Interactive Volume Rendering on a Multicomputer

Ulrich Neumann

Department of Computer Science
University of North Carolina
Chapel Hill NC 27599

Abstract

Direct volume rendering is a computationally intensive operation that has become a valued and often preferred visualization tool. For maximal data comprehension, interactive manipulation of the rendering parameters is desirable. To this end, a reasonable target would be a system capable of displaying 128^3 voxel data sets at multiple frames per second. Although the computing resources required to attain this performance are beyond those available in current uniprocessor workstations, multicomputers and VLSI rendering hardware offer a solution. This paper describes a volume rendering algorithm for MIMD message passing multicomputers. This algorithm addresses the issues of distributed rendering, data set distribution, load balancing, and contention for the routing network. An implementation on a multicomputer with a 1D ring network is analyzed, and extension of the algorithm to a 2D mesh topology is described. In addition, the paper presents a method of exploiting screen coherence through the use of VLSI pixel processor arrays. Though not critical to the general algorithm, this rendering approach is demonstrated in the example implementation where it serves as a hardware accelerator of the rendering process. Commercial graphics workstations use pixel processors to accelerate polygon rendering; this paper proposes a new use of this hardware for accelerating volume rendering.

1. Introduction

Direct volume rendering is the common name that describes the viewing of volume data as a semi-transparent cloudy material. Its advantages are that much or all of the volume may be visible to the observer at one time; there is no need to introduce intermediate geometry that doesn't really exist in the data. We assume the input data is a scalar field sampled at the vertices of a 3D rectilinear lattice - a situation often encountered in medical and simulation data. Plate 1 is an image of a representative medical data set of dimensions 128x128x124. The following 3-step conceptual model of the volume rendering process is based on previously published derivations [Blinn82] [Kajiya+84]. Much of this example comes from Wilhelms and Gelder [Wilhelms+91].

1 - Reconstruct the continuous 3D scalar function **F** by convolving each sample point **f** with a reconstruction filter kernel **K**.

$$\mathbf{F}(x,y,z) = \sum_{x,y,z} \mathbf{f}_{x,y,z} * \mathbf{K}$$

2 - Apply an opacity **O** and shading **S** function to the continuous scalar field. These user definable transfer functions yield a differential opacity $\Omega = \mathbf{O}(\mathbf{F})$ and color emittance $\mathbf{E} = \mathbf{S}(\mathbf{F})$ at each point in the volume as a function of the scalar field properties at that point. The Ω and **E** fields should then be low-pass filtered for resampling in the next step.

3 - Integrate an intensity and transparency function along sample view-ray paths through the volume. The integrals may be taken toward or away from the viewer. When taken towards the viewer, the accumulated intensity **I** and transparency **T** along the sample ray is

$$\mathbf{I}(p) = \mathbf{T}(p)\int_0^p \frac{\mathbf{E}(v)}{\mathbf{T}(v)}\,dv, \quad \text{where } \mathbf{T}(p) = e^{-\int_0^p \Omega(v)\,dv}$$

The intensity equation has an analytic solution if we assume **T** and **E** constant over the interval [0,p]. By applying this constraint over limited size intervals, we may approximate the intensity and transparency of any interval. Successive intervals are composited to obtain the cumulative intensity or color reaching the viewer along the ray.

There are four common algorithmic approaches to approximating the above three step process in actual implementations.

1.1. Ray-casting - The volume is resampled along view rays [Levoy88] [Sabella88] [Upson+88]. The Ω and **E** functions must be reconstructed at the new sample points along the rays. Typically, 3D reconstruction is done by trilinear interpolation of the Ω and **E** function values evaluated at the lattice vertices. Successive samples along a ray are composited to produce the final ray color.

1.2. Serial Transformations - An affine view transformation is decomposed into three sequential 3D shear operations. Each shear is affected by a 1D transformation of the form $x' = Ax + B$ [Drebin+88] [Hanrahan90]. Since these

transformations require only a 1D reconstruction filter, cubic splines are commonly used to facilitate the resampling. The resampled volume is screen-aligned and ready for integration and compositing.

1.3. Splatting - This approach computes the effect of each voxel on the pixels near the point to which it projects. Slices of voxels are sorted by depth order and reconstructed by convolution with a 2D filter kernel. The reconstructed function is resampled and accumulated on a screen-aligned grid. Successive slices are composited to produce the final image. Since the filter is position-invarient for affine transformations, software table methods are often used to quickly approximate it [Westover89].

1.4. Cell Projection - Volumes are decomposed into polyhedra whose vertices are derived from the sampled data lattice [Shirley+90] [Max+90] Wilhelms+91]. The polyhedra are converted to polygons by projecting them under the view transformation. Reconstruction is done to obtain each polygon's vertex values for the opacity and emission functions. The resulting polygons are rendered by conventional means using a painter's algorithm and alpha compositing. These methods make effective use of existing polygon rendering hardware. The reconstruction functions are usually linear since most rendering hardware does linear interpolation of the polygon vertex values.

