Medical Diagnosis using the Cray T3D

Greg Johnson and *Jon Genetti*, Arctic Region Supercomputing Center, University of Alaska, Fairbanks, AK 99775-6020

Abstract

Volume Rendering has produced accurate images of internal anatomy that are useful for both diagnosis and education. While reasonable rendering performance can be achieved on current workstation technology, the emergence of higher resolution data (such as that from the Visible Human Data Set sponsored by the National Library of Medicine) is fast eclipsing the CPU and RAM limitations of single processor workstations. For this reason we have implemented several volume rendering engines on the Cray T3D that use an X client to display the results.

This paper focuses on a parallel rendering application that allows a user to explore large, high-resolution medical data sets. VolRender is a Cray T3D executable, controlled via a graphical user interface created in AVS (Application Visualization System), and runs on an X client that is attached to the T3D's host via HIPPI, FDDI or ethernet. The current system provides display rates of up to 1 frame per second over ethernet and up to 5 frames per second over FDDI.

1 Introduction

Direct volume rendering produces images of internal anatomy that aid in diagnosis and education. By making certain material types (like skin) mostly transparent, we are are able to "see" the anatomy behind. By adjusting the transparency and color of all the material types, the user can emphasize the part (or parts) of the internal anatomy they are most interested in. Figure 1 shows the upper portion of a dog leg with all of the anatomy visible. Figure 2 shows the same data set, but with all materials except bone completely transparent.

While a static image provides a wealth of information, a series of images from different viewpoints provides even more. It becomes easier to detect spatial relationships between parts of the anatomy, such as the position of the bone chips in Figures 1 and 2. A set of images can be precomputed and played back, but they often are not the exact motion needed. Our goal is to allow a user complete interactive control over the viewing position and to be able to generate and display 10-15 images per second. This would give the same interactive feel as rotating a polygonal object on a modern graphics workstation.

2 Direct Volume Rendering

Direct volume rendering involves taking a raw data set (composed of a 3D set of voxels), coloring each voxel based on its material type (bone, skin, muscle, vein, etc), applying a shading algorithm and then extracting a view. For computed tomography (CT) data, the coloring step is trivial. CT data measures the density at each voxel and the density ranges of each type of material are constant. For Magnetic Resonance Imaging data, the coloring step is still a time-consuming process. MRIs measure the movement of molecules in response to a change in a magnetic field and does not measure density directly. Therefore, material types must be identified by a person with a knowledge of anatomy.

The purpose of the shading algorithm is to simulate lighting and is based on the Phong reflection model [4]. This model requires a surface normal and is calculated by examining the densities of the neighboring voxels. For voxel $D_{i,j,k}$, an approximate surface normal is calculated by $N_{i,j,k} = (D_{i-1,j,k} - D_{i+1,j,k}, D_{i,j-1,k} - D_{i,j+1,k}, D_{i,j,k-1} - D_{i,j,k+1})$. A large difference between neighboring voxels indicates a high likelihood of a surface passing through that voxel. A small difference implies that the voxel is in an area of homogeneous mass and the transparency of that voxel will be increased. This is done so "boring" areas will become invisible and "interesting" areas are highlighted. Since this is not a binary decision like that used to produce a polygonal model with marching cubes [7], it produces a more accurate image with more information in it.

Figure 1: All parts of the anatomy visible.

Figure 2: Everything but bone is transparent.

Once all of the voxels have been shaded, there is a set of shaded slices that form the volume. There are many ways to produce an image from these slices and VolRender currently uses the "splatting" method [9] (see [5] for an analysis of direct volume rendering algorithms on the Cray T3D). Splatting is accomplished by projecting each voxel onto the view plane in a front-to-back traversal with respect to the view plane as shown in Figure 3. With a parallel projection, the footprint (or projection) of a voxel on the view plane will be constant for all voxels. Since a voxel will not project onto exactly one pixel, a filter is used to spread out the color and transparency to neighboring pixels.

For multiple images from different viewpoints, there are two choices—hold the light source constant or allow the light to move with the volume as the view position is modified. If the light source stays fixed, this requires the shading process to be re-done prior to re-rendering an image. If the light source can move with the volume, the volume does not need to be re-shaded—just another projection along the new view direction. In VolRender, the light source moves with the object, but may be moved while the view direction is fixed.

