Proceedings of Graphics Interfaces 1996,
Toronto, Canada, May 1996, pp. 70-77..

Multi-Frame Thrashless Ray Casting with Advancing Ray-Front

Asish Law and Roni Yagel

Department of Computer and Information Science
The Ohio State University
2036 Neil Avenue
Columbus, Ohio

Phone: 614-292-0060 Fax: 614-292-2911
Email: {law, yagel}@cis.ohio-state.edu

Abstract

Coherency (data locality) is one of the most important factors that influences the performance of distributed ray tracing systems, especially when object dataflow approach is employed. The enormous cost associated with remote fetches must be reduced to improve the efficiency of the parallel renderer. Objects once fetched should be maximally utilized before replacing them with other objects. In this paper, we describe a parallel volume ray caster that eliminates thrashing by efficiently advancing a ray-front in a front-to-back manner. The method adopts an image-order approach, but capitalizes on the advantages of object-order algorithms as well to almost eliminate the communication overheads. Unlike previous algorithms, we have successfully preserved the thrashless property across a number of incrementally changing screen positions also. The use of efficient data structures and object ordering scheme has enabled complete latency hiding of non-local objects. The sum total of all these result in a scalable parallel volume renderer with the most coherent screen traversal. Comparison with other existing screen traversal schemes delineates the advantages of our approach.

Keywords: parallel rendering, volume visualization, ray casting.

1. Introduction

Volume ray casting is one of the most time consuming and memory intensive techniques used for visualizing three-dimensional data. Such volume data may be, for example, scanned by MRI (Magnetic Resonance Imaging) or CT (Computed Tomography), or simulated by CFD (Computational Fluid Dynamics) programs. Two of the most popular approaches used in volume rendering are based on *image-order* [10] and *object-order* [15] traversals. In image-order traversal, a ray is shot from the eye point and through each screen pixel. This method is also referred to as *forward-projection* or *ray-casting*. The volume is sampled at regular intervals along the ray. The values (color and opacity) at these sampled locations are composited in a front-to-back or back-to-front order to yield a final color for the pixel. Various acceleration techniques can be used to speed up the ray casting process. For example, rays can be made to terminate as soon as the accumulated opacity exceeds a pre-specified threshold value. This is known as *early ray termination* or *opacity clipping*. The sampling of the volume along the ray can also be adapted to rapidly traverse empty spaces [16], leading to significant savings in computation. The disadvantage of the image order approach, however, is that the data access is highly irregular, leading to low object-space coherency.

The object-order rendering approach is more data coherent, as voxels in the volume are traversed in a regular manner, making this approach more amenable to parallelization or vectorization [12]. Each voxel is projected onto the screen, and its color and opacity are composited to the appropriate pixels. The major disadvantage of this approach is that it cannot easily take advantage of acceleration techniques as in the case of image-order approaches. This might lead to considerable amount of unnecessary work by most of the processors. Also, it is more difficult to generate high quality images (e.g., anti-aliased images), especially when viewed in perspective. On the other hand, as each voxel in the volume has to be projected, parallel object-order techniques are inherently load-balanced in the projection stage, and in the compositing stage [11]. Moreover, object-order methods do not suffer from thrashing within a single frame generation. Voxels once brought in and processed are not needed again for the generation of the current frame. The thrashless property of these algorithms can well be preserved for a number of frames also, as a voxel can be projected to all the screen positions before discarding it from memory.

Limited memory and processing power of uniprocessor machines make volume rendering a good candidate for parallelization, the algorithms presented in [1] and [8] being considered as the most efficient parallel vol-

ume renderers. Parallel volume rendering can be classified into two categories: *object-dataflow*, and *image-dataflow*, depending on the type of data transferred between the processors. In the object-dataflow approach, the voxels (objects) are fetched only on demand, and are cached locally. With a big enough cache, this method can significantly reduce cache misses by taking advantage of ray-to-ray and frame-to-frame coherency. In image-dataflow approaches, the object is statically partitioned among processors. Each processor renders the locally available data and passes the resulting image to the appropriate processor according to the visibility order.