2. Rendering Hardware

The rendering method proposed here is a parallelized splatting approach. Using multiple processors with parallel frame buffer access, a splat kernel is produced and merged into an image at many pixels simultaneously. Current graphics workstations [SGI] use multiple processors to allow parallel access of pixel values in a frame buffer. Such groups of processors with parallel frame buffer access are what we refer to with the term *pixel processors*.

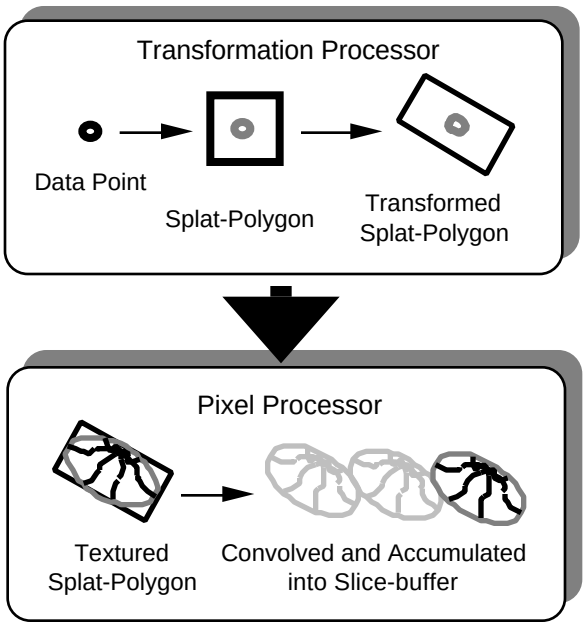

Fig. 1 - Splatting with hardware for textured polygons

This idea of using pixel processors to accelerate volume rendering is not totally new. Cell projection methods were created to make use of it. Laur and Hanrahan approximate splat filter kernels with groups of polygons rendered by dedicated hardware [Laur+91]. The new aspect of the method proposed here is that of coercing the hardware to render a splat filter kernel directly as a single graphic primitive. The next sections describe two methods whereby pixel processor arrays create splat kernels for convolution with voxels.

2.1. Textured Kernel - The kernel primitive can be thought of as a polygon with a nonlinear interpolation function. Arbitrary interpolation functions may be defined as textures. Given an array of pixel processors capable of texture lookup and multiplication, the screen coherence of each splat can be exploited. Current generation graphics workstations have this capability, although they may not yet offer the firmware needed to exploit it [SGI]. The splat-polygon's color and opacity are those of the voxel it represents. The splat-polygon with its texture coordinates is transformed and rendered normally except for the additional processing required by the pixel processors (Fig. 1). The pixel processors compute the texture value based on the texture coordinates at each pixel. This texture is the kernel function which is used to weight the polygon color and opacity. The convolution results at each pixel are accumulated in a slice buffer [Westover89]. When a complete slice of voxels is splatted, the pixel processors composite the slice buffer into the image.

In lieu of texture lookup capability, a kernel may be computed. This latter approach is most appropriate for Pixel-Planes 5, the target machine for the implementation described here. Before detailing the kernel computation algorithm, the next section briefly describes some essential aspects of this machine.

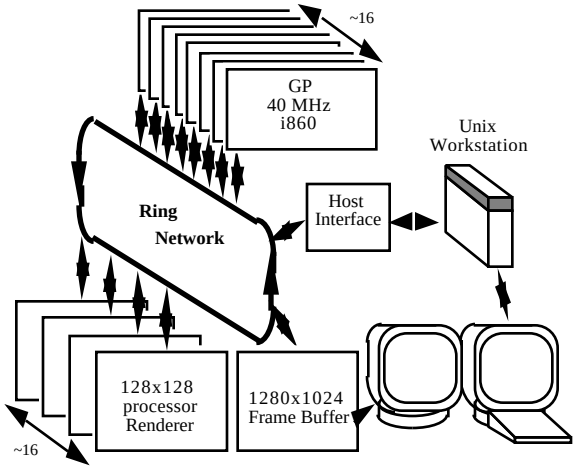

Fig. 2 - Pixel-Planes 5 system components

2.2. Pixel-Planes 5 Overview - This machine has multiple i860-based Graphics Processors (GPs), and multiple SIMD pixel processor arrays called Renderers (Fig. 2). Each Renderer is a 128x128 array of pixel processors capable of executing a general purpose instruction set. GPs send Renderers opcode streams which are executed in SIMD fashion. Renderers also have a Quadratic Expression Evaluator (QEE) that may be configured to occupy any screen

position [Fuchs+89]. Special QEE opcodes evaluate the function

$$Q = Ax + By + C + Dx^2 + Exy + Fy^2$$

at each processor in the Renderer for its unique x,y location. Configuring a Renderer to a new screen position is accomplished by offsetting the QEE so that each pixel processor's QEE result is based on its offset x,y location. The coefficients A - F are part of the instruction stream from the GPs. Renderers also have ports that allow data movement in and out of the processor array under GP control. The GPs, Renderers, a Frame Buffer, and workstation host all communicate over an eight-channel 1D ring-network whose aggregate bandwidth is 160 Mwords per second.

