Parallel Incremental Raytracing of Animations on a Network of Workstations

Bernd Freisleben, Dieter Hartmann
University of Siegen, Germany
{freisleb,dieter}@informatik.uni-siegen.de

Thilo Kielmann
Vrije Universiteit, Amsterdam, The Netherlands
kielmann@cs.vu.nl

Abstract *To reduce the computation times required for rendering animations, a new incremental raytracing method that computes only the changed parts of images in an animation sequence is proposed. This method is integrated into a parallel version of the* POV–Ray *raytracing package implemented on a network of workstations using the MPI message passing interface. The parallelization relies on a manager/worker scheme which incorporates dynamic task assignment to achieve load balancing. The results of several experiments indicate that the incremental raytracing method yields a reduction of computation time roughly proportional to the number of changed pixels in a particular animated scene. Almost linear speedups are obtained for the parallel versions running on up to 18 workstations.*

Keywords: Incremental raytracing, parallelism, network of workstations, MPI

1 Introduction

Raytracing [1] is a technique used for producing complex graphics images involving reflections, shadows, transparent objects and other features of the real world. It relies on the idea of reversing the direction of natural light rays traveling from a light source to objects which absorb some or all of the light and reflect or refract the rest until the rays eventually enter our eyes and determine what we see.

The input to a raytracing algorithm is the *scene* – a description of the geometry of 3D objects and the definition of the objects' materials, the lights, and the imaginary eye or camera. The output is the *image* of the scene, i.e. the intensity value of each pixel of the image (also called a *frame*), as seen by the defined camera. To determine the value of a pixel in an image, a ray starting at the eyepoint moves through this pixel in the viewing plane (i.e. the frame) and tests the objects in the scene for intersection. Since reflections or refractions at each intersection point may create new rays, the amount of computation increases rapidly for complex scenes. However, due to the independence between the calculations of pixels in a frame, the parallelization of raytracing has become practical. In this paper, we present a parallel raytracing approach that not only exploits this independence, but that also benefits from the time coherence between several frames of an animation to further reduce rendering time.

There are two basic approaches to parallelizing the raytracing process. The first is a domain decomposition of the input data, i.e. the scene description is partitioned, and the parts are processed by different processors. This approach is commonly in use because it distributes the possibly large memory requirements among the available processors [2]. Unfortunately, information about other subsets of objects may be needed to perform a processor's computations, leading to a possibly high amount of communication.

The second approach is a domain decomposition of the output to be produced, i.e. the pixels of an image and/or the images of an animation are distributed to different processors and colored independently. In this (task parallel) approach, each processor must have the complete scene description for accessing all the information about the rays involved. Raytracing parallelized in this way constitutes an *embarrassingly parallel* problem (i.e. the amount of inter-process communication is reduced to a minimum) yielding a high speedup potential, whereas the data replication problem may be somewhat alleviated by using a distributed shared memory architecture [2].

In this paper, we extend our previous approach to task-parallel raytracing of single images on workstation networks [3] to the rendering of animated scenes. Animations allow to further reduce the computation times by taking advantage of the temporal dimension of a sequence of frames. Many animations, particularly for scientific or educational purposes, are based on fixed eye- or camera positions and operate under only slightly changing lighting conditions. In this case, once an initial reference frame has been rendered, many pixels of static objects or the background do not change in successive frames; only the pixels belonging to objects moving or changing during the animation have to be re-computed. This leads to the idea of performing raytracing incrementally for (a subset of) frames of an animation [4]. We present a new incremental raytracing approach which is based on efficiently localizing the influences of moving or changing objects by using a suitable (reasonably small) bookkeeping data structure for recording incremental pixel changes. A manager/worker scheme with dynamic task assignment to achieve load balancing is used to perform this incremental raytracing method, integrated into the POV–Ray raytracing package [5], in parallel on a network of DEC Alpha workstations. Examples will be presented to demonstrate the significant reductions of computation time achievable using the proposed approach.