The ray-front scheme proposed here is an image-order traversal implemented with object-dataflow. The algorithm capitalizes on the advantages of both the image-order and the object-order traversal schemes. The method is basically image-order, and thus all the advantages of the image-order scheme are preserved. In addition, the proposed voxel fetching mechanism totally eliminates thrashing, and thus exploits object-space coherency as in the object-order methods. We have also been able to avoid thrashing across a number of images generated for different screen positions. The combination of these attributes results in a system with most coherent screen traversal so much so that voxels once fetched and used are not needed again for a number of frames, thus avoiding thrashing altogether. For colossal volume databases, this provides significant advantage over existing methods. The algorithm also predetermines the order of the non-local cells to be processed, which facilitates effective latency hiding for even minimal cache sizes. Finally, the data structures we employ circumvent the need to search for the rays that should be traversed next.

In the next section, we further illustrate the problem of coherency and emphasize the importance of our approach. Section 3 describes the method in detail, including the data structures and memory requirements. The results of our implementation on the Cray T3D, including comparisons with existing screen traversal methods, are shown in Section 4. The advantages, disadvantages, and some of our future goals are summarized in Section 5.

2. Exploiting Coherency for Efficient Rendering

In their classic paper, Sutherland et al., [13] have described coherency as the extent to which the environment, or the picture of it, is locally constant. In the present context, coherency refers to the degree to which an object required for one ray is used again for other rays, or objects used for the generation of one image (frame) are used again for another frame. In object-dataflow parallel ray-tracing systems, non-local objects are fetched from other processors on demand and are cached locally. With a coherent screen traversal, these objects are likely to be used again for subsequent rays in the current frame. If the cache is large enough, then the system can even take advantage of frame-to-frame coherency [7].

If the cache is not large enough, then it starts to *thrash*. Thrashing is manifested as the repeated transfer of the same data to the same processing node [5]. If a processor's cache cannot hold the number of blocks that it needs to render a single ray, then a cyclic refill of the cache will occur for each ray. As the size of the database increases, the effect of thrashing becomes more visible. In this respect, databases acquired from scientific sources, like sampled data from MRI, CT-scan, or CFD, are enormous, and rendering such scenes on a uniprocessor machine becomes extremely time consuming. To make the visualization of such databases more feasible, efficient parallel algorithms are being designed and implemented. The communication overheads depend on how much coherency is exploited by the algorithm, and thus how much thrashing is avoided.

Thrashing of objects in cache influences the performance in uniprocessor ray casting systems as well since the same objects are cached a number of times. For each cache miss, a penalty has to be paid for fetching the required object from main memory into the cache. The situation is even more aggravating when processing very large datasets that do not fit into the processor's main memory, and has to be fetched from disk. One would like to avoid thrashing to improve the performance of the renderer. This penalty also becomes prominent when objects are fetched from remote memories in parallel ray casting systems. Additional factors crop in when objects travel across the network. The latency is effected by network speed, which is particularly detrimental in case of distributed implementations where communication between computers are sometimes carried over slow links (e.g., Ethernet). In addition, network contention and size of the fetched data all play a combined role to increase latency and decrease performance, and therefore the scalability of the algorithm.

In view of this, it becomes particularly important to exploit object-space coherency so that objects once fetched are maximally utilized, and to ensure that objects once replaced in the cache will not be required again, thus avoiding *thrashing*. Replacement in this context, is not due to different virtual addresses mapping to the same cache location (*conflict replacements*), but due to unavailability of space in the cache (*capacity replacements*).

Visualizing colossal databases, as in the case of scientific visualization, on limited memory multiprocessor systems are prone to thrashing. Equivalently, the perfor-

mance of shared memory systems with very small caches also degrade for the same reason. These databases sometimes become so huge that they have to be stored in a compressed form on remote disks. Objects needed are decompressed and fetched on demand using the bottleneck I/O channels. Finally, there is an increasing demand for rendering multiple databases simultaneously. In all these cases, thrashing becomes unavoidable, thereby warranting a need for a thrashless visualization system.

3. Method

The ray-front visualization method is a distributed memory implementation of parallel volume rendering. The scene is initially distributed among all the participating processors. Each processor employs the image-order scheme (forward projection) for casting rays through each pixel assigned to it. Voxels that are needed in the process but are not available locally, are fetched from other processors using explicit message passing - no data is shared among the processors. We chose this distributed memory paradigm as it provides control to the programmer for mapping the fetched data in local memory, so that conflict replacements are avoided and only capacity replacements take place. By restricting the size of locally available memory, we can demonstrate the effect of capacity misses even with smaller databases. The algorithm has been implemented on the 32-processor Cray T3D available at the Ohio Supercomputer Center.