2.3. Computed Kernel and Slice Splatting - Polynomials are reasonable approximations to a gaussian filter kernel and easily computed. The function

$$2r^3 - 3r^2 + 1 \quad \text{where} \quad 0 < r < 1$$

is a low order gaussian approximation, but is somewhat awkward to compute directly. A quartic

$$Q^2 = (1 - r^2)^2$$

is a more practical solution since a quadratic term $Q = (1-r^2)$ may be computed directly by the QEE and later squared at all pixels in parallel.

```
for (xs = 0; xs < 4; xs++) {          /* cycle all 16 passes */
  for (ys = 0; ys < 4; ys++) {
    for (x = xs; x < slice_xsize; x += 4) {
      for (y = yz; y < slice_ysize; y += 4) {
        /* take every fourth voxel in x and y */
        GP computes QEE coefficients needed to produce Q at
                position S in the Renderer array;
        GP sends opcode and coefficients to Renderer which
                computes Q values in the pixel array;
   Renderer enables pixels with Q > 0;
    /* only enabled pixels participate in the next instruction */
   Renderer pixels save Q and load voxel color and
                opacity sent by GP;
    }}        /* end of pass */
    GP instructs Renderer to square the saved Q values at
                ALL pixels;                    /* one mult */
    GP instructs Renderer to scale the color and opacity
                by Q2;                         /* two mults */
    GP instructs Renderer to accumulate scaled color and
                opacity into slice buffer;     /* two adds */
}}          /* end of slice */
 GP instructs Renderer to composite slice buffer into image;
```

Fig. 3 - Pseudocode for splatting one slice using Pixel-Planes 5 Renderers

The kernel value (Q^2) at each pixel scales a voxel's color and opacity. Volumes are splatted a slice at a time by summing the scaled color and opacity values into a slice buffer and then compositing the slice buffer into the accumulated image. The squaring and scaling operations are expensive and should be factored out of the per-voxel inner loop. The kernels of adjacent voxels, however, typically overlap so we must square and scale more than once per data slice. By limiting the kernel radius to two inter-voxel distances, every fourth voxel in x and y will affect a disjoint set of pixels (Fig. 4). Therefore, in one pass we can splat one-sixteenth of the voxels in a slice before squaring, scaling, and accumulation into the slice buffer must be done. (A kernel radius of about 1.6 seems to yield the best overall image.) Pseudocode to implement the slice splatting process on Pixel-Planes 5 is given in figure 3.

When all the voxels in slice i are splatted, the accumulated slice color and opacity at each pixel are composited behind the current image of i-1 slices to produce an image of i slices. The compositing operation is efficiently done in parallel for each pixel of the array.

$$\text{Alpha}_i := \text{Alpha}_{i-1} + \text{Alpha}_{slice} * (1 - \text{Alpha}_{i-1})$$
$$\text{Color}_i := \text{Color}_{i-1} + \text{Color}_{slice} * (1 - \text{Alpha}_{i-1})$$

For arbitrary rotations, different slice orientations are used and the kernels are made elliptical to preserve the independence of pixels during each pass (Fig. 4). For affine projections, the elliptical shape is constant for all voxels making the D, E, and F quadratic coefficients constant over the whole frame. In this case, a linear expression evaluator (LEE) is all that is needed on the pixel processor since the D, E, and F terms may be computed once per frame at each pixel and added to each data point's linear term. This, however, exacts a small performance penalty, so in this implementation the available QEE was used.

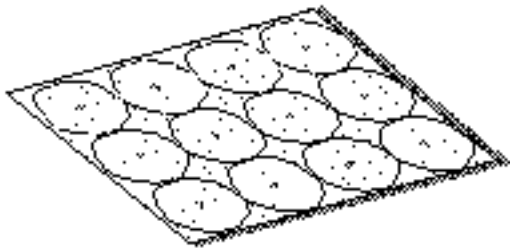

Fig. 4 - Elliptical kernel extents for one pass showing independence of every fourth voxel in x and y

The elliptical kernel coefficients are computed from the scaling and rotation portions of the view transformation **V**. We first scale **V** to account for the kernel radius T, specified as the number of inter-voxel distances.

$$[\mathbf{M}] = T[\mathbf{V}]$$

Now the pixel coordinates <x, y, z> must be transformed back to the data space coordinates <i, j, k> where the computed kernels are always radially symmetric, of unit radius, and therefore, the kernels of each pass are non-overlapping.

