Rendering Computer Animations on a Network of Workstations

Timothy A. Davis
Edward W. Davis
Department of Computer Science
North Carolina State University

Abstract

Rendering high-quality computer animations requires intensive computation, and therefore a large amount of time. One way to speed up this process is to devise rendering algorithms which reduce computation by taking advantage of image characteristics, such as frame or temporal coherence. Another way is to run the rendering process on specialized or enhanced hardware, such as a multiprocessor machine. This paper details our experiences in combining these techniques through a parallel rendering algorithm exploiting frame coherence run on a network of workstations acting as a single processing system. This system provides a powerful and general method for obtaining high-quality animations with a significant reduction in computation and overall processing time. We present several techniques for partitioning the data across the workstation processors and report our results for each. While all of the partitioning schemes are scalable, some handle load balancing more effectively than others.

1. Introduction

Ray tracing is a popular and powerful method for generating realistic graphics images and animations depicting a broad range of objects from a variety of applications. Some examples include molecular chemistry models, topographical terrain fly-overs, astronomical phenomena, and various images used in motion pictures and television. The proliferation of ray tracers freely available on the internet is a testament to the continued use and popularity of the technique.

The generation of such high-quality images, however, has a price: large amounts of computation and long rendering times. Some short animation sequences, lasting only seconds, can take hours or days to render. For long, complex animations, the rendering time can be intractable. As an example, consider the Pixar/Disney venture *Toy Story* [9]. (Although this motion picture was not rendered using ray tracing, the point is still well illustrated.) To generate its 144,000 individual frames, *Toy Story* required about 43 years of CPU time. If not for the 117 machines participating in the computation, the movie's production could not be realized.

Since rendering requires such large amounts of time, a great number of techniques for reducing computation and thus rendering time have been suggested [6] [7]. In many animation sequences, an individual frame will not differ markedly from the previous frame. This continuity of image data is termed *frame coherence*. Surprisingly few ray tracing algorithms actually take advantage of this type of coherence; instead, they produce successive frames individually from the scene description. Even fewer algorithms (perhaps none) exploit frame coherence in conjunction with the data parallelism inherent in rendering ray-traced animations.

Schaufler [11] uses frame coherence in a virtual reality system for maintaining an acceptable interactive frame rate while presenting high-quality images to the user. This scheme exhibits good performance within the scope of the virtual world for which it was intended; however, the representation is an approximation; that is, if we actually rendered the full scene, it would differ from the scene presented. Our goal is to render scenes more quickly without compromising on image content.

Adelson and Hodges [1] use coherence to compute stereoscopic pairs in a ray-tracing environment. Their results show that up to 95% of the pixels computed for the left-eye view can be copied to the right-eye view to attain an approximation of the fully ray-traced image. While these results reflect a substantial time savings, the algorithm is designed to take advantage of coherence only within a single frame of the animation, not within the time dimension of successive frames.

Although specialized graphics multiprocessors have been designed [4], for our experiments, we employ a network of workstations, also termed a *clustered computing*

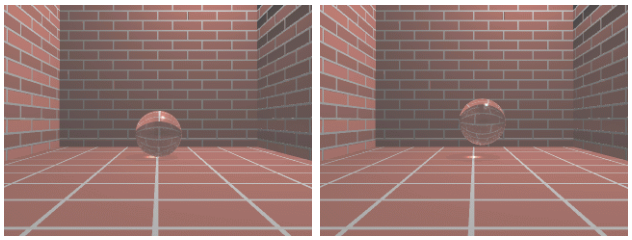
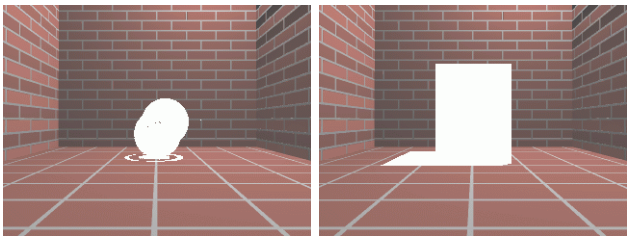


Figure 1: first two frames of a sample animation



(a) (b)

Figure 2: (a) actual pixel differences between frames (b) pixel differences as computed by the frame coherence algorithm

environment, as our parallel platform. Clustered computing environments are effective for graphics processing [2] and in general, continue to gain popularity for several reasons.

