NASA Contractor Report 191158

Feed-Forward Volume Rendering Algorithm for Moderately Parallel MIMD Machines

Roni Yagel
Ohio State University
Columbus, Ohio

July 1993

Prepared for
Lewis Research Center
Under Grant NGT–26–027–011

National Aeronautics and
Space Administration

FEED-FORWARD VOLUME RENDERING ALGORITHM FOR MODERATELY PARALLEL MIMD MACHINES

Roni Yagel
Ohio State University
Columbus, Ohio 43210

Abstract

In this report algorithms for direct volume rendering on parallel and vector processors are investigated.Volumes are transformed efficiently on parallel processors by dividing the data into slices and beams of voxels. Equal sized sets of slices along one axis are distributed to processors. Parallelism is achieved at two levels. Because each slice can be transformed independently of others, processors transform their assigned slices with no communication, thus providing maximum possible parallelism at the first level. Within each slice, consecutive beams are incrementally transformed using coherency in the transformation computation. Also, coherency across slices can be exploited to further enhance performance. This coherency yields the second level of parallelism through the use of the vector processing or pipelining. Other ongoing efforts include investigations into image reconstruction techniques, load balancing strategies and improving performance.

1.0 Introduction

Representation by *spatial-occupancy enumeration* methods allows a simple yet versatile method of generation and display of three-dimensional objects. The most common spatial enumeration is obtained when a solid is decomposed into identical cells called *voxels* and arranged in a fixed, regular, rectilinear grid. Curvilinear and unstructured grid decompositions also exist and are being increasingly used. To ascribe visual properties, spatial occupancy functions and colors are assigned to voxels. *Binary* voxel [Herman79] and *variable-density* voxel techniques ([Drebin88], [Levoy88]) arise out of the use of binary and continuous spatial occupancy functions, respectively.

Several methods exist for the rendering of volumes. A class of these methods convert the voxel representation into surface and line primitives [Lorensen87]. However, methods of this class suffer from various disadvantages that mainly arise from the ambiguity of decision determining the exact position of the surface.

Direct methods have been developed to render volumes. These methods can be classified as *object order* or *image order*. Object order methods require the enumeration of all voxels of a solid and the determination of the affected pixel on a screen. Image order techniques,

on the other hand, determine all the voxels of a solid which affect a given pixel on the screen. Hybrid methods exist which rely on some intermediate representations. Levoy [Levoy 90a] has categorized and created a taxonomy of volume rendering methods. Westover [Westover90] employs the synonymous terms feed-forward and backward-feed for object and image order methods. We shall also employ these terms to classify volume rendering methods.

Feed-forward methods are also called projective methods. The viewing transformation matrix is applied to all voxels (enumerated in some order), thus providing an intermediate volume which is then projected to a two dimensional screen. Feed-forward methods are easy to implement and are comparatively inexpensive. However, such a direct application of the transformation matrix introduces artifacts in the image. Reconstruction or interpolation in three-dimensions can be performed to annul the effect of artifacts. Hanrahan proposed a method of decomposing the transformation matrix into a series of lower dimensional shears. Such a decomposition would allow for the easier supersampling operation along a single dimension [Hanrahan90]. Splatting [Westover90] is another technique employed to reduce the impact of artifacts in which each voxel is rendered as a group of pixels rather than a single pixel.

Backward-feed methods, which include ray-casting have been specifically developed for rendering volumes. The use of the template method [Yagel92] is one example of an efficient backward-feed method.

Hybrid Methods have been investigated ([Levoy90b], [Upson88]). In these methods the volumes are traversed in object or image order. The contribution of a set of voxels to a pixel is then computed in either image or object order.

Hidden volume elimination must be an integral part of the rendering methods and several standard methods used for hidden surface elimination can be employed fruitfully. Scanline methods have been used [Herman79]. However, *Z-buffer* and *front-to-back* [Reynolds87] and *back-to-front* [Frieder85] methods are more commonly used.

Shading is necessary for realistic images. Shading can be done in either object space or image space. Image space shading [Pommert89], also called deferred shading is comparatively inexpensive. *Depth shading*, *gradient shading*, *context-grading shading* are popular image shading methods [Yagel et al 92].