2 Incremental Raytracing

The foundation of incremental raytracing is the temporal coherence of (subsets of) frames of an animation. Several approaches have been proposed to take advantage of this frame coherence. For example, Badt [6] has presented algorithms for determining the regions in the current frame which have changed compared to an initial frame. Glassner [7] exploited the temporal coherence of frames to reduce the number of ray/object intersection tests; he performed 4D raytracing based on a particular 3D space subdivision technique for creating non-overlapping hierarchies of bounding volumes. Müller [8] proposed a so-called look-ahead algorithm that builds up a binary tree of scenes of an animation and traverses it to determine whether a ray has already been traced in a previous frame. Jevans [9] introduced voxel subspaces to be able to re-compute only rays that intersect with a voxel subspace containing a moving object. Davis and Davis [10] extended Jevans' approach to exploit frame coherence at the pixel level and parallelized it. The term 'incremental raytracing' was presumably first used by Hirota and Murakami [4] who subdivided the object space and maintained binary shade trees of all rays to estimate the object subspace that will influence the rays in the next frame. Since the memory needed to preserve the binary shade trees becomes huge when the number of pixels gets large or the scene becomes complex, Horiguchi et al. [11] improved this algorithm by introducing the concept of a *locus cell*. It represents the subspace through which the objects of a frame are moving and which includes all the rays that need to be recalculated by moving objects in the next frame.

The aim of our incremental raytracing method is to re-compute only the pixels in a frame that are influenced by moving or changing ('dynamic') objects (including their reflections and refractions). In contrast to previous incremental approaches, we do not consider the changed pixels between pairs of successive frames in the sequence. Instead, the first step of our method is to compute an initial frame which is used as the common reference point for the changes occuring in each frame of an animation. This reference frame is not part of the animation itself, but it is additionally generated to contain the background and all static objects; the idea is to represent each frame of the animation only by its difference to this reference frame. The use of a reference frame was motivated by the results of several experiments, in which the computation times of the reference frame method were found to be less than those of an incremental method based on pairs of successive frames. The reference frame is computed as part of the following procedure:

- First, the scene description for the animation is parsed, and for each dynamic object a so-called *dynamic-object-box* (DOB) is created and inserted into the scene description data. DOBs are simple 'invisible' bounding volumes covering the extent of a dynamic object after its first appearance in all successive frames. Since they have no textures and rays simply pass through them without being reflected or refracted, they can be computed very fast.

- Then, the raytracing algorithm is started (a) to render the reference frame, and (b) to determine the pixels influenced by dynamic objects in each frame. Three cases must be distinguished:

  1. If a ray does not intersect with any object in the frame sequence, the corresponding pixel in the reference frame gets the background color.

  2. If a ray intersects with a static object in the frame sequence, the corresponding pixel in the reference frame gets the color of the static object.

  3. If a ray intersects with a dynamic object or a DOB, the corresponding pixel gets marked, and the ray is assumed to pass through them without changing its direction. This is continued for each dynamic object or DOB on the ray's way, until an intersection with a static object or the background is encountered. This color will be entered into the corresponding pixel position in the reference frame.

  This procedure is repeated for all rays reflected and refracted from static objects.

  The 'invisible' DOBs are simple and efficient means to determine the pixels that need to be marked for re-computation; the reflections or refractions from dynamic objects influencing the intensity values of such pixels are handled during the actual re-computation process following later (see below). The pixel marks are stored in a 3D data structure, in which each pixel of every frame is represented by a single bit, indicating whether the pixel is marked ('1') or not ('0').

The results of this procedure are a rendered reference frame and the marking of all pixels different from the corresponding pixels of the reference frame. Now, each frame of the animation is computed by processing the original scene description without the DOBs. Prior to performing raytracing, it is checked whether the corresponding pixel in the current frame has been marked. If not, then the color of the reference frame is taken; otherwise, the raytracing algorithm is started to compute the pixel. To visualize a frame, the newly computed pixels are simply copied into (a copy of) the reference frame. Fig. 1 visualizes the results of the proposed approach by showing the reference frame and several snapshots of a simple animation in which two balls are fixed and others roll around them.

The reduction of computation time obtained with this method as compared to calculating each frame independently is proportional to the number of newly computed pixels. As already mentioned, the computational overhead of the method is small, and the memory space required for storing the data structure for marking the pixels is only one bit per pixel. This memory requirement is smaller than that of alternative incremental approaches based on binary shade trees [4, 11], and more importantly, does – in contrast to them – not grow with the complexity of the animation. However, the scene data is enlarged depending on how many DOBs are required for a particular animation. The method is independent of a particular raytracing algorithm, and it allows