$$\left[M^{-1}\right]\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} i \\ j \\ k \end{bmatrix}$$

Let k = 0 always. Solving for z produces z = ax + by where

$$a = \frac{M_{31}}{M_{33}} \quad \text{and} \quad b = \frac{M_{32}}{M_{33}}$$

Now we can express data coordinates <i, j> in terms of pixel coordinate <x, y>.

$$[P]\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} i \\ j \end{bmatrix}$$

where $[P] = \begin{bmatrix} M_{11} + a\,M_{13} & M_{21} + a\,M_{23} \\ M_{12} + b\,M_{13} & M_{22} + b\,M_{23} \end{bmatrix}$

The kernel function is $Q^2 = (1 - r^2)^2$, but we render

$$Q = (1 - r^2) = 1 - [(i - i_0)^2 + (j - j_0)^2]$$

where i_0 and j_0 define the center of the kernel. Using **P**, we transform Q into a function of pixel coordinates <x, y>, and after some algebra we obtain the coefficients that allow the QEE to directly evaluate Q in the Renderer.

$$A = 2[x_0 (P_{11}^2+P_{21}^2) + y_0 (P_{11}P_{12}+P_{21}P_{22})]$$

$$B = 2[y_0 (P_{12}^2+P_{22}^2) + x_0 (P_{11}P_{12}+P_{21}P_{22})]$$

$$C = 1 - x_0 (P_{11}+P_{21}) - y_0 (P_{12}+P_{22})$$

$$D = - P_{11}^2 - P_{21}^2$$

$$E = -2(P_{11}P_{12}+P_{21}P_{22})$$

$$F = - P_{12}^2 - P_{22}^2$$

Note that D, E, and F do not depend on the kernel position <x_0, y_0>. Given any kernel position and allowing for precomputation of the view dependent terms, computing A or B requires only two multiplications and one addition. Computing the C coefficient requires two multiplications and two additions.

3. Multicomputer Rendering Algorithm

Parallel volume rendering algorithms must cope with potentially moving massive amounts of data every frame. For this reason, optimizing the distribution of data among the memory spaces in the machine is important. Full data replication is a trivial option deemed too expensive for most cases. Partial replication is often necessary or desirable. Data subsets may be *slabs* or *blocks*, *packed* or *interleaved*, and *static* or *dynamic*. The proposed algorithm makes use of a static, interleaved, slab distribution. It is static because each voxel is assigned a home node (or nodes) where it remains. It is interleaved since each node has several subvolumes of data, slices to be exact, that are not adjacent to each other. Slices are identified as slabs since they extend to the volume boundaries in two dimensions. This distribution is simply achieved by assigning slices to nodes in a round-robin fashion. It was chosen for load balancing and memory limitation reasons. If packed slices (a single slab) were used, there exists a strong possibility of the outer slice sets having less non-zero data to render than the inner slice sets. By interleaving slices, the spatial distribution of data at each node is similar.

The memory space issue arises from the need to buffer an entire slice's Renderer instruction stream as well as store three sets of slice data. This distribution stores three copies of the data set since we need slice sets oriented perpendicularly to each of the data axes i, j, and k. The set of slices most parallel to the view plane are used when traversing the data set.

The proposed algorithm attempts to maximize the utilization of Renderers and GPs without requiring an executive processor. Although this implementation uses special hardware Renderer nodes, rendering could be performed by general purpose processors. In fact, the latter would offer freedom in allocating GP or Renderer tasks as necessary to achieve optimal performance. The algorithm makes use of image parallelism by assigning each Renderer to a unique 128x128 pixel screen region. Renderers receive splatting instructions only for voxels that project to their region. Voxels near region boundaries are splatted at two or four Renderers to eliminate seams in the image. Since compositing must proceed in front-to-back or back-to-front order, Renderers must receive slices in sequence.