The sheer amount of data that arises even from a not too fine resolution volume representation is enormous. To reduce the total rendering time, several specialized hardware solutions have been proposed [Kaufman et al 90]. All these solutions suffer from a lack of generality. General purpose methods that can be executed on general purpose machines and specially on the increasing number of general purpose commercially available parallel processors.

Some implementations of volume rendering have recently been reported for a variety of parallel architectures. Some of them are specifically designed for experimental hardware - such as the Princeton Engine [Schroder and Stoll92] or DASH [Neih and Levoy92], while others are designed for commercially available massively parallel machines such as the

CM-2 ([Schroder and Salem91],[Schroder and Stoll92]), MasPar [Vezina et al92], Hypercube [Montani et al92], or nCube [Elvins 92].

Researchers also differ in the viewing algorithm they adopted. Several are based on feed-backward methods. In [Neih and Levoy92] and [Montani et al92] ray-casting [Levoy 88] is used, while in [Schroder and Stoll92] our template-based ray-casting [Yagel and Kaufman92] was implemented. Some researchers ([Schroder and Salem91], [Vezina et al92]) have adopted a feed-forward method such as shear-and-composite [Hanrahan90] while others [Elvins 92] implemented rotate-and-splat [Westover 91]. We developed an algorithm that follows a feed-forward rotate-and-draw paradigm. Unlike previous work we uncover several inherent parallelism schemes and demonstrate superior speed and linear scalability.

In this work we present algorithms that are amenable to efficient implementation on parallel and vector processors. In section 2.0 we develop an efficient method to transform volumes and show that it is amenable to parallel implementation. Section 3.0 presents a parallel feed-forward renderer. The final section contains some concluding remarks.

2.0 Parallel Transformation of Volumes

A volume represented by a rectilinear grid can be embedded in the 3-dimensional Euclidean space. If the grid lines are regularly spaced in all dimensions then the voxels are cubes. We denote a object space grid point v by a 4-tuple $[v_x , v_y , v_z , 1]$ in the homogenous 4-D homogenous space XYZ1 of resolution N.

2.1 Decomposition of Volumes

A *slice* is the set of grid points having one coordinate the same. A *z-slice at k*, is therefore the set

$$S_{z:k} = \{(x,y,z) / z = k,\ 0 \le k < n\ , 0 \le x, y < n\}.$$

Thus, all voxel planes perpendicular to the z-axis are z-slices. Similarly we can define *x-slices* and *y-slices*.

We define a *beam* to be the set of grid points with two of the three coordinates being fixed. Thus a *x-beam at j, k* is the set

$$B_{x:(j,k)} = \{(x,y,z) / y = j, z=k, 0 \le j, k < n,\ \ 0 \le x < n\ \}.$$

In a similar way we define *y-beams* and *z-beams*. A z-slice at k would therefore have x-beams parallel to the x-axis and y-beams parallel to the y-axis. Figure 1 shows the decomposition of a volume into beams and slices.

2.2 Spatial Order and Spatial Coherency

A voxel set can be thus divided into sets of beams and slices. There exists an inherent *spatial ordering* between consecutive slices determined by the viewing direction. Spatial ordering exists between beams in a slice and between voxels in a beam. The inherent spatial order of slices in a volume lends itself well to efficient hidden voxel elimination such as the back-to-front and front-to-back algorithms [Frieder85].

Spatial coherency exists between voxels in a beam and in a slice. This coherency stems from adjacency of grid points in object space. In the case of regular grids, grid points are separated from each other by a constant distance. This allows the efficient transformation of volumes through the use of constant incremental updates. The next section shows the use of regular spatial coherency to enhance computation performance.