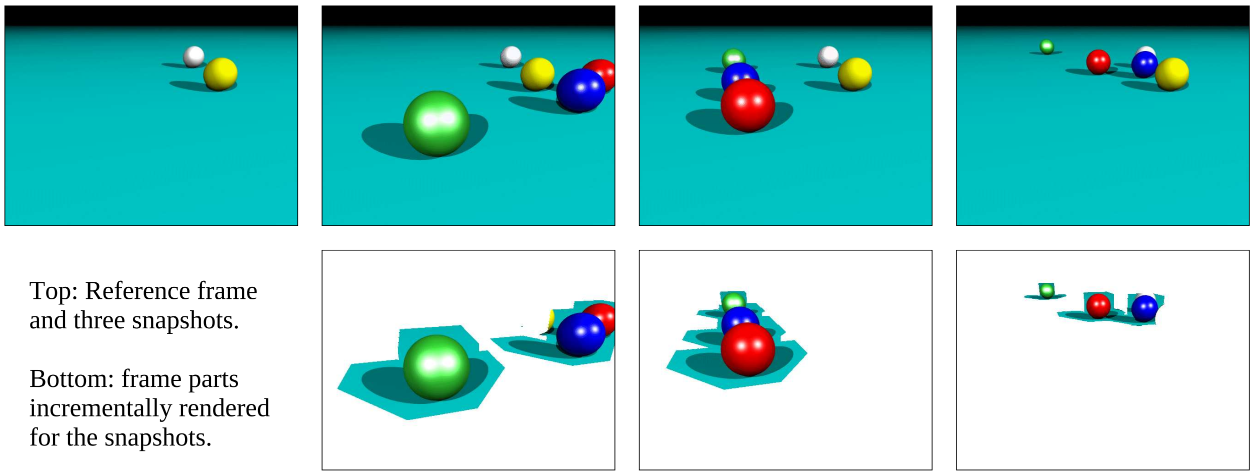
Top: Reference frame and three snapshots.
Bottom: frame parts incrementally rendered for the snapshots.

Figure 1: Reference frame and snapshots from the *Balls* animation.

to store a raytraced image either entirely by itself, or by a pair [*reference frame, set of pixels different to it*]. Clearly, as all incremental raytracing approaches, the proposed method does not offer its benefits when the camera moves or the light sources change during an animation. In this case, nearly all pixels of the affected frames have to be re-computed anyway.

3 Parallel Raytracing

Embarassingly parallel problems such as raytracing are typically approached by a *manager/worker* scheme in which a central manager process is given responsibility for the distribution of work and the processing of results. The partitioned subproblems are given to or requested by the worker processes, each of which is running on a different machine.

The implementation of a manager/worker scheme is particularly challenging in the context of a raytracing application performed on a workstation network. There are two main reasons: (a) each ray will take a different amount of processor time to compute, depending on the complexity of the intersections that occur, soon leading to load balancing problems, and (b) the computational capabilities of the various workstations are typically different, and the runtime behavior is influenced by the workload of other users. Thus, the aim of any manager/worker scheme must be to distribute work and use the available resources dynamically and adaptively.

This paper extends our previously published work in which an algorithm for parallel raytracing of single images has been presented [3]. That work relies on two fundamental properties, namely on maximizing the operation overlap between manager and workers, and on adaptively scheduling tasks to worker processes with "blind" consideration of heterogeneous execution speeds and irregularities of the scene itself. It basically implements a task parallel approach in which every worker process gets the complete information about the scene by parsing the scene description at startup time. Our parallel and incremental adaptation of the POV–Ray raytracing package [5] (based on POV–Ray's release 3.0) is called PiPov. It implements three basic operation modes, namely for rendering single images, for rendering animations (image sequences) completely, and for rendering animations incrementally. All three modes exploit parallel worker processes.

Rendering single images. The adaptive load balancing scheme for rendering single images is based on a recursive subdivision of the image data by initially assigning one half of the image in equally sized chunks of lines to the worker processes, hence assigning tasks to all workers as soon as possible. The second half of the image is dynamically assigned to the workers. Therefore, an agenda data structure is created in which task descriptions are stored. These task descriptions are created by recursive subdivision, starting with the second half as the new image of which the first half is divided in equally sized chunks of lines, while its second half is subject to treatment in the next level of recursion. This division into smaller and smaller tasks is performed until a certain threshold size is reached. Assignment of these tasks is performed in a first-come-first-serve manner: Every worker requests a new task as soon as it has completed its current one. By assigning the larger tasks first, fast workers will be assigned larger tasks than slower workers, leading to an adaptation of task sizes to worker speed and computational complexity of the tasks. This division scheme is illustrated in Fig. 2. Additionally, workers send computed image data (their results) line by line, as soon as the computation of a single image line has been completed. This behavior leads to additional operation overlap between manager and workers and has been shown to have superior runtime behavior compared to static as well as other dynamic task assignment strategies based on small, equally-sized tasks [3].