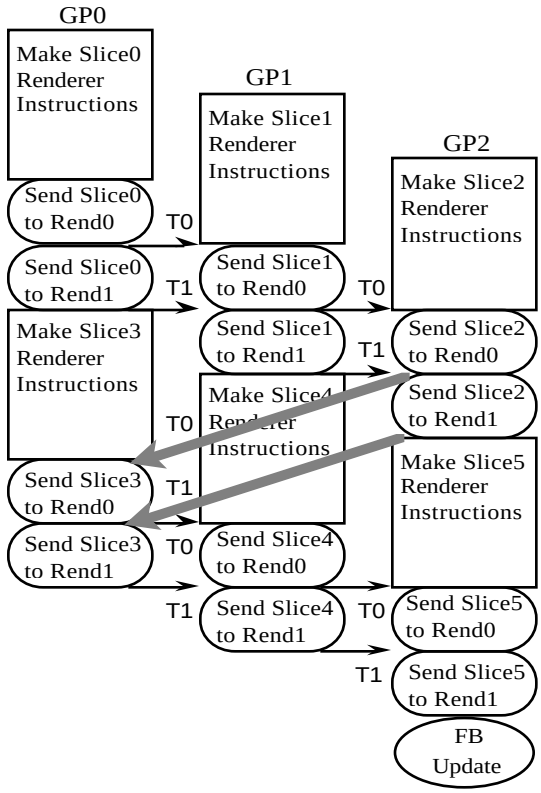

Fig. 5 - Rendering a Frame

Figure 5 illustrates three GPs and two Renderers computing a frame of a six slice data set. At the start of a frame, each GP shades and transforms their front-most slice. Phong shading is accomplished via a lookup table indexed by the voxel's gradient vector. An affine transformation is performed by DDA methods requiring only three adds per point after setup. Renderer instructions for splatting the shaded and transformed voxels are sorted by screen regions and placed into separate buffers. A token for each Renderer is circulated among the GPs to indicate permission to send splat instructions to that Renderer. Initially, all renderer tokens originate at the GP with the front-most slice. The T0 and T1 arcs in figure 5 represent the tokens for Renderer0 and Renderer1 respectively. Upon receipt of a token, a GP transmits the splat instructions for that Renderer's region. The token is then passed to the GP with the next slice. The circulating tokens ensure that Renderers receive slices in front-to-back sequence. The tokens also allow multiple GPs to simultaneously transmit instructions to different Renderers. When all the tokens have passed through a GP, it computes the Renderer instructions for its next slice. The GP with the last slice is responsible for instructing the Renderers to transmit their final color values to the frame buffer.

3.1. Mesh Topology Extension - Since large mesh topology machines are being built and commercially offered, it is of interest to note that this general approach does extend to them. Extension to square N x N mesh topologies requires that at least one edge of the mesh be connected to N Renderers. The data slices are assigned to mesh nodes sequentially row by row (Fig. 6). Tokens (N of them) are circulated through the nodes as before. To avoid contention, we specify that manhattan-style routing is performed for all Renderer messages; messages travel as far as possible in the direction sent, and then, if needed, with one turn they head to their destination. In effect, we utilize the mesh as a cross-bar interconnect. With some inspection it should become apparent that, if tokens move in-step, Renderer splat instructions will never compete for a communication link; routing hardware will always be able to forward messages.

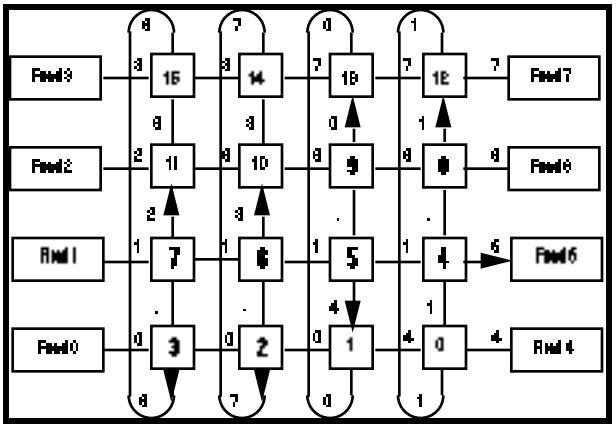

Fig. 6 - Data distribution, Renderer region assignment, and message paths for a mesh topology

It should be noted that although the number of nodes increases as $O(N^2)$, the number of Renderers increases only as O(N). A scheme that accommodates a limited N range is to place Renderers along both sides of the array and allow 2N tokens to circulate. In a vertically wrapped mesh, all 2N Renderers can receive instructions simultaneously without incurring contention for the network. Figure 6 shows sixteen nodes and eight renderers with eight tokens in circulation. Nodes with arrow arcs leaving them (nodes 2 - 9) have tokens and are transmitting Renderer instructions. Utilized paths are marked with the message's destination Renderer number.

A practical issue inhibiting this and other implementations of this sort, is the general difficulty of constructing distributed frame buffers and their consequent commercial unavailability.

4. Performance

To understand the behavior of this system, we first analyze the performance of each system element. Then we look at the load balance between the elements and how that affects the overall system performance.

4.1. Renderer Performance - The Renderers digest six instruction words per splat point and compute Q to ten bit precision in 77 cycles of a 40 MHz clock. The squaring and scaling operations use about 900 cycles per pass. Compositing at the end of each slice requires about 700 cycles. For a 128^3 data set, the pass and slice overhead totals 1,932,800 cycles for each Renderer, or about 50 ms. Based on these cycle counts, splatting 128^3 voxels on one Renderer should take 4.09 seconds including the 50 ms overhead. Experimental data correlates well with this predicted Renderer performance. Using twenty GPs and one Renderer, a 128^3 cube of voxels is splatted in 4.38 seconds. This corresponds to a Renderer throughput of about 478,000 voxels per second. Use of multiple Renderers distributes the voxel load while increasing only the GP token passing overhead. With four Renderers, the same 128^3 voxels are splatted in 1.29 seconds, or equivalently, a combined Renderer throughput of 1.622 Mvoxels per second.