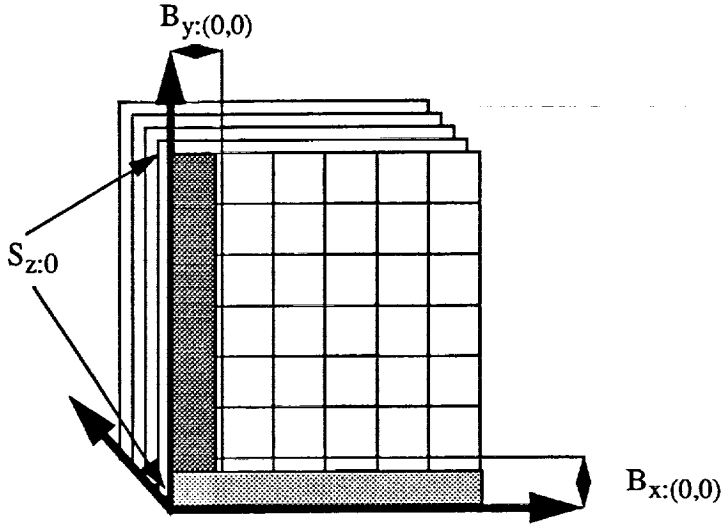

FIGURE 1. Decomposition of volumes into slices and beams. $S_{z:0}$ is the first slice of the volume, $B_{x:(0,0)}$ and $B_{y:(0,0)}$ are the first x and y beams in $S_{z:0}$.

2.3 Transformations of Volumes using Spatial Coherency

The word transformation here includes all viewing transforms, shape/position changing transformations, and inverse texture mapping transformations. All transformations are assumed to preserve the connectivity of grid points, grid resolution and grid spacing. A transformation of a volume consists of several matrix vector multiplies and is expressed by the matrix equation:

$$[v_x', v_y', v_z', 1] = [v_x, v_y, v_z, 1] \cdot M$$

where v and v' represent the position of grid points in a 4-D object and image spaces respectively and M is the transformation matrix of order 4:

$$\begin{bmatrix} m_{00} & m_{10} & m_{20} & 0 \\ m_{01} & m_{11} & m_{21} & 0 \\ m_{02} & m_{12} & m_{22} & 0 \\ m_{03} & m_{13} & m_{23} & 1 \end{bmatrix}$$

Application of the above multiplication to a data set of N^3 voxels requires a total of $16N^3$ floating point multiplications and $12N^3$ floating point additions. However, by using spatial coherency the number of operations and hence rendering time can be greatly reduced. It will be shown that a volume of resolution N^3 can be transformed with only $3N^3$ additions.

Suppose u and v are two adjacent grid points along x-beam at $B_{x:(j,k)}$ with v being enumerated after u, that is:

$$v_x = u_x + 1, v_y = u_y = j, v_z = u_z = k$$

After the transformation let u' and v' be the positions of the grid points u, v. The transformed point v' is first expressed in terms of v. In addition it true that

$$[v'_x, v'_y, v'_z, 1] = [u_x + 1, u_y, u_z, 1] \cdot M$$

Through simple algebra we can express the above product as the sum of two vectors. The right most vector is the first column of the transformation matrix:

$$[v'_x, v'_y, v'_z, 1] = [u_x, u_y, u_z, 1] \cdot M + [m_{00}, m_{10}, m_{20}, 0]$$

Finally,

$$[v'_x, v'_y, v'_z, 1] = [u'_x, u'_y, u'_z, 1] + [m_{00}, m_{10}, m_{20}, 0]$$

Each incremental update requires 3 additions. To begin the incremental process we need one matrix vector multiplication (16 multiplications and 12 additions) to compute the updated position of the first grid point. For the remaining N^3 grid points we need a total of only $3N^3$ additions. It should be noted that using the above approach we have both reduced the number of operations and also eliminated the need for floating point multiplications, which are more expensive than floating point additions.

2.4 The Vector Nature of Incremental Updates

It is not necessary that the same incremental scheme be used to transform the entire volume. The incremental computation can be organized into several phases. Each phase consists of updates in a particular direction. Figure 2 shows a particular scheme of incremental computation which has been implemented.

The incremental scheme can be divided into four phases

- ***Seed Phase:***

In this phase, a single grid point (usually the lowest point of the grid of the first slice k) is transformed through the use of a matrix vector multiply. This grid point forms the seed for consequent updates.

- ***Vertical Beam Phase***

Incremental computation is used to transform an entire beam. The first vertical beam, $B_{y:(0,\,k)}$ is updated using the seed. As shown in the previous section, the second column of the transformation matrix M is added to a previously enumerated grid point. The updates here are of a scalar nature.

