Parallel Point Reprojection

Erik Reinhard Peter Shirley Charles Hansen

University of Utah www.cs.utah.edu

Abstract

Improvements in hardware have recently made interactive ray tracing practical for some applications. However, when the scene complexity or rendering algorithm cost is high, the frame rate is too low in practice. Researchers have attempted to solve this problem by caching results from ray tracing and using these results in multiple frames via reprojection. However, the reprojection can become too slow when the number of samples that are reused is high, so previous systems have been limited to small images or a sparse set of computed pixels. To overcome this problem we introduce techniques to perform this reprojection in a scalable fashion on multiple processors.

CR Categories: I.3.7 [Computing Methodologies]: Computer Graphics—3D Graphics

Keywords: point reprojection, ray tracing

1 Introduction

Interactive Whitted-style ray tracing has recently become feasible on high-end parallel machines [5, 6]. However, such systems only maintain interactivity for relatively simple scenes or small image sizes.

By reusing samples instead of relying on brute force approaches, these limitations can be overcome. There are several ways to reuse samples. All of them require interpolating between existing samples as the key part of the process. First, rays can be stored along with the color seen along them. The color of new rays can be interpolated from existing rays [1, 4]. Alternatively, the points in 3D where rays strike surfaces can be stored and then woven together as displayable surfaces [7]. Finally, such points can be directly projected to the screen, and holes can be filled in using image processing heuristics [8].

Another method to increase the interactivity of ray tracing is *frameless rendering* [2, 3, 6, 9]. Here, a master processor farms out single pixel tasks to be traced by the slave processors. The order in which pixels are selected is random or quasi-random. Whenever a renderer finishes tracing its pixel, it is displayed directly. As pixel updates are independent of the display, there is no concept of frames. During camera movements, the display will deteriorate somewhat, which is visually preferable to slow frame-rates in frame-based rendering approaches. It can therefore handle scenes of higher complexity than brute force ray tracing, although no samples are reused.

The main thrust of this paper is the use of parallelism to increase data reuse and thereby increase allowable scene complexity and image size without affecting perceived update rates. We use the *render cache* of Walter et al. [8] and apply to it the concept of frameless rendering. By distributing this algorithm over many processors we are able to overcome the key bottleneck in the original render cache work. We demonstrate our system on a variety of scenes and image sizes that have been out of reach for previous systems.

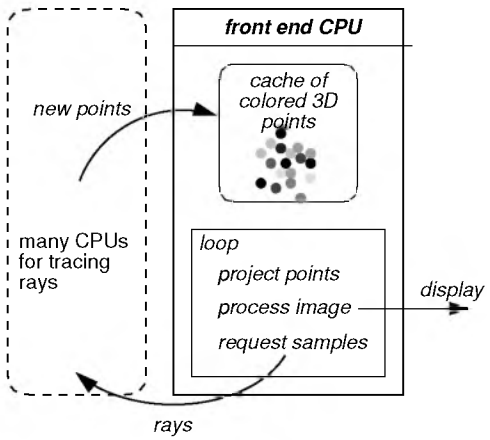

Figure 1: *The serial render cache algorithm [8].*

2 Background: the render cache

The basic idea of the render cache is to save samples in a 3D point cloud, and reproject them when viewing parameters change [8]. New samples are requested all over the screen, with most samples concentrated near depth discontinuities. As new samples are added old samples are eliminated from the point cloud.

The basic process is illustrated in Figure 1. The front-end CPU handles all tasks other than tracing rays. Its key data structure is the cache of colored 3D points. The front end continuously loops, first projecting all points in the cache into screen space. This will produce an image with many holes, and the image is processed to fill these holes in. This filling-in process uses sample depths and heuristics to make the processed image look reasonable. The processed image is then displayed on the screen. Finally, the image is examined to find "good" rays to request to improve future images. These new rays are traced by the many CPUs in the "rendering farm". The current frame is completed after the front end receives the results and inserts them into the point cloud.

From a parallel processing point of view, the render cache has the disadvantage of a single expensive display process that needs to feed a number of renderers with sample requests and is also responsible for point reprojection. The display process needs to insert new results into the point cloud, which means that the more renderers are used, the heavier the workload of the display process. Hence, the display process quickly becomes a bottleneck. In addition, the number of points in the point cloud is linear in image size, which means that the reprojection cost is linear in image size.

The render cache was shown to work well on 256x256 images using an SGI Origin 2000 with 250MHz R10k processors. At higher resolutions than 256x256, the front end has too many pixels to reproject to maintain fluidity.

3 Distributed render cache

Ray tracing is an irregular problem, which means that the time to compute a ray task can vary substantially depending on depth complexity. For this reason it is undesirable to run a parallel ray tracing algorithm synchronously, as this would slow down rendering of each frame to be as slow as the processor which has the most expensive set of tasks. On the other hand, synchronous operation would allow a parallel implementation of the render cache to produce exactly the same artifacts as the original render cache. We have chosen responsiveness and speed of operation over minimization of artifacts by allowing each processor to update the image asynchronously.