Although our algorithm is essentially an image-order method, it exploits the advantages of both the image-order algorithm (like opacity clipping and adaptive sampling) and object-order algorithm (like object-space coherency and no thrashing due to capacity replacements) to implement a caching system which totally eliminates thrashing across a number of frames by efficiently traversing the image plane while caching data in an efficient manner. In the method discussed below, we first describe how thrashing is avoided for a single frame, and then extend it to preserve its thrash-free property across multiple frames.

3.1 Screen and Scene Subdivision

The volume is subdivided into cubical cells (Figure 1a) similar to [4][5][8]. The cells are statically distributed to the processors in a pseudo-random manner to avoid hot-spotting. A cell is assigned to exactly one processor, referred to as the cell's *home node*. The screen is subdivided into stripes that are distributed cyclically to the processors (Figure 1b) to accomplish partial load-balancing among processors. We have not taken ample care to perfectly load-balance the system, as the static scheme provides ample load-balancing among processors.

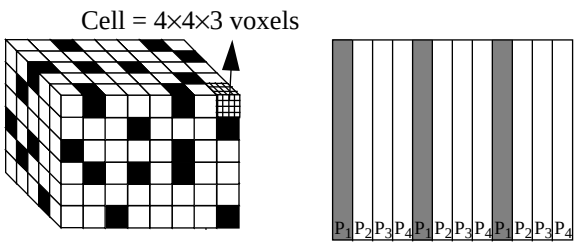

FIGURE 1. (left) A volume made up of 32×24×15 voxels is divided into 8×6×5 cells each of size 4×4×3 voxels. Each processor is home to 60 random cells in a 4-processor system. (right) A screen divided into 12 stripes of equal width and distributed cyclically to 4 processors, P1, P2, P3, and P4. For example, the black cells and the dark screen segments are assigned to P1.

3.2 Preprocessing

The first step in the preprocessing stage is to divide the volume and image among the processors as described above. The local memory is partitioned into two segments: the first segment is used to store the home cells, while the other segment is used as a cache [3]. The size of the home memory in number of cells equals the total number of cells in the volume divided by the number of processors. The home region of the memory is static as cells residing in this region (home cells) are never replaced. The rest of the memory, in number of cells is denoted by C, and is used as cache.

3.3 Ray Casting

For generating an image the following procedure is executed. Each processor computes which cells lie inside the view frustum defined by its image segment(s). The list of these cells is then ordered in a front-to-back manner depending on the position and orientation of the screen. This list is referred to as FTBL (Front-To-Back-List). Each processor also determines the first cell entered by each ray assigned to it. Next, each processor sends a request to fetch the first C non-home cells in the FTBL.

The ray casting algorithm with advancing ray-front is given in Figure 2. All the rays are initially marked as unfinished. A ray is finished if it had either accumulated enough opacity or if it exited the volume. Each ray is also marked with the cell it initially enters and is linked to a list of rays waiting for the same cell as we describe later. The algorithm traverses all the cells in FTB order, but has only one cell active at a time. All rays waiting for the active cell are advanced till they exit the cell. The active cell is then removed from the cache memory (if it is not a home cell) and a request is sent for the next non-home cell in the FTBL.

If there is space for exactly one cell in the cache, then the latency of the requested cell may not be completely

hidden. But if there is enough space for a few cells, then the latency of fetching non-home cells can be hidden, except for the first few cells. This is done by sending requests for the next few cells in the FTBL, while working on the currently active cell.

Preprocessing:

Divide the volume into cells.
Randomly distribute the cells to the processors.
Divide the screen into stripes.
Distribute the stripes to the processors in a cyclic manner.

Rendering on each processor:

```
/* Initialization..... */
Determine the FTBL of all the cells in the volume.

for each ray r do
    determine the first cell r enters -
        call this Ray[r].entering_cell
    if r does not hit the volume then Ray[r].finished = true
    else Ray[r].finished = false

/* Rendering.... */
for each cell c in FTBL order do
    for each ray r do
      if Ray[r].finished = false then
          if Ray[r].entering_cell = c then
              advance ray r through cell c
              update Ray[r].entering_cell
              if ray r exits volume or
                      Ray[r].opacity>threshold_opacity then
                  Ray[r].finished = true
```

FIGURE 2. Ray-casting algorithm with advancing ray-front.