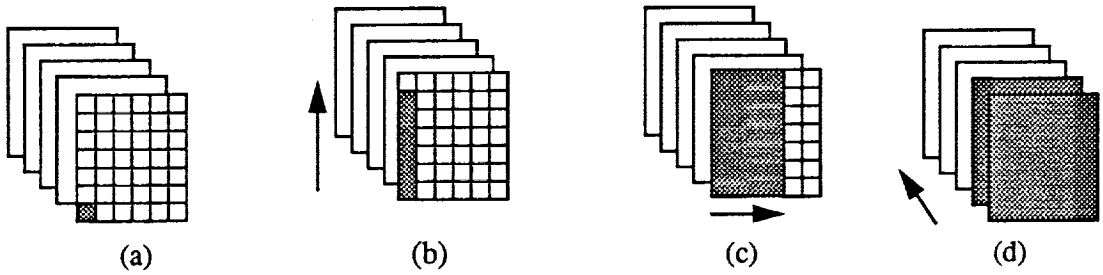

FIGURE 2. The four phases of incremental updates: (a) seed phase (b) vertical beam phase (c) horizontal beam phase (d) slice phase.

- ***Horizontal Beam Phase***

In the third phase beams $B_{x:(j,k)}$, $0 < j < n$ are updated. Beam $B_{y:(j,\,k)}$ is used to update beam $B_{y:(j+1,\,k)}$. The first column of matrix M is used for incremental updates, since the enumeration is along the x-axis. If each beam is considered to be a vector of length N, then the updates in this phase are essentially vector additions. A processor with vector capabilities can exploit the vector nature of the updates. At the end of this phase, slice $S_{z:k0}$ is completely transformed.

- ***Slice phase***

Updated slice $S_{z:k}$ is used to incrementally update the next slice $S_{z:k+1}$. The third column of M is used for updates in this phase (since the enumeration is along the z-direction). Once again the updates are essentially vector additions, except that the length of the vectors is N^2.

In summary, we have reduced the projection process from a set of matrix vector multiplication into a sequence of vector additions. The same scheme can be implemented for any other order of the axes, as commonly required by a back-to-front algorithm.

2.5 Parallel Transformation of Volumes

The decomposition of volumes into beams and slices and the use of incremental computation allows for an efficient parallel implementations. There exist two levels of parallelism to be exploited.

Slices can be processed independently by different processors without the need for any communication between processors. This is the first level of parallelism. Next, each processor can process its slices by incrementally updating the beams as shown in the previous section, thus exploiting the second level of parallelism. If a processor is assigned multiple slices, the coherency between different slices can also be exploited. If individual processors have either vector processing capabilities or pipelined functional units, the performance is even better as explained in section 2.4.

It is important to observe that all voxels including empty ones are transformed. Thus, medium to very dense volumes would benefit most from such a brute-force approach. Figure 3 shows schematically a parallel/vector implementation of transformations.

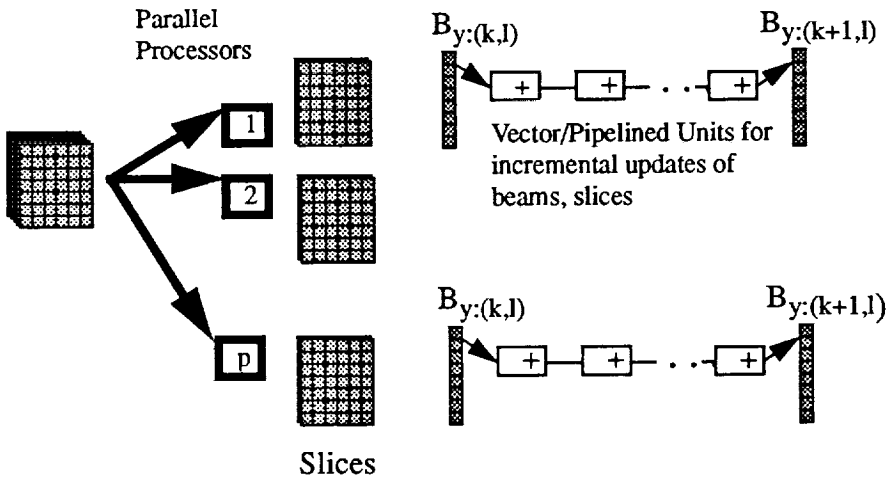

FIGURE 3. Transformation of Volumes on Parallel/Vector Processors

3.0 Parallel Feed-Forward Volume Rendering

The feed-forward algorithm for volume rendering implements the classical graphics pipeline for rendering which consists of the following stages:

- Creation of a viewing transformation from the viewing parameters.
- Transformation of the volume by the view matrix.
- Hidden voxel elimination to determine visible voxels.
- Display of the visible voxels by assigning colors to the appropriate screen-pixels.