3 Parallel Splatting

High level language compilers for parallel systems are relatively new and unsophisticated. They are not yet adept at hiding many of the major issues which are peculiar to parallel programming. Consequently, the programmer must take into consideration such factors as the distribution of the data, distribution of the workload, synchronization and more. As with most things computer-related, each different configuration of solutions to these issues has its advantages and disadvantages. Fortunately, the dimensions of the main data structures involved in this problem are minimal (the shaded slices forming the data volume, and the raster) and a nearly optimal configuration is easy to pick out.

3.1 Distribution of the Data

The distribution of the data across multiple processors is inextricably linked with the distribution of the workload. A successful data distribution is one which minimizes internal and external IO, does not generate a significant level of work to achieve, and allows for a relatively equal distribution of the workload across all processing elements. For reasons which will become clear in the sections below, the distribution of the raw data slices is as follows. Contiguous regions relatively equal in size are distributed to each processing element as shown in Figure 4.

Note that if the number of slices n is not evenly divisible by the number of processing elements p, $\lfloor n/p \rfloor + 1$ slices are distributed to the first $n - \lfloor n/p \rfloor * p$ processors, and $\lfloor n/p \rfloor$ slices to the remaining processors. No provision is made for allocating parts of a slice to one or more processors, so the size of the processing group is chosen to be some number equal to or less than the number of raw slices in the data set to be rendered.

3.2 Distribution of the Workload

3.2.1 Shading

The calculations required to create a shaded slice z involve only the data contained in the raw slices $z - 1$, z, and $z + 1$. Consequently, a processor can shade its partition of raw slices independently of the rest, as this is essentially the shading problem applied to a smaller data

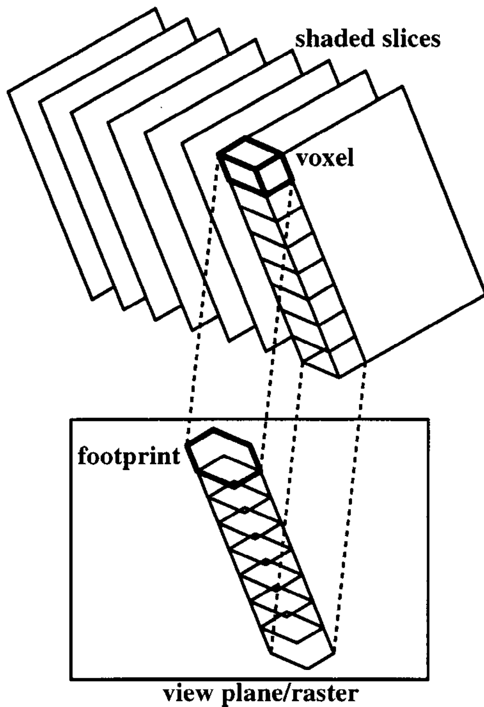

Figure 3: Splatting voxels.

set. Clearly shading a voxel based on its six neighbors does not effect a data or work dependency on any other processor performing similar calculations on a different slice. Furthermore, once the shading process is complete, every processor is left with exactly the shaded versions of the slices in its partition, thereby maintaining the original data distribution. At this point the raw slices are discarded as they are not used any further in the rendering process.

3.2.2 Splatting

For every processor to determine the color and opacity contribution of its partition of shaded slices to the same pixel, each must be aware of the general orientation of the data volume with respect to the view plane, and which corner of the volume is closest. Because a parallel view is used, the orientation of a processor's partition with respect to the view plane is the same as the orientation of the entire volume. All processors can determine which traversal order is appropriate, along with which splatting filter is best. Note that these calculations are identical for every processing element (PE). Clearly all processors can perform the computation required to generate the partial-color components for any particular pixel simultaneously and without concern for data or work dependencies. At this point, each processor p_i contains a full-size image, I_i, created from its set of shaded slices.

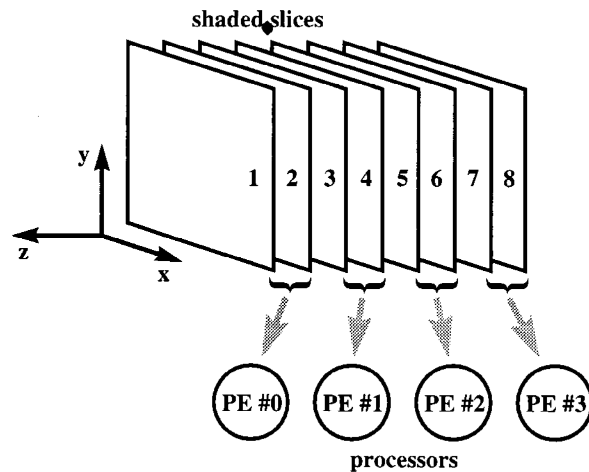

Figure 4: Distribution of the shaded slices.