First, clustered computing environments are cost-effective. Traditionally, multiprocessor machines are expensive; however, many sites already have a network of workstations in place. Additionally, clustered environments can be enhanced incrementally. That is, additional processors (workstations) can be purchased and linked with the existing cluster over a long period of time. Parallel algorithms can often take advantage of the heterogeneity of networked machines by matching the computation of a subproblem to the most appropriate processor. Finally, message-passing systems, such as PVM (Parallel Virtual Machine) [5] and MPI (Message Passing Interface) [8], are robust, easy to use, and available without cost.

To parallelize the frame coherence algorithm, an efficient method must be determined for decomposing the problem and distributing the computation to the workstations. Load balancing should be handled automatically, while keeping communication costs as low as possible on the ethernet network, which is relatively slow compared to interconnection networks found on multiprocessor machines.

Section 2 of this paper presents the frame coherence algorithm, while Section 3 discusses data partitioning schemes for animated sequences. In Section 4, performance results are provided for a sample animation. Finally, Section 5 gives conclusions and some future directions of this work.

2. Frame coherence algorithm

Frame coherence was first described in [12], along with a long list of other types of coherence that can be exploited in rendering as well. Frame coherence, as described in the preceding section, denotes the continuity, or lack of change, from one frame to the next. Figure 1 shows the first two scenes of a ray-traced animation in which a glass ball bounces around a brick room. By performing a pixel-by-pixel comparison between the two frames, we get the image in Figure 2(a), where the white areas denote those pixels that changed from the previous frame. One striking feature is the large number of pixels that do not change -- these do not need to be re-computed for the next frame.

The frame coherence algorithm is therefore predictive in nature. Once a frame has been rendered, we must determine which pixels, if any, in the next frame actually need to be re-computed. To accomplish this task, the object space is divided into voxels (or cubes) through uniform spatial subdivision. As rays are fired during the rendering process, the frame coherence algorithm tracks their paths and marks all of the voxels that they pass through. For a given pixel, numerous rays may be generated, including the initial camera ray, reflected rays, refracted rays, and shadow rays. Each of these rays passes through a modified 3D-DDA algorithm to determine which voxels they traverse. If a particular voxel experiences some sort of change (e.g., an object moving into it) in the next frame, all of the pixels whose rays pass through that voxel must be updated.

A general outline of the algorithm appears in Figure 3. The algorithm was implemented in C as an addition to the

```
parse the user input parameters
initialize frame coherence data structures

for each frame of the animation
    for each pixel that needs to be computed
        for each voxel that a ray associated with this
          pixel intersects
            add the pixel to the voxel's pixel list

    find the voxels in which change occurs in the next
      frame as compared with this one

    mark those pixels on the pixel list of the changed
      voxels for recomputation in the next frame
```

Figure 3: frame coherence algorithm

popular Persistence of View ray tracer (POV-Ray), version 3.0, which contains numerous advanced ray tracing features and is freely available on the internet. This approach resembles the scheme proposed by Jevans [10], but differs in essentially three ways. First, Jevans' algorithm computes coherence for blocks of pixels (that is, if one pixel in the block needs to be updated, all pixels in the block are re-computed). Our algorithm, in contrast, computes coherence on a much finer level of granularity of individual pixels. Second, we are also exploring the use of frame coherence in the generation of shadows. Finally, Jevan's algorithm is serial in nature, while ours is targeted for parallel platforms. A more detailed description of the algorithm can be found in [3].