4.2. GP Performance - The GP cost of splatting a voxel is the view transformation followed by four additions and six multiplies to compute the QEE coefficients. The transformed voxel must also be sorted by screen region so that the six instruction words to splat it are placed in the proper Renderer's buffer. For a 128^2 slice, a GP processes 16,384 voxels into 98,304 buffered instruction words in 0.16 seconds, achieving a computation throughput of about 102,000 voxels per second. As tokens arrive, the buffered instructions are transmitted to the Renderers. Pixel-Planes 5 GPs use specially addressed read cycles to move data to the ring. This scheme achieves >30 Mword per second peak throughput into the transmit FIFO. The ring requires about 5 milliseconds to transmit this data. Message software overhead adds roughly 4 milliseconds for the 192 message packets transmitted, hence a GP is able to transmit a slice's buffered Renderer instructions in under 10 milliseconds assuming no ring or Renderer contention.

In most volume data sets, many voxels are transparent and therefore no Renderer splat instructions are generated for them. Passes or slices that contain only transparent voxels produce no Renderer instructions for splatting or overhead

operations. Figure 7 shows performance statistics for the 128x128x124 data set shown in Plate 2. About 32% of the voxels (664,486) are non-transparent and actually rendered. The image size is determined by the number of Renderers. Four Renderers produce a 256x256 image while nine and sixteen Renderers produce 384x384 and 512x512 images respectively. It is unusual, but with this sort of hardware larger images render faster because of the increased Renderer parallelism.

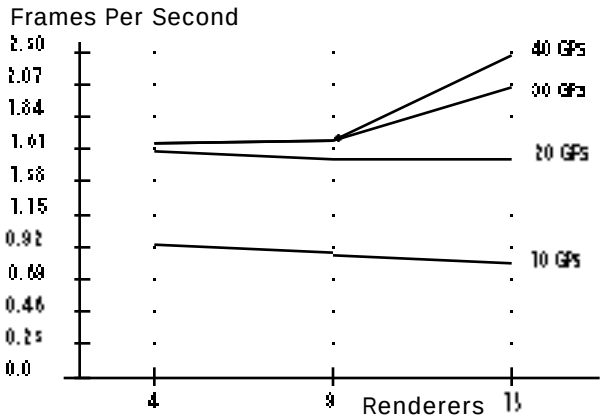

Fig. 7 - Renderer-Bound Performance

4.3. Load Balance - In heterogeneous systems load balancing is difficult. Computing resources are not interchangeable and therefore can not be shifted (without swapping boards) as needed to the task most burdening the system. In this implementation, either Renderers or GPs can limit system performance.

Figure 7 illustrates the case where performance is limited by the number of Renderers. This is often the case if there are many non-transparent voxels to be splatted. Adding more than twenty GPs has minimal effect unless the number of Renderers is increased above nine. In the case of thirty GPs and four Renderers, GPs are waiting over 0.5 seconds total for their first Renderer tokens after they have finished processing slices. Figure 8 illustrates a Renderer-bound frame with three GPs, two Renderers, and a six slice data set. The shaded areas are wasted GP waiting time. Circulating tokens are shown as arcs and marked T0 and T1 for their respective Renderers.

When a very high percentage of voxels are transparent, the system behavior changes. This occurs in the case of isosurface extraction. Figure 9 shows performance statistics for a 128x128x128 data set where about 11% of the voxels (238,637) are non-transparent and actually rendered. Here, the GP slice traversal time dominates the system performance. With so few voxels actually getting splatted, using more than nine Renderers has no appreciable benefit. Figure 5 illustrates a GP-bound frame with three GPs, two Renderers, and a six slice data set. The shaded arcs represent idle Renderer time. Tokens have been sent to GP0 where they must wait for the traversal of the next slice to complete before Renderer instructions can be sent.