After advancing each ray through a cell, the buffers are polled for messages with a non-blocking probe. A software handler is provided for each kind of message [6]. Depending on the type of message a corresponding action is taken. For example, if the message contains cell information, it is read from the buffers and directly put in a proper place in memory (cache). If the message is a request for a cell, it is immediately serviced by sending the requested cell to the requesting processor using a non-blocking send. The interleaving of these non-blocking sends and receives provides ample latency hiding of data in the network. At the same time, it prevents deadlock due to filling up of communication buffers.

The method described above traverses the screen in the most coherent manner, as all the rays entering a cell are advanced before any other rays are processed. This implies that cells once used will not be required again for the current frame. The raw algorithm given in Figure 2 requires one to traverse the complete list of rays and advance only those rays which are waiting for the currently active cell. This gives the search complexity of the rendering process as O(*NumCells*×*NumRays*), where *NumCells* is the total number of cells in the volume and *NumRays* is the total number of rays to be traced by a processor. The traversal of the complete set of rays can be partially avoided by limiting the search to the bounding box of the cell's projection on the screen. A simpler and more efficient scheme is used here.

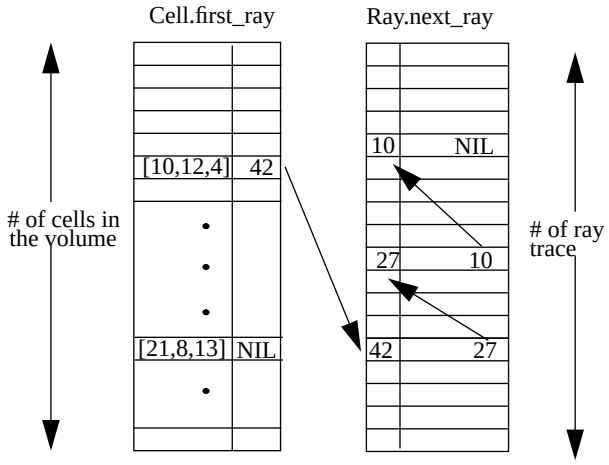

FIGURE 3. Illustration of the linked list data structure used for efficiently advancing only certain rays through a cell. The example shows that the first ray entering cell [10,12,4] is the 42nd ray, and there are only 3 rays entering this cell (42, 27, and 10). A *NIL* in the first_ray field of a cell implies that no rays enter this cell.

We use a linked list to keep track of all the rays entering a cell (Figure 3). Each cell *c*, points to the first ray entering itself (*first_ray*), and each ray *r*, points to the next ray entering the same cell as itself (*next_ray*). Whenever a particular cell becomes active (i.e., a cell brought into the cache), this linked list is traversed starting from the *first_ray* of the cell until all rays in the list are processed. This data structure precludes the necessity for traversing the complete list of rays for each cell in the volume. If no rays enter a cell, its *first_ray* itself will be marked as *NIL*, as shown for cell [21,8,13] in Figure 3. This reduces the search complexity of the algorithm to O(*NumRays*).

The complete image generation process can be viewed as being composed of several *passes*, as shown in Figure 4. In the first pass, all the rays will proceed (advance) approximately the same distance from the screen. In the next pass, the rays continue approximately the same distance again. In this manner, a ray-front moves through the volume like a wave, generating a partial image in each pass. The image converges to the final image with each pass. The first few passes of the thrash-

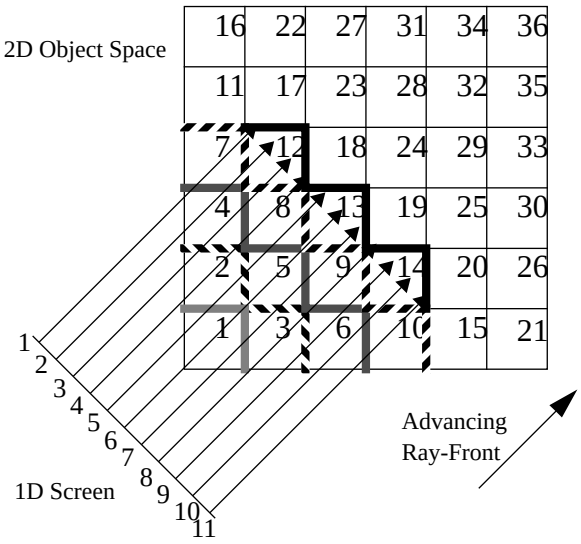

FIGURE 4. An example of advancing ray-front with 11 rays. The figure shows the advancement of the ray-front for the first 5 passes only. The 2D object space is divided into cells, and the numbers in each cell indicate its position in the FTBL.

less advancing ray-front method are exemplified in Figure 4 and in Table 1.

