http://www.lcse.umn.edu/movies .

Downloads for the PC applications described in this paper may be found on the web site
http://www.lcse.umn.edu/hvr/hvr.html .
Code for generating block tree hierarchies from regularly sampled data is also included on this web page. This code is wrapped in a subroutine, which may be called from code which either reads or generates your data. A Fortran 90 example is given in this download. A users guide for hvr_gui may also be downloaded from this web site.

4. Conclusions and Future Work

We have presented a practical way to interactively render, and efficiently produce high quality visualizations of, very large data sets on PCs. The size of the data is limited by disk capacity and not system memory. These techniques easily extend to clusters of PC for improved though put of movies as well as interactive tiled wall presentations. These techniques take advantage of the inexpensive and high performance hardware acceleration, as well as high performance disk IO, which are available on PCs today.

A component object module design of the volume rendering server facilitates a variety of uses, including interactive sessions on a desktop or laptop PC, interactive investigation of very large data sets which are staged at a remote site, parallel rending of movie frames in a cluster of PCs, and interactive tiled wall presentations.

Future work includes implementation of the componentized rendering server for Linux systems. Both Corba and DCOM for Linux will be tested for component and RPC functionality. OpenGL managed by glut will probably be used for Linux implementations.

These rendering techniques could be used in a collaborative mode. Two or more interactive sessions could be run simultaneously, with each graphical user interface being a client to all of the rendering servers. In a wide area application, data would be replicated on each of the servers. The small key frame meta data would be sent to each of the servers. Each participant would see the same view, and could control the view seen by everyone else.

A hierarchy of PC clusters could enable full resolution rendering of multi-billion voxel data at interactive rates. Each panel would be subdivided into a tiling of frusta. Each small frustum would be assigned to on PC in a large rendering cluster with locally attached drives and data replication for fast, parallel IO. Both the fraction of the volume to be rendered, as well as the number of pixels to be rendered, is reduced in proportion to the number of PC. Hence, a linear speed up should be possible on sufficiently large volume data.

We would like to acknowledge generous support for this work from the Department of Energy, through grants DE-FG02-87ER25035 and DE-FG02-94ER25207, contracts from Livermore and Los Alamos, and an ASCI VIEWS contract through NCSA; from the National Science Foundation, through its PACI program at NCSA, through a research infrastructure grant, CDA-950297, and also through a Grand Challenge Application Group award, ASC-9217394, through a subcontract from the University of Colorado; and from NASA, through a Grand Challenge team award, NCCS-5-151, through a subcontract from the University of Chicago. We would also like to acknowledge local support from the University of Minnesota's Minnesota Supercomputing Institute. This work was performed in part under the auspices of the U.S. DoE by the Lawrence Livermore National Laboratory under contract NO. W-7405-ENG-48.

References

[1] Porter, D.H., and Woodward, P.R 1989, "Simulations of Compressible Convection with PPM", in ACM SIGGRAPH Video Review Special Issue #44, "Volume Visualization State of the Art", produced and directed by Laurin Herr, copy right 1989, Pacific Interphace/Dupont.

[2] D. H. Porter; 1991, “Perspective Volume Rendering,” University of Minnesota Supercomputer Institute Research Report, UMSI 91/149.

[3] Mason Woo, Jackie Neider and Tom Davis, 1997 “OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.1”, Addison-Wesley

[4] Thuan L. Thai, 1999, “Learning DCOM,” O'Reilly & Associates, Inc., 101 Morris St., Sebastopol, CA 95472 Ed. Andy Oram

[5] I.V.Sytine, D.H.Porter, P.R.Woodward, S.H.Hodson, and Karl-Heinz Winkler 2000, “Convergence Tests for Piecewise Parabolic Method and Navier-Stokes Solutions for Homogeneous Compressible Turbulence,” J. Computational Physics, Vol. 158, pp. 225-238.

[6] D. H. Porter, P., R., Woodward, and A. Pouquet, 1998, “Inertial Range Structures in Compressible Turbulent Flows,” Physics of Fluids, Vol. 10, Issue 1, pp. 237-245

[7] D. H. Porter, P. R., Woodward, and M. L. Jacobs, 2000, “Convection in Slab and Spheroidal Geometries,” Proceedings of the Fourteenth International Annual Florida Workshop in Nonlinear Astronomy and Physics, “Astrophysical Turbulence and Convection,” University of Florida, Feb. 1999, J. Robert Buchler ed. Annals of the New York Academy of Sciences Vol. 898, pp. 1-20.

[8] M. L. Jacobs, D. Porter, and P. Woodward 2001, “Turbulent Convection in Deep Spheroidal Geometry,” In preparation for the Astrophysical Journal

[9] P.R.Woodward, D.H.Porter, I.Sytine, S.E.Anderson, A.A.Mirin, B.C.Curtis, R.H.Cohen, W.P.Dannevik, A.M.Dimits, D.E.Eliason, K.-H. Winkler, S.W.Hodons, 2000 "Very High Resolution Simulations of Compressible, Turbulent Flows," World Scientific.

[10] Ronald H. Cohen, William P. Dannevik, Andris M. Dimits, Donald E. Eliason, Arthur A. Mirin, Ye K. Zhou, David H. Porter, and Paul R. Woodward, 2002 “Three-Dimensional Simulation of a Richtmyer-Meshkov Instability with a Two-Scale Initial Perturbation,” LLNL Preprint UCRL-JC-144836, Submitted to Physics of Fluids.