3.2.3 Compositing

One caveat associated with this data distribution is that every pixel in an image I_i must visit every processor before a final image is created. Unfortunately, to complete an mxn pixel raster, this means that $mxnx(p-1)$ pixels worth of data must be moved between p processing elements. However, consider that the size of the raster is very likely to be less than the cumulative size of the raw data slices. Clearly, fixing some section of the raster in the memory of each processor, and shuffling around all of the slices of the volume as required to color each section is much less efficient than fixing a partition of the data volume in each PE's RAM and shuffling around the pixels of the raster as required.

To this end, the tiling arrangement shown in Figure 5 is used. In terms of this four processor example, the compositing works as follows. Once the PEs have calculated a partially colored raster, processor I fetches the Ith $1/p$ section of processor $I+1$'s partially colored raster, and composites it with the corresponding section of its own raster. At the same time, processor $I+1$ fetches the $I+1$th $1/p$ section of processor $I+2$'s partially colored raster, and composites it with the corresponding section of its own, and so on. At stage t, processor I fetches the Ith $1/p$ section of processor $I+(t \bmod p)$'s raster, and combines it with the corresponding section of its own. Note that since every processor is operating on a unique section of the image, data dependencies are avoided. This process continues until t becomes equal to p, at which time every processor will own a $1/p$ section of the completed image. Clearly the workload is well balanced. Furthermore, this configuration allows the sections of the raster to be written virtually simultaneously. The efficiency of this approach is clearly dependent on the hardware. The ARSC Cray T3D is currently equipped with two MIOGs to the host Y-MP. As a result the writing of the tiles cannot occur entirely in parallel, though the IO gateways are more likely to be used to capacity

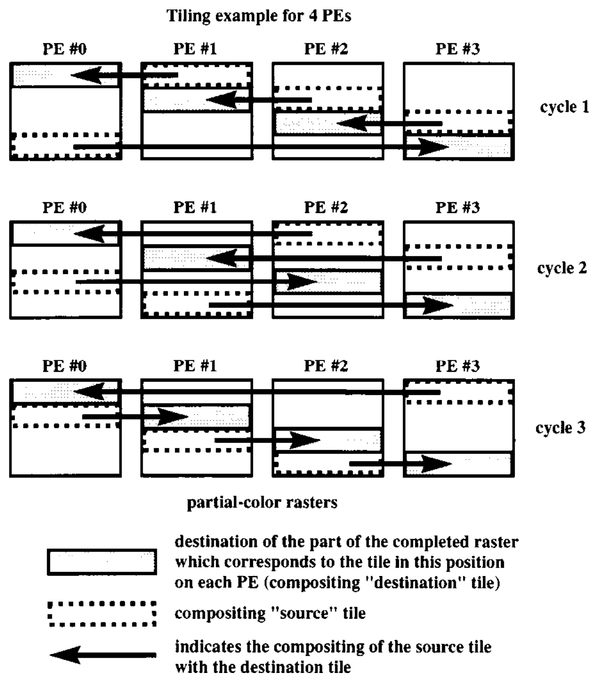

Figure 5: Compositing raster using a tiling approach.

when requested by multiple PEs.

4 The Complete System

The complete medical imaging system as currently running at the Arctic Region Supercomputing Center includes a graphical user interface along with a rendering engine based on the parallel design detailed in the previous section.

The GUI is written in C, with the application framework provided by the data flow visualization system AVS. Assembling the interface with AVS allows for a rapid prototyping design model and reduces development time by allocating widget communication and mouse event handling tasks to the AVS kernel. The fact that no T3D version of AVS exists, combined with dependency of the Cray T3D on a host system for external IO support, implies a distributed application architecture in which the interface runs under AVS on the Cray T3D host system while the volume processing engine resides on the T3D itself.

The parallel rendering engine is also written in C, with parallelization achieved through calls to functions in the PVM (Parallel Virtual Machine) and shmem (Cray Shared Memory) libraries [1, 2, 3]. Specifically, synchronization facilities are provided through the Cray MPP version of the popular distributed computing library PVM, while inter-processor communication is achieved through the use of the shmem library.