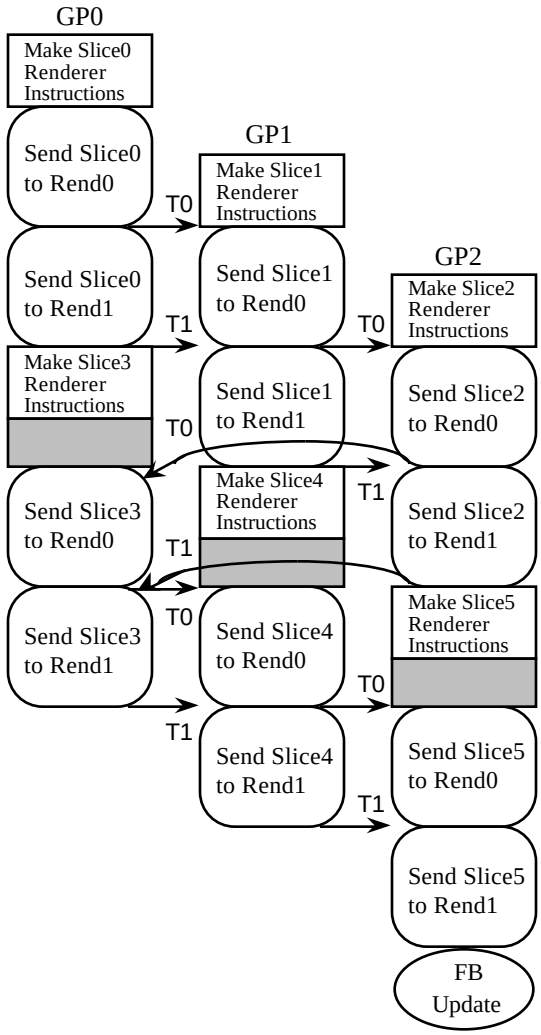

Fig. 8 - Renderer-Bound Frame

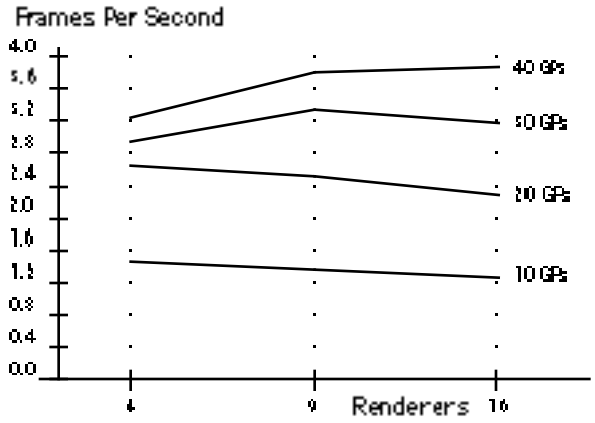

Fig. 9 - GP-Bound Performance

Finally, it should be observed that the limiting resource utilization does not approach 100% in either the GP-bound or Renderer-bound test case. Utilization peaks only in the

unlikely situation where each GP's slices are an identical workload and the Renderers are each hit by the same number of voxels every slice. Since the system resynchronizes every slice due to the token passing, each slice has it's own load balance. The overall behavior of the system could resemble GP-bound, yet a number of slices in a frame may actually be Renderer-bound, thereby lowering the GP utilization. In most cases then, neither the GPs nor Renderers are fully utilized. This is unfortunately symptomatic or many parallel algorithms - computing resource utilization often decreases as parallelism increases.

4.4. Progressive Refinement - The interactive response of a system can often be increased at the expense of image quality. Usually this is done by undersampling somewhere during the rendering process. Using the pixel processor approach, there is no advantage to undersampling in screen space and then interpolating the remaining pixels; in fact, as pointed out before, frame rate often increases as the number of Renderers, and therefore screen pixels, increases. Instead we may undersample the volume itself. The undersampling may be adaptive [Laur+91] or a regular skipping of some fraction of voxels. The latter is simple to implement by rendering every other voxel in all directions of the data set. A 128^3 data set is, for example, effectively rendered as 64^3. While the speed up varies due to load balance issues and relative frame overhead, this technique usually yields at least a factor of five. As an example, consider the small system of ten GPs and four Renderers that produces the image in Plate 2 at 0.93 Hz. Undersampling the volume as a 64x64x62 data set produces a sightly blurred image at a rate of 6.15 Hz; a speed up of over 660%. Undersampled image quality can be improved by rendering a separate prefiltered data set instead of simply skipping voxels.

5. Summary and Discussion

This paper presented a distributed algorithm for volume rendering on multicomputers along with two methods for using a pixel processor array to accelerate splatting. The algorithm was implemented on a 1D ring topology and its extension to a 2D mesh topology was outlined. The splat acceleration technique was demonstrated on a processor array with QEE capability. An alternative approach using texture table lookup was proposed for other pixel processor array architectures.

This implementation is not presented as *the* fastest or best way of doing volume rendering, but as a promising *alternative* approach whose merits are system and application dependent. The algorithm was implemented on Pixel-Planes 5 since that machine was available and was the inspiration of the pixel processor rendering idea to begin with. This system has no less than five different parallel volume rendering approaches implemented on it at this time. It is a credit to its designers that this is so, since volume rendering was never an explicit design consideration. The algorithm is implemented in C; some of the low level communications library routines were crafted in i860 assembly code.