TABLE 1. The first 5 passes of the ray-casting algorithm with advancing ray-front for the example shown in Figure 4.

Pass #	Active cells in pass (in order)	Rays advanced in pass	Ray-Front at line
1	1	5 - 7	
2	2 - 3	3 - 9	
3	4 - 6	1 - 11	
4	7 - 10	1 - 11	
5	12 - 14	1 - 11	

The algorithm as described so far is thrashless within a single frame. Now we extend the method so that such coherency can be exploited across a number of similar frames also. By similar frames, we mean those screen positions for which the FTBL order remains the same. For example, in Figure 5, if the screen position remains in region I while viewing towards the center of the volume, then the FTBL order remains the same for all the frames, and thus the cells will be processed in the same order. The algorithm can now be followed for updating all such frames in the same pass. A frame number is attached to each ray to be processed. All the rays for all frames in a region are advanced simultaneously before the currently active cell is given up. This provides an algorithm which is thrashless across several frames. All the frames with the same FTBL is referred to as a *phase*.

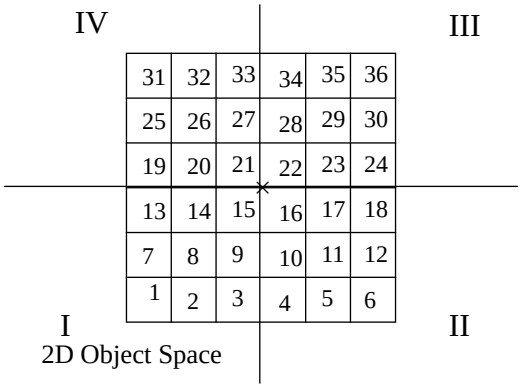

FIGURE 5. For all viewing positions in region I, and when viewed towards the center of the volume, the FTB order of the cells are as shown. × denotes the center of the volume.

4. Results

4.1 Number of Frames per Phase

Figure 6 shows the times and number of cells received as the number of frames in a phase increases. Timings are taken for generating 30 incrementally changing frames for a 256^2 screen rotating around a 128^3 volume. *Frames/phase* indicates the number of frames across which thrash-free operation is preserved. Figure 6(a) shows a consistent decrease in total animation time with increase in frames/phase. This is mainly due to the savings in the communication required to fetch the drastically reduced number of cells, along with other associated overheads, like less frequent updating of local memory with the fetched cells, and reduced network contention. The bump in the curve at 5 and 6 frames/phase is not clear at this time.

The improvement in timing performance is not significant as the Cray T3D uses extremely fast and efficient communication channels for transferring data. The communication of extra cells has minimal effect of the performance of the system. We expect the savings in time to be much more significant when the communication is not as efficient, especially on a network of workstations with slower Ethernet links.

Although the improvement in time is not considerable, the total number of cells fetched decreases drastically. When all 30 frames in the animation process are processed all in the same phase, then a cell is fetched

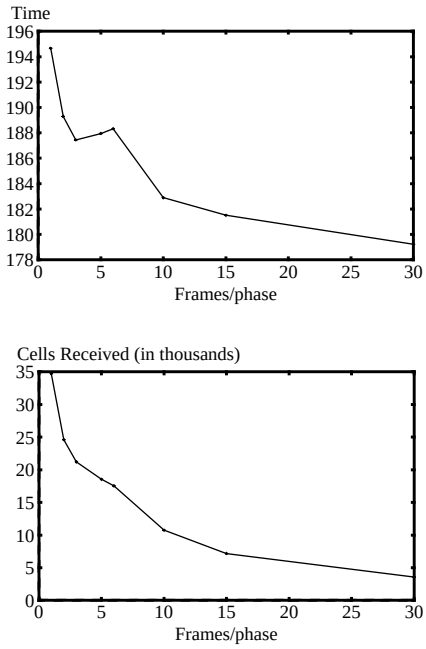

FIGURE 6. Times (in secs.) and Number of cells received (in thousands) with number of frames generated in each phase of the algorithm for generating 30 frames.