3. Data partitioning

To parallelize the algorithm, an appropriate method for decomposing the problem must be determined. The intensity for each pixel is determined by

$$I = I_{local} + k_{rg}I_{reflected} + k_{tg}I_{transmitted}$$

where

I_{local} is the local term due to direct illumination

k_{rg} is a wavelength-independent constant indicating the amount of reflectivity

$I_{reflected}$ is the amount of reflected light

k_{tg} is a wavelength-independent constant indicating the amount of refraction

$I_{transmitted}$ is the amount of transmitted light

Fortunately, the calculation for each pixel in a ray-traced image can be computed independently. We can therefore divide each frame into regions of any size, all of which can be computed in parallel. The master process handles this task in addition to collecting rendered image information and writing this information out to files. The only interprocessor communication occurs between the master and each of the slaves; the slaves themselves do not need to communicate with each other.

The interesting part of the data partitioning surfaces in the time domain: determining the best scheme for keeping frames together to exploit coherence while breaking the problem apart into many pieces to exploit parallelism. Since the frame coherence algorithm proposed here works only for sequences in which the camera is stationary, any camera movement logically separates one sequence from another. These shorter sequences represent the computational tasks for which parallelization and frame coherence will be exploited.

One method of parallelizing such a sequence, which we will term as *sequence division*, involves dividing up whole frames among the available processors so that each receives a subsequence of the full animation (see Figure 4(a)). For example, if four processors participate in the computation of 120 240x320 frames of animation, each processor would be assigned 30 240x320 frames to generate. Note that the frames must be consecutive to take advantage of any frame coherence between them. A potential drawback to this method occurs if the number of frames assigned to each processor is static. The situation may lead to load imbalance due to differing processor speeds and the complexity of the subsequences. Each sequence, however, can be adaptively subdivided such that a faster processor can receive more work once it completes its sequence.

Another possible method for decomposition is *frame division*. Under this scheme, each frame is divided into subareas, each of which is computed by a separate processor for the entire animation sequence (see Figure 4(b)). As an example, consider the four-processor system again. For 120 240x320 frames of animation, each processor would render 120x160 pixels of each of the 120 frames. Load balancing remains an issue here also, especially since different subareas are likely to experience radically different amounts of change, but we can once again recursively subdivide remaining subsequences for redistribution. This method, however, has the advantage of requiring less memory of each of the processors to execute the frame coherence program since memory requirements are directly proportional to the size of the image area.

Reducing the size of the subarea in frame subdivision can result in better load balancing. That is, we may divide the 240x320 frames into areas of 80x80 pixels. Now we have more subareas than processors, so whenever a processor finishes its sequence, it can request another one.

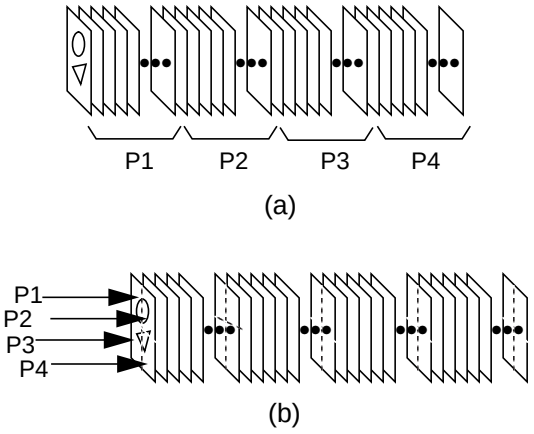


Figure 4: (a) sequence division
(b) frame division

This scheme becomes most effective when each frame has large dimensions or contains objects with complex characteristics since these cases have high memory requirements. At the extreme, we could assign each processor a single pixel to compute for the entire sequence; however, the overhead of message passing, as well as other bookkeeping tasks, would result in inefficiency and longer execution time.