Rendering image sequences. PiPov's load balancing strategy for rendering animations follows the goals of its single–image mode. The first set of tasks is immediately distributed to the workers while further tasks are assigned on request: Whenever a worker completes its current task, it gets a new one assigned. Analogously, image data is sent line-by-line from the workers to the manager in order to overlap operations of both sides.

The main difference between the two modes for single images and for animations is task granularity. One of the lessons learned from [3] is to minimize communication overhead by keeping the number of generated tasks relatively small. Hence, for rendering animations, task granularity is based on complete image frames. Therefore, in a configuration with w workers and an animation with f frames to be computed, PiPov first generates $f - w$ tasks, each consisting of a single image. In order to provide a finer task granularity for the final computation phase, the last w frames are handed out as $2w$ tasks, each denoting half an image. In this mode for complete computation of image frames, the worker processes produce image data for every pixel of a frame.

Incremental rendering. PiPov's third operation mode is for parallel, incremental rendering of animations. Incremental rendering is performed analogous to the complete–rendering mode, except that (a) workers only compute those parts of an image frame that differ from the reference frame, and (b) the computation is divided into two phases. In the first phase, the reference frame is computed in parallel, exploiting PiPov's single–image mode. In the second phase, the dynamically changing parts of the images are computed as within the complete–rendering mode.

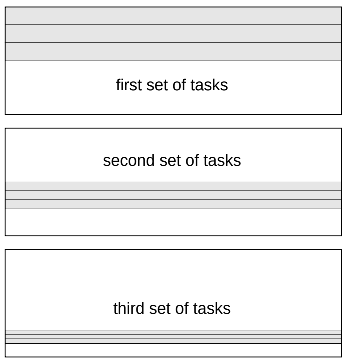

Figure 2: Adaptive partitioning for 3 workers.

4 Experimental Results

PiPov's runtime behaviour has been extensively evaluated on a network of 18 DEC Alpha workstations running Digital UNIX 4.0 and using the MPICH 1.1 implementation of the MPI message passing standard [12] as the communication library. Because our workstation network has been evolving throughout the last years, the machines available to our experiments are of different age and hence have

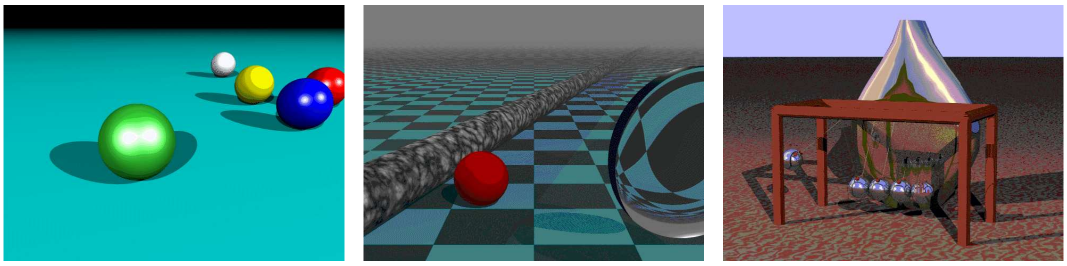

Figure 3: Snapshots from the test animations *Balls*, *Magglass*, and *Swing*.

different CPU clocks and span three generations of the processor chip set. Hence, care must be taken in order to compute expressive speedup values for parallel runs. Therefore, we have taken the speed index as it is computed by the WINNER resource management system [13] as the basis of our speedup computations. WINNER performs a benchmarking process on the machines of a network yielding speed index values relative to each other. We have normalized the speed index of the machine used for sequential computations to the value 1.0. Parallel runs have been performed with 12, 15, and 18 workstations. Their added CPU speeds result in values of 12.7, 15.7, and 18.7, respectively.

PIPOV has been evaluated with three test animations, snapshots of which are shown in Fig. 3. The first one, *Balls*, is relatively simple. Here, two balls remain fixed while others roll around them. This test animation needs a relatively short computation time and lends itself well to incremental rendering, because large parts of the scene remain the same within all frames. The second animation, *Magglass*, shows a checkered ground over which a ball and a magnifying glass move from left to right and vice versa. This animation requires more computation time due to the effects visible inside the magnifying glass. Furthermore, large parts of the scene change in every frame of the animation. Finally, the *Swing* animation shows a chain of metal balls swinging from left to right in front of a reflecting vase. This animation needs the most computation time, but has larger constant parts of the scene.