4.1 Interface

The functionality of the interface provided by the AVS module is accessed in the same way as that of other AVS modules. The module is loaded into the AVS network editor subsystem, and an instance of it dragged onto the work area (see Figure 6). As the volume rendering module itself is not intended as a standalone application, no effort was made to handle the display of the resulting images on the local host. Rather, this functionality is provided through the AVS Imageviewer module, and the X Window display system. Therefore the typical AVS network in which the VolRender module is expected to be used as shown in Figure 6.

The output of the VolRender module is an AVS 2D 4-vector byte regular field and the rendered images can be read and post processed by a variety of other AVS modules. Some image post processing techniques currently provided with AVS version 5.0 include convolution, contrast stretch, edge detection, image type conversion, crop, and vector element statistics.

Once instantiated, the interface panels for the VolRender module become available to the user. The 18 widgets in this interface provide for two methods of feeding rendering parameters to the rendering engine. The first method allows a user to set these parameters from the contents of an ASCII file. The second allows the user to set them interactively. The parameters are categorized by general function and include view position, lighting, tissue categorization, color, and opacity, processor group size, image size, and raw data characteristics.

4.2 T3D Executable

All of the work associated with the direct volume rendering of an image is accomplished by the rendering engine executing on the T3D. This executable is broken into three distinct pieces including a control function, a function for shading the raw data slices, and a function for splatting the shaded slices onto a raster. Control of the rendering process is directed by the VolRender module as described below. The control function is responsible for reading the parameters required to shade or render an image, and invoking the appropriate compute routine. As indicated previously, parameters associated with the lighting, color and opacity of the materials in the data volume, and raw data characteristics, are read and passed to the shading routine after a widget associated with one of these values is modified. The output is then rendered to generate an updated image. Alternatively, parameters dealing with the view direction with respect to the data volume, and the size of the raster, are updated and passed to the rendering function as required by the modification of a related widget.

4.3 Communication

The distributed nature of this volume rendering application requires a system of communication for handling process spawning, and inter-process synchronization and data movement. These tasks are accomplished with three separate communication facilities provided by version 8.0.x.x of the UNICOS operating system.

4.3.1 `execvp`

Once the VolRender module has been loaded into AVS and instantiated, the module loads the commands and parameters necessary to begin execution of the volume rendering engine into a character array. This array is then passed to the UNICOS command **execvp** to spawn the T3D executable responsible for the rendering.

4.3.2 Signals

Synchronization between the VolRender module running on the T3D host and the rendering engine running on the T3D itself is accomplished through the use of UNIX signals. This system provides a number of pre-defined interrupts which can be issued to a particular process to modify its operation. In addition, Cray UNICOS makes available two undefined signal values (`SIGUSR1` and `SIGUSR2`), to which a user definable function can be attached. A process receiving one of these signals temporarily halts execution while the action attached to that interrupt is processed.

From the VolRender module perspective, the receipt of `SIGUSR1` indicates that the rendering engine has completed an initialization routine and is ready to render an image. The receipt of `SIGUSR2` indicates that the T3D executable has completed a rendered image. Conversely, the receipt of `SIGUSR1` by the T3D process indicates that the value of a parameter in the VolRender interface which requires that the raw data be re-shaded and rendered, has been modified, while the receipt of `SIGUSR2` by the T3D process indicates that the value of a parameter which requires that the data volume be re-rendered, has been modified.

With this in mind, the order of execution of the VolRender module and the rendering engine is as follows. After instantiation, the VolRender module spawns the T3D process and waits until this process signals that it is initialized. At that time, the T3D executable waits until the VolRender module signals that a parameter has changed and an image needs to be rendered. After this signal is sent, the VolRender module waits until the rendering engine signals that an image has been completed, at which time it reads the image and passes it downstream to the AVS Imageviewer module for display. Meanwhile the T3D process waits once more for the VolRender module to signal that a parameter has changed and a new image needs to be rendered. The transfer of execution control continues in this manner until the VolRender module process is killed. At that time a termination signal is sent to the T3D process.

Synchronization in this manner helps prevent data dependency situations in which the VolRender module attempts to read an image before the rendering engine has completed it, or the T3D executable attempts to read a set of parameters before the VolRender module can finish updating them.

4.3.3 Process Table

The movement of parameters and image data between the VolRender module and the T3D rendering engine is handled through the use of the process table. This table allows the T3D executable to read from and write to sections of memory on the T3D host. The process on the T3D references the host memory as though it were a standard disk file. Reads and writes to this "process file" by the T3D are transparent to the host. It is this factor which necessitates the inter-process synchronization described in the previous section.