Of course, many other decomposition schemes exist, such as a hybrid of the two methods proposed above (i.e., each processor computes pixels in a subarea of a frame for a subsequence of the entire animation), primarily due to the high parallelism and lack of interprocessor communication. Even for a medium-sized frame of 240x320, there are 76,800 independent calculations to perform (one for each pixel) per frame. Using the frame coherence algorithm, each of the 4 processors would receive 19,200 pixels to process, if each frame were subdivided evenly. Computation for this number is easily tractable, but certainly not trivial, and outweighs any time spent communicating with the master process.

4. Performance results

In this section, we include some execution results for a sample POV-Ray animation. All experiments were performed on equipment at North Carolina State University's Multimedia Lab. The test machines consisted of one SGI Indigo 2 running at 200 MHz with 64 MB of memory, one SGI Indigo 2 running at 100 MHz with 32 MB of memory and one SGI Indigo also running at 100 MHz with 32 MB of memory. For the single processor case, the fastest machine was used; for the distributed computing tests, PVM 3.1 was used to coordinate the processing. The POV-Ray renderer generated animation frames with 240x320 resolution in *targa* format with 24-bit color. Image quality was set to high with a maximum ray depth of 5. All times are reported in hours:minutes:seconds.

The *Newton* animation, designed by Chris Gulka, consists of a set of suspended chrome marbles, which when set into motion by raising the marble on either end, illustrates the law of the conservation of energy (a single frame is shown in Figure 5). This animation, consisting of one plane, five spheres, and sixteen cylinders, is broken into two separate rendering runs; we will focus on the first.

Our results show the benefits of using frame coherence by itself as a means of decreasing rendering time on a single processor. Next, we include results which show the benefits of using distributed computing by itself as a means for improving performance. We then combine the use of frame coherence and distributed computing to show the multiplicative advantages of applying both. For rendering involving distributed computing, two decomposition methods were used: sequence division (in which each subsequence was initially assigned to a processor, but later adaptively subdivided to keep all of the processors busy) and frame division (in which 80x80 pixel subareas were assigned to processors to compute for the entire 45 frames, or until the sequence was adaptively subdivided).

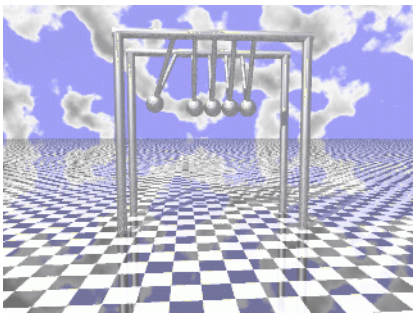


Figure 5: frame 22 of the Newton animation

Table 1 shows the performance results for rendering both with and without the frame coherence algorithm on a single processor (columns (1) - (3)). The measurements for the first frame rendering are provided to show the overhead associated with the algorithm. Here, overhead constitutes a reasonable 12% of the total generation time. (First frame rendering times are not provided for the other methods since they do not necessarily complete frames in order.) Overall, the results reported here are encouraging: the total number of rays produced decreased by a factor of 5, while total animation generation speed increased nearly by a factor of 3.

The frame coherence algorithm performs well on this particular animation because performance depends on the amount of frame coherence we can actually extract from the scene. Only a small area of the scene changes per frame, allowing us to avoid computing the majority of the pixels. Additionally, those pixels that did not change were not easily calculated to begin with -- many involved numerous reflections and shadows, thus contributing to the savings.

Columns (4) and (5) show the results of applying distributed computing to the rendering process, but without frame coherence. Each frame is subdivided into 80x80 blocks which are distributed to the machines as they request them. Rendering is about twice as fast here, as expected: the single processor tests were performed on a machine which runs twice as fast as each of the other two.