In this section we report on a feed-forward renderer which allows the orthographic viewing of volume data using the ideas developed in section 2.0.

3.1 Parallel Implementation

The *functional parallelism* paradigm has been used to implement the graphics pipeline on parallel machines (or ensembles) where each part of the graphics pipeline is executed by a separate active process. An alternative way is to use the *single-program-multiple-data* (SPMD) model of parallelism, where each process executes the same code. The SPMD model allows a better utilization of the parallel ensemble of processors. We implemented our renders in this paradigm. Each active process is either assigned to processors deterministically or is assigned to the first available processor by the operating system.

---

procedure *feed_forward*
  Construct viewing matrix M
  Read subset of slices $S_{z:k}$, $k = k_0,...,k_1$
  for slices $S_{z:k}$, $k = k_0,...,k_1$ **do**
    $B_{y:(0,k)}[0] = [0, 0, k]\ M$ /* *seed phase* */
    for i = 1 **to** N-1**do** /* *vertical beam phase* */
      $B_{y:(0,k)}[i] = B_{y:(0,k)}[i-1] + [m_{01}, m_{11}, m_{21}]$
    endfor
    Draw $B_{y:(0,k)}$ on local image I_p
    for i = 1 **to** N-1**do** /* *horizontal beam phase* */
      for j = 0 **to** N-1**do**
        $B_{x:(i,k)}[j] = B_{x:(i-1,k)}[j] + [m_{00}, m_{10}, m_{20}]$
      endfor
      Draw $B_{x(j,k)}[0...N-1]$ on I_p
    endfor
  endfor
  Compose I_p with final image I_f
end procedure *feed_forward*.

FIGURE 4. The *feed_forward* algorithm employing parallelism and beam-coherency.

Each processor executes the graphics pipeline on the subset of slices assigned to it. The subset is transformed, and an image is created by each processor independently after performing hidden volume elimination on the set of assigned slices. Ideally all processors should be given equal number of slices to process. An equal distribution of slices would lead to a highly load-balanced operation, wherein all processors perform the same amount of work. However it is not always possible on all machines to explicitly assign the slices to processors. For instance, on the CRAY/Y-MP, parallel execution is based on the creation of processes and their allocation to processors, and the user has no control over the assignment of slices to processors.

The methods of section 2.0 are most efficient on parallel processors with vector nodes. Implementations on commercial machines like the CRAY/Y-MP, IBM Power Visualization System (PVS), Intel iPSC/2 (with vector nodes) perform significantly faster.

Each processor creates a local image which are then combined to obtain the final image. A local image can be created by employing the Z-buffer hidden voxel elimination or the back-to-front compositing methods. The actual method of combining depends on the viewing chosen and the interconnection network of the parallel ensemble. Shading is currently done in the image space on the combined image, by a single processor.

The algorithm in Figure 4 is executed on each processor of a parallel ensemble. We assume that the viewing matrix M is of order 4 and that beam B is an array of N triples (where the volume resolution is N^3). The subscripts for B describe the type and location of the beam.

In the above algorithm only three of the four incremental computation phases described in section two are implemented. The slice phase is not used This is the most general implementation which would work even when each processor is given one slice. In each z-slice, a seed is first found, then the first vertical beam is updated, and finally, after the horizontal beam phase, the entire slice is updated. The horizontal beam phase updates are executed in the vector mode. This version of the renderer is implemented on the CRAY/Y-MP and the Silicon Graphics.

In the above the algorithm, *Draw* performs hidden volume elimination. Currently we employ a Z-buffer algorithm towards this purpose. *Shade* implements currently *depth* shading and *depth gradient* shading.

3.2 Salient Features of the Parallel Feed Forward Approach

- Volume data is distributed only once to all processors.
- Changes in viewing parameters does not cause any movement of data between processors.
- Each processor is assigned equal number of slices to process if possible.
- All voxels are transformed, including the empty ones. This is done so that the computing pipeline in vector processors is never interrupted.
- The amount of data transferred between processors is of $O(N^2)$. Communication occurs only when the parallel architecture supports a memory hierarchy of local and shared memories. Communication is required in the compositing phase.
- Although Z-buffer algorithms can be used, it is preferable to employ back-to-front or front-to-back algorithms. The reason for this preference is the use of the brute force strategy that, as above, keeps the pipeline busy.