In this application, two forms of data transfer are required. The first involves getting the shading and rendering parameters from the VolRender process to the rendering engine. This is done by first storing all parameter values in a C structure. The address of this structure in memory is then passed to the T3D executable via the command line as it is spawned. The rendering engine accesses these values by opening the process file (the name of which is also passed to the T3D process at spawn time) and seeking to the appropriate address for a particular parameter value. As the structure of floating point values varies between the T3D and the host, parameters with floating point values are encoded as integer values prior to storage in the parameter structure, and decoded after being read by the T3D executable.

The second form of data transfer involves getting a completed image from the T3D back to the host. The rendering engine does so by seeking to the appropriate location in the process file and writing the data. Correct updates of the image display by the VolRender module requires that the memory associated with an image on the host be freed and reallocated each time a new image is produced. Consequently the address in memory to which the T3D process writes an image may shift as new rasters are produced. Therefore this address is part of the parameter structure mentioned above, and is re-read by the rendering engine prior to writing an image.

5 Results and Analysis

The target performance of this application requires that the data volume be projected onto a raster, and the resulting image updated on a user's display in coordination with the user's modification of the viewing parameters, at a rate sufficient to be termed interactive. This implies that any useful measurement of its performance must take

into consideration not only the time required to render an image but also the time required to transfer the image to the user's local display.

5.1 Compute Time

The computation time required to project the data volume onto a raster using a splatting approach is the primary reason why this project could not be successfully completed on workstation level computing equipment. Consider the rendering of a 256x256x64 voxel data set onto a 300x300 pixel raster. Memory limitations aside, this task requires 30 seconds to a minute on a Silicon Graphics Indigo^2 workstation. Note that the specialized polygonal rendering hardware of this machine is not useful to a direct volume rendering application such as this. Now consider the compute times shown in Table 1 for various rendering configurations on the Cray T3D.

Notice that the numbers along a row relate to a nearly constant workload per processor for varying numbers of processors, and that these numbers do not significantly increase as the number of processors is increased. Recall that the composition of the rasters on p processors involves the transfer of $p*(1-p)$ messages, each of which is $1/p$ times the size of the raster in length. As additional processors are added, larger numbers of smaller messages flood the interconnect network of the T3D. One might expect network contention via message collisions and consequently the rendering time to increase, as more processors are added. Clearly, at least for the workload configurations shown in the table, this does not seem to be the case. The minimization of collisions is likely due to the bidirectionality of the interconnect network switches, along with the dimension ordered message routing scheme.

Furthermore, notice that the numbers along a column relate to a varying workload per processor for a constant number of processors, and that these numbers increase by nearly a constant as the workload per processor is increased by a constant. As indicated previously, the amount of inter-PE IO is dependent upon the size of the raster and the number of processors only. Therefore the quantity of IO within a column is static. The fact that a constant increase in the rendering workload leads to a nearly constant increase in rendering time indicates that the rendering engine is compute bound, rather than IO bound. Were the latter true, it would indicate that this parallel implementation is not well suited to the architecture of the T3D system.

Finally, with respect to scaling the problem size, the rendering times clearly favor maintaining a constant 1:1 slice per PE ratio, while varying the number of processing elements. An intuitive expectation is that there should exist a point at which this ratio is no longer desirable. Such a situation may exist when the number of PEs is increased to the point that the time required for a message to travel the distance of the diameter of the network becomes a significant factor. However this number seems to be well beyond the 128 PE capacity of ARSC's T3D.

5.2 External IO Considerations

In addition, the distributed design of this application hints at several potential areas where bottlenecks can occur between the major system components. Specifically, bottlenecks are most likely to appear either in the transfer of an image between the T3D and Y-MP, or in the transfer of an image between the Y-MP and the display of the local host.

5.2.1 T3D - Y-MP Communication Time

Currently, the T3D system at the Arctic Region Supercomputing Center is operating under phase 1 of Cray Research's three phase external IO plan for the T3D product. Under phase 1, all IO requests from the T3D to devices external to the machine are handled through the T3D host system. The transfer of data and control between the T3D and its Y-MP host at ARSC occurs via two IO gateways. The limited number of gateways to processing elements (128 in the ARSC T3D) gives rise to the possibility of resource contention for the use of these gateways, and a subsequent bottleneck. Furthermore, as all IO requests must be serviced by the host system, such requests must compete with native processes for host CPU time.