Many issues remain for consideration in future work. The errors produced by view-dependent filter kernels need further analysis. Faster pixel processor arrays are desirable, perhaps with nibble or byte-wide data paths. The textured filter kernel method should be explored on a suitable machine. Other parallel algorithms that cope with load imbalance and offer adaptive processing savings should be investigated. The algorithmic impact of different data distributions should also be studied - particularly dynamic distributions in which data migrates among the GPs as the view changes [Neumann91].

6. Acknowledgements

John Eyles provided a special Renderer splat instruction which is appreciated. Many persons at UNC participated in useful discussions that influenced this work. Tim Cullip's ideas about parallel algorithms were particularly inspiring. The following agencies helped support this work:

NSF, DARPA/ISTO: NSF Grant # MIP-860 1552. DARPA order #6090. “Curved Surfaces and Enhanced Parallelism in Pixel-Planes”.

Corporation for National Research Initiatives: “VISTAnet: A Very High Bandwidth Prototype Network for Interactive 3D Medical Imaging”, A collaborative project among BellSouth, GTE, Microelectronics Center of NC, and UNC at Chapel Hill.

National Cancer Institute: NIH grant #1 PO1 CA47982-01. “Medical Image Presentation”.

7. References

[Blinn82] Jim Blinn Light Reflection Functions for Simulation for Clouds and Dusty Surfaces. Proceedings of SIGGRAPH '82, Computer Graphics 16, 3. July 1982, pp. 21 - 29.

[Drebin+88] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume Rendering. Proceedings of SIGGRAPH '88, Computer Graphics 22, 4. August 1988, pp. 65 - 74.

[Fuchs+89] Henry Fuchs, John Poulton, John Eyles, Trey Greer, Jack Goldfeather, David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs, Laura Israel. A Heterogeneous Multiprocessor Graphics System Using Processor Enhanced Memories. Proceedings of SIGGRAPH '89, Computer Graphics 23, 4. August 1989, pp. 79 - 88.

[Hanrahan90] Pat Hanrahan. Three-Pass Affine Transforms for Volume Rendering. Proceedings of the San Diego Workshop on Volume Visualization, Computer Graphics 24, 5. November 1990, pp. 71 - 78.

[Kajiya+84] Jim Kajiya and Brian Herzen. Ray Tracing Volume Densities. Proceedings of SIGGRAPH '84, Computer Graphics 18, 3. July 1984, pp. 165 - 174.

[Laur+91] David Laur and Pat Hanrahan. Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering. Proceedings of SIGGRAPH ‘91, Computer Graphics 25, 4. July 1991, pp. 285 - 288.

[Levoy88] Marc Levoy. Display of Surfaces From Volume Data . IEEE Computer Graphics and Applications 8, 3. March 1988, pp. 29 - 37.

[Max+90] Nelson Max, Pat Hanrahan, and Roger Crawfis. Area and Volume Coherence for Efficient Visualization of 3D Scalar Functions. Conference Proceedings, San Diego Workshop on Volume Visualization. Computer Graphics 24, 5. November 1990, pp. 27 - 33.

[Neumann91] Ulrich Neumann. Taxonomy and Algorithms for Volume Rendering on Multicomputers. Tech Report TR91-015, Dept. of Computer Science, UNC at Chapel Hill, January 1991.

[Sabella88] Paolo Sabella. A Rendering Algorithm for Visualizing 3D Scalar Fields. Proceedings of SIGGRAPH '88, Computer Graphics 22, 4. August 1988, pp. 51 - 58.

[Shirley+90] Peter Shirley and Allen Tuchman. A Polygonal Approximation to Direct Scalar Volume Rendering. Conference Proceedings, San Diego Workshop on Volume Visualization. Computer Graphics 24, 5. November 1990, pp. 63 - 70.

[Shroder+91] Peter Shroder and James B. Salem. Fast Rotation of Volume Data on Data Parallel Architectures. IEEE Visualization'91. pp. 50 - 57.

[SGI] Silicon Graphics Power Series Technical Report. Doc # 1000027.

[Upson+88] Craig Upson and Michael Keeler. V-Buffer: Visible Volume Rendering. Proceedings of SIGGRAPH '88, Computer Graphics 22, 4. August 1988, pp. 59 - 64.

[Westover89] Lee Westover. Interactive Volume Rendering. Conference Proceedings, Chapel Hill Workshop on Volume Visualization. May 1989, pp. 9 - 16.

[Westover91] Lee Westover. Splatting - a Feed Forward Approach to Volume Rendering. Ph.D. Dissertation. Dept. of Computer Science, UNC at Chapel Hill. TR91-?? July 1991.

[Wilhelms+91] Jane Wilhelms and Alan Van Gelder. A Coherent Projection Approach for Direct Volume Rendering. Proceedings of SIGGRAPH '91, Computer Graphics 25, 4. July 1991, pp. 275 - 284.