3.3 Example Implementations

3.3.1 CRAY/Y-MP

The first algorithm *feed_forward* (with no slice coherency) was implemented on the CRAY/Y-MP. On the Y-MP the multitasking facility called microtasking was employed to obtain concurrency at the slice level.

The outer loop in the above algorithm is divided into as many tasks or processes as the number slices. Each process would then be executed by an available processor. The immediate outcome of this strategy is that the program has no control over which slice will be assigned to which processor. Therefore, there is no way to exploit slice coherency but only beam coherency. To increase the granularity of the tasks, loop unrolling can be done. An immediate benefit from this would be the reduction in synchronization overhead and the exploitation of concurrency across slices. Loop unrolling has not been implemented yet.

The vector facilities of the Y-MP are used to achieve concurrency at the beam level. Hidden volume elimination was achieved through the use of a common Z-buffer, and depth shading was done on the final image by a single processor. Table 1 lists the transformation times for volumes of different sizes. It is noted that the incremental update algorithm scales well with the volume size.

Table 1. Transformation Time on a CRAY/Y-MP (8 processors)

N	Time (sec.)
64	0.089
128	0.175
256	0.696
512	5.27

3.3.2 Silicon Graphics Workstation

The implementation of the forward rendered on a multi-processor Silicon Graphics is actually a slightly diferrent from the CRAY implementations. The distribution of slices is explicit which allows us to assign consecutive slices to each processor and employ slice-cohèrency. However, since there is no local memory, all processors access one Z-buffer and the data is not distributed a-priori, like in the CRAY implementation. In addition, since the Silicon Graphics Workstation does not have vector processing capabilities,

speedup is gained only because we have eliminated multiplication and not because we compute in a vector mode. Table 3.0 contains the transformation times.

Transformation Times for SGI workstation for different ensembles

N	P=1	P=2	P=4
64	0.32	0.46	0.76
128	6.14	3.12	1.77
256	45.68	25.19	12.57

Acknowledgments

I would like to thank Raghu Machiraju for his help in the implementation of the CRAY vesion. This project was also supported by the National Science Foundation under grant CCR-9211288.

4.0 References

1. Drebin, R. A, Carpenter, L., Hanrahan, P., "Volume Rendering," *Computer Graphics*, Vol. 22, No. 4, August 1988, pp. 65-74.
2. Freider, G., Gordon, D., and R. Reynolds, "Back to Front Display of Voxel-Based Objects," *Computer Graphics and Applications*, Vol. 5, No. 1, January 1985, 52-60.
3. Goldwasser, S. M., Reynolds, R. A., Talton, D. A., Walsh, E. S., "High Performance Graphics Processors for Medical Imaging Applications," *Parallel Processing for Computer Vision and Display*, Dew, P.M., Earnshaw, R. A., and Heywood, T. R., (eds.), Addison-Wesley, 1989, pp. 461-470.
4. Hanrahan, P.,"Three-Pass Affine Transforms for Volume Rendering," *Computer Graphics*, Volume 24, No. 5, November 1990, pp. 71-77.
5. Herman, G. T. and Liu H. K., "Three-Dimensional Display of Human Organs From Computer Tomograms," *Computer Graphics and Image Processing*, Vol. 9, No. 1, January 1979, pp. 1-21.
6. Kaufman A., Bakalash, R., Cohen D. and Yagel R. "A Survey of Architectures for Volume Rendering", *IEEE Engineering in Medicine and Biology*, Vol. 9, No. 4, December 1990, pp.18-23.
7. Levoy, M., "Display of Surfaces from Volume Data", *IEEE Computer Graphics and Applications*, Vol. 8, No. 5, May 1988, pp. 29-37.
8. Levoy, M.,"A Taxonomy of Volume Visualization Algorithms," *SIGGRAPH, Course Notes on Volume Visualization Algorithms and Architectures*, August 1990, pp. 6-12.

9. Levoy, M., "A Hybrid Ray Tracer for Rendering Polygon and Volume Data," *IEEE Computer Graphics and Applications*, Vol. 10, No. 3, March 1990, pp. 33-40.