All three animations have been rendered in two sizes, namely 320 × 240 pixels (commonly used with MPEG-1) and 768×576 pixels (commonly used with MPEG-2). Because both sizes yielded similar results, we will only discuss the results for the larger image size below. The animations have been rendered with 50, 150, and 250 frames each. All results presented below are average values over multiple runs. Table 1 summarizes all measured application runtimes. It shows results for 1, 12, 15, and 18 workstations, comparing incremental and complete rendering with each other.

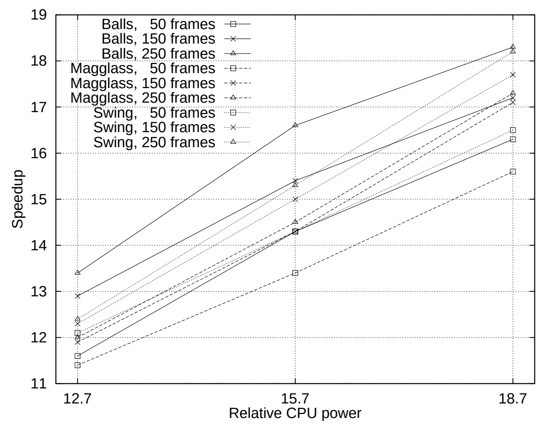

Figure 4: Speedup of complete animation rendering.

Fig. 4 shows the speedup achieved by parallel rendering of complete image frames, relative to sequentially computing complete frames on the machine with speed index 1.0. As can be seen from the figure, all tests show almost linear speedups, with gradually better results for the longer animation sequences. A few results show even super-linear speedups, indicating thrashing problems of the machine used for

Table 1: Application runtimes (in seconds) for 1, 12, 15, and 18 workstations.

Scene	Frames	1 cmp.	1 inc.	12 cmp.	12 inc.	15 cmp.	15 inc.	18 cmp.	18 inc.
Balls	50	6245	1224	538	102	436	89	384	87
Balls	150	18942	3061	1468	261	1229	216	1104	225
Balls	250	33754	5353	2511	433	2035	357	1845	303
Magglass	50	9896	5964	869	525	738	463	633	432
Magglass	150	29838	17783	2504	1410	2094	1119	1749	985
Magglass	250	49807	29585	4145	2311	3438	2062	2886	1560
Swing	50	17968	5911	1487	537	1257	461	1090	412
Swing	150	53823	18305	4375	1537	3586	1260	3040	1053
Swing	250	90208	30204	7248	2429	5913	1963	4958	1726

sequential runs, which is equipped with 64 MB of real memory only.

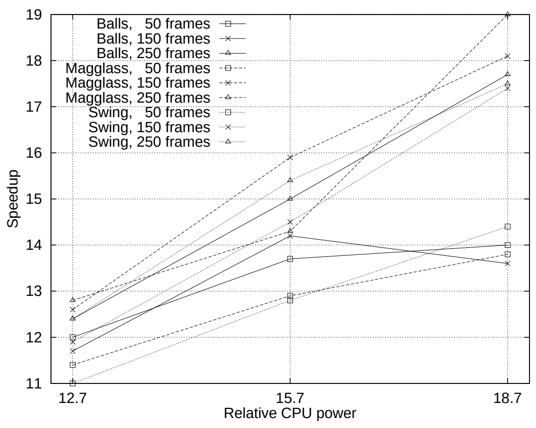

Figure 5: Speedup of incremental animation rendering.

Fig. 5 shows the speedup achieved by parallel incremental raytracing, relative to incremental raytracing performed sequentially on the machine with speed index 1.0. Here, two groups of results can be identified. The first group yields almost linear speedup as with complete rendering, indicating the efficiency of our incremental approach. The second group, consisting of the short animations with 50 frames and the simple *Balls* animation with 150 frames still shows considerable but significantly less speedup. For the short animations, this result indicates the overhead generated by computing the additional reference frame. Because with only 50 frames, each worker will (on the average) compute just 5 frames, and thus this one–frame overhead slightly decreases the achievable speedup. For the *Balls* animation with 150 frames this result means that our incremental approach reduces the necessary computations very efficiently such that the break–even point has already been reached, after which further addition of computational power will not improve the speedup.