only once during the whole process. From Figure 6(b), we can see that when only a single frame is processed in a pass (phase), then about 35000 cells are fetched from distant memory. In contrast, if all 30 frames can be processed in the same pass, then this figure drops down to the minimum required (about 4000). The algorithm is much more effective when it is used with compression caches or when data is being fetched from disk on demand. In such cases, thrash-free operation across a number of frames will have an enormous impact on the

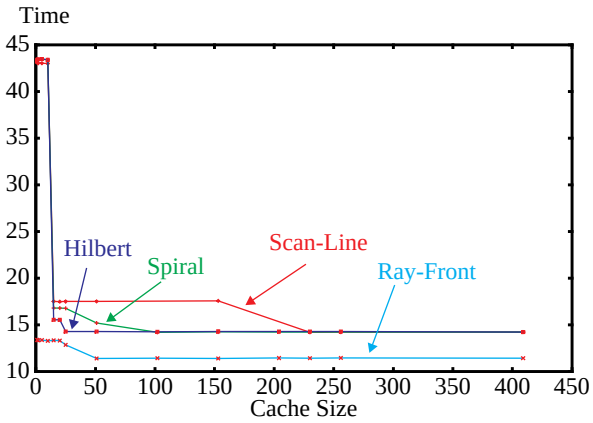

FIGURE 7. Comparison of four different screen traversal schemes - scan-line, spiral, hilbert, and rayfront. The graph shows the times taken for generating 30 frames in the animation.

performance of the system. For example, for rendering using compression caches, 35000 cells will undergo decompression in the case of 1 frame/phase as opposed to 4000 cells in the case of 30 frames/phase.

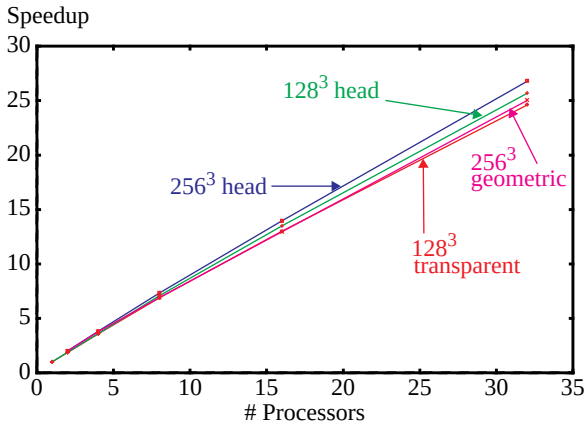

FIGURE 8. Scalability of the ray-front algorithm for different sized volumes

4.2 Comparison

Figure 7 compares the performance of the RayFront algorithm with three of the most common screen-traversal algorithms: scan-line, spiral, and Hilbert [2]. In each of these, the screen regions were distributed to the processors in exactly the same manner as in our algorithm. The only difference was the way in which the respective regions were traversed by each algorithm. Out of these, the Hilbert is believed to be the most coherent screen-traversal scheme [17]. It is evident from is graph that the screen traversal used for the ray-front algorithm outclasses the others at all cache sizes for parallel projection ray casting. The performance gain at lower cache sizes is particularly noteworthy. The primary advantage

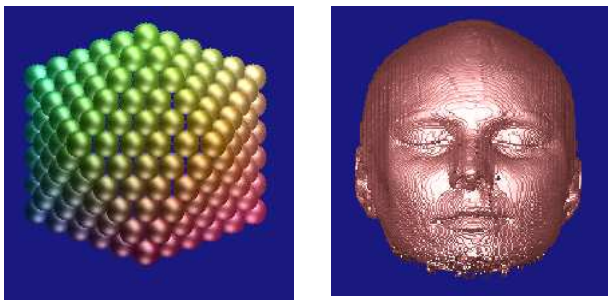

FIGURE 9. Volume rendered images of geometric object (left) and MRI dataset (right). Both are 256^3 resolution and are rendered to 360^2 image.

is the algorithm's thrash-free property even for minimal cache sizes.

The consistent improvement in the timings at all cache sizes can be attributed to two main reasons. First, for parallel projection ray casting, the ray-front algorithm predetermines the order in which the cells should be processed. It further culls all the cells which do not fall within its view frustum. A processor thus fetches exactly those cells that are needed, and in the correct order. Second, the predetermination of the cells facilitates latency hiding - an attribute which cannot be exploited advantageously by the other algorithms.