Despite these issues, consider the transfer timings listed in Table 2. These timings show the total quantity of data transferred per second of wall-clock time. Note that the values within a column are reasonably constant, as the data transferred (pieces of the final raster) is not related to the size of the data volume on each PE. What variance between values in a column exists, is likely the result of fluctuations in available CPU time on the T3D host. Multiple entries per column are merely intended to provide an indication of the magnitude of these fluctuations, and their corresponding effect on the transfer times.

The maximum number of processors most users are able to access in interactive mode on the ARSC T3D is 32 or less. Recall that the current implementation of the rendering engine can produce up to 3 300x300 pixel (4 bytes per pixel) images per second. Clearly, at the 32 processor level, the transfer of these images between the T3D and its host is not the primary factor limiting the performance of this application.

5.2.2 Y-MP - Local Host Transfer Time

Perhaps the most significant potential for a bottleneck situation lies with the transfer of a rendered image between the T3D host system and the display of the user's local computer system. The magnitude of this bottleneck is obviously dependent upon the distance, usage, and form of the network between the two systems. Our experience here at ARSC with running this application across a heavily loaded 10Mbps local ethernet network, has been delay times between 0.5 and 116 seconds.

Table 1: Average rendering times (in seconds) for splatting 256x256 sli es onto a 300x300 raster.

		Number of PEs							
		1	2	4	8	16	32	64	128
Shaded	1	0.26	0.27	0.28	0.28	0.28	0.33	0.30	0.31
Slices	2	0.71	0.72	0.72	0.72	0.72	0.73	0.73	0.73
Per	3	0.97	1.00	1.00	1.00	1.01	1.01	1.02	1.02
PE	4	1.28	1.28	1.28	1.28	1.29	1.29	1.29	1.29

Table 2: T3D-YMP transfer time (in MB/second) for 360,000 bytes (a 300x300 raster).

		Number of PEs							
		1	2	4	8	16	32	64	128
Shaded	1	36.32	5.51	8.47	5.76	5.19	3.07	0.47	0.33
Slices	2	36.59	6.26	8.47	6.63	5.19	3.19	0.96	0.50
Per	3	36.17	5.53	8.37	6.39	5.44	1.47	1.27	0.28
PE	4	36.77	5.55	8.20	6.37	5.32	3.12	0.50	0.28

6 Conclusion

The VolRender rendering engine demonstrates that medical imaging on a Cray T3D is feasible. The addition of an AVS interface allows a user to interactively explore high resolution medical data sets. When connected to the T3D by FDDI, display rates of up to 5 frames per second give an interactive feel. An ethernet interface is also provided in part to serve remote users. While the display rates will not be interactive, it allows a remote user the ability to view data sets that are too large to render locally.

To achieve 10-15 images per second, a parallel implementation of shear-warp transform [6] or fourier projection-slice [8] will probably be necessary. These two algorithms give an order of magnitude speed up over the splatting algorithm on single processor workstations. One of these algorithms, displaying across HIPPI or FDDI, will provide interactive performance.

7 Acknowledgements

This research was supported by Cray Research Inc. and the Strategic Environmental Research and Development Program (SERDP) under the sponsorship of the Army Corps of Engineers Waterways Experiment Station. We would like to thank Dan Katz from Cray Research for his assistance with transferring data from the T3D to the Y-MP and Jim Snell of Texas A&M University for providing the dog leg volume data set.

References

[1] Cray Research Incorporated, Minnesota. *MPP Overview (TR-MPPOV)*, 1st edition, 1993.

[2] Cray Research Incorporated, Minnesota. *MPP Software Guide (SG-2508V)*, 1st edition, 1993.

[3] Cray Research Incorporated, Minnesota. *PVM and HeNCE Programmer's Manual (SR-2500/SR-2501)*, 3rd edition, 1993.

[4] James D. Foley, Andries van Dam, Steven K. Feiner, and John Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley, Reading, Mass., 2nd edition, 1990.

[5] Greg Johnson and Jon Genetti. High resolution interactive volume rendering on the cray t3d. In *1994 Fall Proceedings (Cray Users Group)*, pages 119–125, 1994.

[6] Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In Andrew S. Glassner, editor, *Computer Graphics (SIGGRAPH '94 Proceedings)*, pages 451–458, 1994.

[7] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In Maureen C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 163–169, July 1987.

[8] Takashi Totsuka and Marc Levoy. Frequency domain volume rendering. In James T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 271–278, August 1993.

[9] Lee Westover. Footprint evaluation for volume rendering. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 367–376, August 1990.