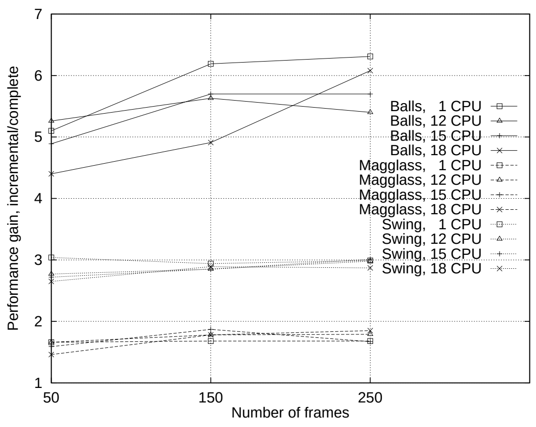

Figure 6: Performance gain of incremental w.r.t. complete rendering.

The conclusions concerning the efficiency of our incremental algorithm drawn so far are backed very well by Fig. 6. In general, the ratio of pixels that change throughout an animation compared to its total number of pixels is a static property of a given animation. As demonstrated by Fig. 6, the relations between runtimes of complete and incremental raytracing of the *Magglass* and *Swing* animations show this constant ratio independent of the numbers of frames to be computed or CPUs in use. Only the *Balls* animation shows some variation of this relation, which is due to its low computational requirements, but gradually also shows a constant ratio. These results indicate that overheads introduced by our incremental algorithm (e.g. the additional reference frame and additional data structures) hardly influence the runtime efficiency of the rendering process. Table 1 shows that the combination of incremental rendering with parallel computations leads to very short application runtimes compared to the original POV–Ray program.

5 Conclusions

In this paper, a new incremental raytracing method for rendering animations was proposed, based on localizing the influences of moving or changing objects by using an adequate bookkeeping data structure for recording incremental pixel changes. This method was executed in parallel on a network of workstations using a manager/worker scheme which incorporated dynamic task assignment to achieve load balancing. Experimental results have shown that the proposed incremental method is able to reduce the computation time roughly proportional to the number of changed pixels in a particular animated scene. Almost linear speedups were obtained for the parallel versions running on up to 18 workstations.

There are several issues for future research, such as (a) extending the scene parser to automatically enable and disable incremental rendering for parts of an animation depending on changing camera positions and lighting conditions, (b) finding optimal shapes for DOBs in order to reduce the number of intersection points, and (c) investigating further manager/worker schemes for dynamic load distribution.

References

[1] A. Glassner. *An Introduction to Ray Tracing.* Academic Press, 1989.

[2] T. Crockett. An Introduction to Parallel Rendering. *Parallel Computing*, 23:819–843, 1997.

[3] B. Freisleben, D. Hartmann, and T. Kielmann. Parallel Raytracing: A Case Study on Partitioning and Scheduling on Workstation Clusters. In *Proc. HICSS'30*, Vol. 1, pp. 596–605, Wailea, HI, 1997. IEEE.

[4] K. Hirota and K. Murakami. Incremental Ray Tracing. In *Proc. Eurographics Workshop on Photosimulation, Realism, and Physics in Computer Graphics*, 1995.

[5] A. Enzmann, L. Kretzschmar, and C. Young. *Ray Tracing Worlds with POV–Ray.* The Waite Group, 1994.

[6] S. Badt. Two Algorithms for Taking Advantage of Temporal Coherence in Ray Tracing. *The Visual Computer*, 4(1):123–132, 1988.

[7] A. Glassner. Spacetime Ray Tracing for Animation. *IEEE Computer Graphics and Applications*, 8(2):60–70, 1988.

[8] H. Müller. Time Coherence in Computer Animation by Ray Tracing. *Proc. Int. Workshop on Computational Geometry and its Applications*, LNCS 333, pp. 187–201, Springer-Verlag, 1988.

[9] D. Jevans. Object Space Temporal Coherence for Ray Tracing. In *Proc. Comp. Graphics Interface'92*, pp. 176–183, 1992.

[10] T.A. Davis and E.W. Davis. Rendering Computer Animations on a Network of Workstations. In *Proc. IPPS/SPDP 1998*, pp. 726–730.

[11] S.Horiguchi, M.Katahira, and T.Nakada. Parallel Processing of Incremental Ray Tracing on a Shared-Memory Multiprocessor. *The Visual Computer*, 9:371–380, 1993.

[12] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message–Passing Interface.* MIT Press, 1994.

[13] O. Arndt, B. Freisleben, T. Kielmann, and F. Thilo. Dynamic Load Distribution with the Winner System. In *Workshop ALV'98*, pp. 77 – 88, Munich, Germany, 1998.