That the latency is significantly hidden is manifested by number of cells fetched. The pattern of the number of transferred cells as a function of cache size is similar to what is shown in Figure 7. Previous algorithms, like scan-line, spiral, and Hilbert, fetch cells only if and when needed. The RayFront algorithm, on the other hand, makes a conservative estimate of the cells which may be required in the future. For higher cache sizes, the number of cells fetched becomes more than that of the other algorithms. This is because it is difficult to predetermine the nature of the object's transparency properties, making it impossible to predict if a back-cell will be traversed by a ray or not. As a result, some unneeded back-cells are also fetched. In spite of fetching these extra cells, the total animation time remains constant, implying total latency hiding of these cells.

4.3 Scalability

The efficiency and speedup graphs of the ray-front algorithm for different volumes are shown in Figure 8. Four different datasets were used to test the scalability of the algorithm: a 128^3 totally transparent volume, a 128^3 illuminated head, a 256^3 illuminated head, and a 256^3 geometric object. The completely transparent object was chosen to ensure that each ray will traverse its entire length. As such, all the cells along the ray will be fetched. This maximizes the number of cells communicated between processors. The geometric object was made up of 252 spheres, organized in the form of a dodecahedron with triangular faces. The images generated by the algorithm for the 256^3 head and the geometric object are shown in Figure 9. The algorithm demonstrates about 80% efficiency for 32 processors for all these test volumes. The good speedup also suggests that considerable load-balancing has been achieved using the static block-cyclic scheme as described in Section 3.1.

5. Discussion

The RayFront algorithm provides the most coherent screen traversal scheme to avoid thrashing in object dataflow parallel ray casting systems. Its main advantage lies in the arena of rendering colossal databases, where thrashing is bound to occur due to shortage of memory. Thrashing manifests itself in the degraded turn-around time of the rendering system. Our method is particularly applicable in such cases, and will show significant advantage over existing screen traversal schemes. It is our intent to show that the ray-front method is advantageous even on uniprocessor systems. This is because with the proposed advancing ray-front scheme, the cache efficiency improves, and thrashing is avoided even in uniprocessor machines. This can be asserted by verifying that the improvement gained by efficient caching of cells is not offset by the traversal of the data structures employed by our algorithm.

Our method is advantageous over other similar implementations [14], as we have achieved thrash-free property across a number of frames also. Our efficient data structures optimizes the complexity of the ray search, and the cell ordering scheme we employ facilitates effective latency hiding making the algorithm scalable. Finally, we have brought the two classes of volume rendering algorithm, image-order and object-order, together, and successfully exploited the advantages of both these approaches.

The main disadvantage of this method is the extra memory used for maintaining the data structures. Also, as the rays are not traversed to completion, the attributes of all the rays have to be stored. This is not required in traditional approaches as a ray starts and traverses to completion before starting the next ray. The parallel-projection system developed here should also be extended to include perspective projection. It will not be trivial to determine the FTBL of cells when viewed in perspective, making it more difficult to hide the latency for non-local fetches.

With this space-time trade-off, we want to extend the proposed parallel projection ray casting method to ray tracing also. In this sense, we suggest a breadth-first processing of rays instead of the commonly used depth-first approach. In existing parallel ray-tracing systems, the primary ray and all its secondary rays are processed before proceeding to the next pixel. For huge databases, or with sufficient depth of the ray-tree, this method is prone to thrashing. If a breadth-first approach is adopted instead, all the primary rays entering a cell can be processed before moving on to the next level of secondary rays. Data structures similar to the one used here can be employed to efficiently keep track of all the primary and secondary rays entering a cell. All the rays waiting for a particular cell should be advanced once this cell has been fetched. Of course, this method is not free from thrashing, but the chances of thrashing will reduce. The determination of which cell to bring next is an open issue, as for ray-tracing systems, a front-to-back order cannot be assigned to the cells.