Columns (6) and (7) show the frame coherence results for distributed computing using a sequence division partitioning scheme; columns (8) and (9) for frame division. Note here that we achieve a multiplicative effect in performance, resulting in significant speedups of 5 and 7 for

	(1) single processor	(2) single processor with fc	(3) ratio of (2) to (1)	(4) 3 processors	(5) ratio of (4) to (1)	(6) 3 processors with fc (seq div)	(7) ratio of (6) to (1)	(8) 3 processors with fc (frame div)	(9) ratio of (8) to (1)
# rays	21,970,900	4,342,799	0.198	22,239,046	1.012	5,943,877	0.271	4,601,380	0.209
first frame rendering time	2:30	2:48	1.120	--	--	--	--	--	--
average frame rendering time	2:34	0:57	0.368	1:18	0.505	0:26	0.172	0:22	0.144
total frame rendering time	1:55:51	42:28	0.367	58:17	0.503	19:52	0.171	16:40	0.144

Table 1: Performance results for *Newton*

sequence and frame division, respectively. We actually do a little better than the multiplicative expectation (18.5%) due to the increased aggregate memory of multiple machines and the overlapping of computation and file writing.

5. Conclusion and future directions

We have shown that combining frame coherence with distributed computing can substantially reduce the amount of time required to render a ray-traced animation. Some care must be taken, however, when deciding how to partition the data for distribution to the processors (workstations) to achieve the best load balancing. Our results give some insight into these decisions, although further tests with heterogeneous environments, as well as more homogeneous ones, will prove beneficial in producing general recommendations and heuristics. Even with our current environment and test animations, we obtain good performance results with a reasonable amount of overhead.

Depending on the number of workstations participating in the computation and the performance power of each of the machines, one can build an extremely powerful rendering environment. This type of system may provide the capability to produce incredible computer animations that are otherwise impossible to render. Our future research goals include the refinement of adaptive partitioning schemes, development of frame coherence algorithms with shadow generation, and experimentation with large, complex animations that can more fully benefit from the frame coherence techniques.

References

[1] Adelson, S. J. and L. F. Hodges, "Stereoscopic Ray-Tracing," *The Visual Computer*, Vol. 10, pp. 127-144, 1993.

[2] Davis, T. A.,"Parallelizing Graphics Applications with PVM and MPI," *Proceedings of the 1997 Cluster Computing Conference*, Atlanta, GA, March, 1997.

[3] Davis, T. A., "Generating Computer Animations with Frame Coherence in a Distributed Computing Environment," *Proceedings of the 1998 ACM Southeast Conference*, Atlanta, GA, March, 1998.

[4] Fuchs, H., J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, and L. Israel, "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories," *Computer Graphics*, Volume 23, No. 3, July, 1989, pp. 79-87.

[5] Geist, A., A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V Sunderam, *PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing*, The MIT Press, Cambridge, MA, 1994.

[6] Glassner, A. S., "Space Subdivision for Fast Ray Tracing," *IEEE Computer Graphics and Applications*, Vol. 4, No. 10, pp. 15-20, Oct., 1984.

[7] Glassner, A. S. (ed.), *An Introduction to Ray Tracing*, Academic Press, 1989.

[8] Gropp, W., E. Lusk, and A. Skjellum, *Using MPI - Portable Parallel Programming with the Message-Passing Interface*, The MIT Press, Cambridge, MA, 1994.

[9] Henne, M., H. Hickel, E. Johnson, and S. Konishi, "The Making of *Toy Story*," *IEEE COMPCON '96 Digest of Papers*, 1996.

[10] Jevans, D., "Object-Based Temporal Coherence," *Proceedings of the 1992 Computer Graphics Interface Conference*, 1992.

[11] Schaufler, G., "Exploiting Frame-to-Frame Coherence in a Virtual Reality System," *Proceedings of the 1996 Annual Virtual Reality International Symposium*, IEEE Press, pp. 95-102.

[12] Sutherland, I. E., R. F. Sproull, and R. A. Schumacker, "A Characterization of Ten Hidden-Surface Algorithms," *ACM Computing Surveys*, Vol. 5, No. 1, pp. 1-55, Mar., 1974.