10. Lorenson, W. E., Cline, H. E.,"Marching Cubes: A High resolution 3D Surface Construction Algorithm," *Computer Graphics*, Vol. 21, No. 4, July 1987, pp. 163 - 169.

11. Montani C., Perego R., and Scopigno R., "Parallel Volume Visualization on a Hypercube Architecture", *Proceedings of the 1992 Workshop on Volume Visualization*, Boston, MA, October 1992, pages 9-16.

12. Neih J., and Levoy M., "Volume Rendering on Scalable Shared-Memory MIMD Architecture", *Proceedings of the 1992 Workshop on Volume Visualization*, Boston, MA, October 1992, pages 17-24.

13. Pommert, A., Tiede, U., Wiebecke, G., Hoehne, K. H., "Image Quality in Voxel-Based Surface Shading," *Proceedings of the International Symposium on Computer Assisted Radiology*, CAR'89, Berlin, West Germany, 1989, 737-741.

14. Reynolds, R., Gordon, G., Chen, L.,"Display of Slice-Represented Objects," *Computer Vision, Graphics, Image Processing*, Vol. 38, No. 3, June 1987, pp. 275-298.

15. Schroder, P., Salem, J. B.,"Fast Rotation of Volume Data on Data Parallel Architecture", *Proceedings of Visualization'91*, San Diego, CA, IEEE Computer Science Press, October 1991, pages. 50-57.

16. Schroder, P., and Stoll G., "Data Parallel Volume Rendering as Line Drawing", *Proceedings of the 1992 Workshop on Volume Visualization*, Boston, MA, October 1992, pages 25-32.

17. Upson, V., Keeler, M.,"V-Buffer: Visible Volume Rendering," *Computer Graphics*, Vol. 22, No. 4, August 1988, pp. 59-64.

18. Westover, L.,"Footprint Evaluation for Volume Rendering," *Computer Graphics*, Vol. 24, No. 4, August 1990, pp. 367-376.

19. Westover L.,"Splatting: A Parallel, Feed-Forward Volume Rendering Algorithm", Ph.D Thesis, Department of Computer Science, University of North Carolina at Chapel Hill, July 1991

20. Vezina G., Fletcher P., and Robertson P., "Volume Rendering on the MasPar MP-1", *Proceedings of the 1992 Workshop on Volume Visualization*, Boston, MA, October 1992, pages 3-8.

21. Yagel R., Kaufman A., and Cohen D., "Normal Estimation in 3D Discrete Space", *The Visual Computer*, Vol. 8, No. 5-6, June 1992, pages. 278-291.

22. Yagel R., Kaufman A.,"Template-Based Volume Viewing", *Proceedings of Eurographics'92*, Cambridge, England, September 1992, pages 157-168.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	July 1993	Final Contractor Report

4. TITLE AND SUBTITLE

Feed-Forward Volume Rendering Algorithm for Moderately Parallel MIMD Machines

5. FUNDING NUMBERS

C–NGT–26–027–011

6. AUTHOR(S)

Roni Yagel

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Ohio State University
Columbus, Ohio 43210

8. PERFORMING ORGANIZATION REPORT NUMBER

E–7907

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Lewis Research Center
Cleveland, Ohio 44135 –3191

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

NASA CR–191158

11. SUPPLEMENTARY NOTES

Project Manager, Maureen Cain, (216) 433–8928.

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified - Unlimited
Subject Category 61

12b. DISTRIBUTION CODE

13. ABSTRACT (*Maximum 200 words*)

In this report algorithms for direct volume rendering on parallel and vector processors are investigated. Volumes are transformed efficiently on parallel processors by dividing the data into slices and beams of voxels. Equal sized sets of slices along one axis are distributed to processors. Parallelism is achieved at two levels. Because each slice can be transformed independently of others, processors transform their assigned slices with no communication, thus providing maximum possible parallelism at the first level. Within each slice, consecutive beams are incrementally transformed using coherency in the transformation computation. Also, coherency across slices can be exploited to further enhance performance. This coherency yields the second level of parallelism through the use of the vector processing or pipelining. Other ongoing efforts include investigations into image reconstruction techniques, load balancing strategies and improving performance.

14. SUBJECT TERMS

Computer graphics; Volume rendering; Irregular grids

15. NUMBER OF PAGES
14

16. PRICE CODE
A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	