6. Conclusion

We have developed a distributed memory ray-casting scheme which incorporates the advantages of both object-order (no thrashing, regularity of access, object-space coherency) and image-order (opacity clipping - avoiding extraneous calculations, higher image quality, simplicity, and usage of other acceleration techniques) algorithms. We have shown that our most coherent screen traversal method exploits coherency far more efficiently than the traditional counterparts. For rendering colossal databases, this provides significant improvement by making the system thrashless. The algorithm has been extended to avoid thrashing for a number of frames also. Efficient data structures have been suggested to improve the time complexity, and to facilitate effective latency hiding. This makes the method scalable to a number of processors. In the future, we would like to extend the algorithm to view in perspective and to use a similar scheme for ray tracing also.

Acknowledgments

This work has been partially supported by the National Science Foundation under grant CCR-9211288, and DARPA under BAA 92-36. We thank The Ohio Supercomputer Center for allowing us to use of the Cray T3D.

7. References

1. M.B. Amin, A. Grama, V. Singh. “Fast Volume Rendering Using an Efficient, Scalable Parallel Formulation of the Shear-Warp Algorithm”, *Proceedings of Parallel Rendering Symposium*, Atlanta, October 1995, pp. 7-14.
2. J. Arvo. “Space-Filling Curves and a Measure of Coherence”. *Graphics Gems II*, Chapter 1.8, pp. 26-30.
3. D. Badouel, K. Bouatouch, T. Priol. “Ray Tracing on Distributed Memory Parallel Computers: Strategies for Distributing Computations and Data”, SIGGRAPH ‘90, Parallel Algorithms and Architecture for 3D Image Generation, Course Notes. pp. 185-198.
4. J. Challinger. “Parallel Volume Rendering on a Shared-Memory Multiprocessor”, Department of Computer and Information Sciences, UC Santa Cruz, Technical Report UCSC-CRL-91-23, revised March 1992.
5. B. Corrie, P. Mackerras. “Parallel Volume Rendering and Data Coherence”, *Proceedings of Parallel Rendering Symposium*, San Jose, California, October 1993, pp. 23-26.
6. T. von Eicken, D.E. Culler, S.C. Goldstein, K.E. Schauser. “Active Messages: a Mechanism for Integrated Communication and Computation”, *ACM Transactions* 1992, pp. 256-266.
7. S. Green, D. Paddon. “Exploiting Coherence for Multiprocessor Ray Tracing”, *IEEE Computer Graphics and Applications* 9, (6), pp. 12-26, November 1989.
8. P. Lacroute. “Real Time Volume Rendering on Shared Memory Multiprocessors Using the Shear-Warp Factorization”. *Proceedings of Parallel Rendering Symposium*, Atlanta, October 1995, pp. 15-22.
9. A. Law, R. Yagel. “CellFlow: A Parallel Rendering Scheme for Distributed Memory Architectures”, *Proceedings of the International Conference on Parallel and Distributed Techniques and Applications*, Atlanta, November, 1995, pp. 1-10.
10. M. Levoy. “Display of Surfaces from Volume Data”, *IEEE Computer Graphics and Applications*, Vol. 8, No. 5, May 1988, pp. 29-37.
11. K.L. Ma, J.S. Painter, C.D. Hansen, M.F. Krough. “A Data Distributed, Parallel Algorithm for Ray-Traced Volume Rendering”, *Proceedings of Parallel Rendering Symposium*, San Jose, California, October 1993, pp. 15-22.
12. R. Machiraju, R.Yagel. “Efficient Feed-Forward Volume Rendering Techniques for Vector and Parallel Processors”, *Proceedings of Supercomputing ‘93*, Portland, OR, pp. 699-708.
13. I.E. Sutherland, R.F. Sproull, R.A. Schumacker. “A Characterization of Ten Hidden-Surface Algorithms”, Computing Surveys, Vol. 6, No. 1, March 1974, pp. 1-55.
14. R. Westermann, S. Augustin. “Parallel Volume Rendering”, *Proceedings of International Parallel Processing Symposium*, 1995, pp. 693-699.
15. L. Westover. “Footprint Evaluation for Volume Rendering”., Computer Graphics (SIGGRAPH ‘90 Proceedings), Vol. 24, 1990, pp. 367-376.
16. R. Yagel, Z. Shi. “Accelerating Volume Animation by Space-Leaping”, *Proceedings of Visualization’93*, San Jose, California, October 1993, pp. 62-69.
17. H. Zhang, S. Liu. “Order of Pixel Traversal and Parallel Volume Ray Tracing on the Distributed Volume Buffer”. Presented at the Eurographics Workshop on Volume Visualization, 1995.