Our approach is to distribute the render cache functionality with the key goal of not introducing synchronization, which is analogous to frameless rendering. In our system there will be a number of renderers which will reproject point clouds and render new pixels, thereby removing the bottleneck from the original render cache implementation. Scalability is therefore assured.

We parallelize the render cache by subdividing the screen into a number of tiles. A random permutation of the list of tiles could be distributed over the processors, with each renderer managing its set of tiles independently from all other renderers. Alternatively, a global list of tiles could be maintained with each processor choosing the tile with the highest priority whenever it needs a new task to work on. While the latter option may provide better (dynamic) load balancing, we have opted for the first solution. Load balancing is achieved statically by ensuring that each processor has a sufficiently large list of tiles. The reason for choosing a static load balancing scheme has to do with memory management on the SGI Origin 3800, which is explained in more detail in Section 4.

Each tile has associated with it a local point cloud and an image plane data structure. The work associated with a tile depends on whether or not camera movement is detected. If the camera is moving, the point cloud is projected onto the tile's local image plane and the results are sent to the display thread for immediate display. No new rays are traced, as this would slow down the system and the perceived smoothness would be affected. This is at the cost of a degradation in image quality, which is deemed more acceptable than a loss of interaction. It is also the only modification we have applied to the render cache concept.

If there is no camera movement, a depth test is performed to select those rays that would improve image quality most. Other heuristics such as an aging scheme applied to the points in the point cloud also aid in selecting appropriate new rays. Newly traced rays are both added to the point cloud and displayed on screen. The point cloud itself does not need to be reprojected.

The renderers each loop over their alotted tiles, executing for each tile in turn the following main components:

1. Clear tile Before points are reprojected, the tile image is cleared.

2. Add points Points that previously belonged to a neighbouring tile but have been projected onto the current tile are added to the point cloud.

3. Project point cloud The point cloud is projected onto the tile image. Points that project outside the current tile are temporarily buffered in a data structure that is periodically communicated to the relevant tiles.

4. Depth test A depth test is performed on the tile image to determine depth discontinuities. This is then used to select rays to trace.

5. Trace rays The rays selected by the depth test function, are traced.

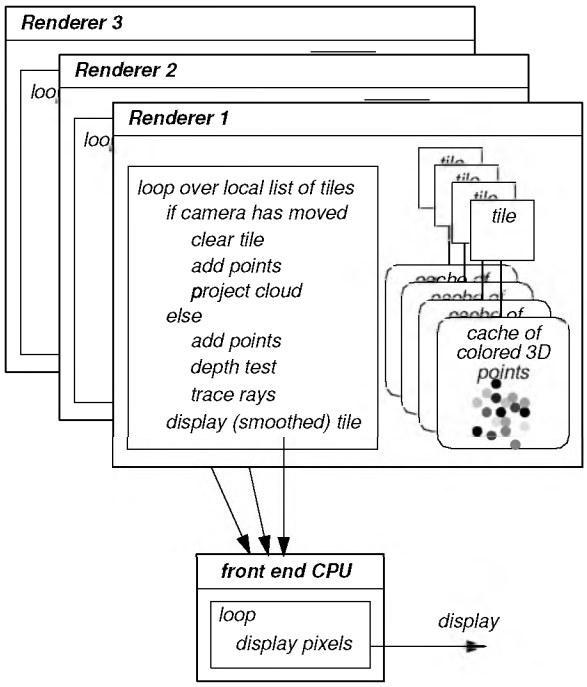

Figure 2: *The parallel render cache algorithm.*

6. Display tile The resulting tile is communicated to the display thread. This function also performs hole-filling to improve the image's visual appearance.

If camera movement has occurred since a tile was last visited, items 1, 2, 3 and 6 in this list are executed for that tile. If the camera was stationary, items 1, 2, 3 and 6 are executed. The algorithm is graphically depicted in Figure 2

While tiles can be processed largely independently, there are circumstances when interaction between tiles is necessary. This occurs for instance when a point in one tile's point cloud projects to a different tile (due to camera movement). In that case, the point is removed from the local point cloud and is inserted into the point cloud associated with the tile to which it projects. The more tiles there are, the more often this would occur. This conflicts with the goal of having many tiles for load balancing purposes. In addition, having fewer tiles that are larger causes tile boundaries to be more visible.

As each renderer produces pixels that need to be collated into an image for display on screen, there is still a display process. This display thread only displays pixels and reads the keyboard for user input. Displaying an image is achieved by reading an array of pixels that represents the entire image, and sending this array to the display hardware using OpenGL. When renderers produce pixels, they are buffered in a local data structure, until a sufficient number of pixels has been accumulated for a write into the global array of pixels. This buffering process ensures that memory contention is limited for larger image sizes.

Finally, the algorithm shows similarities with the concept of frameless rendering, in the sense that tiles are updated independently from the display process. If the size of the tiles is small with respect to the image size, the visual effect is like that of frameless rendering. The larger the tile size is chosen, the more the image updating process starts to look like a distributed version of the render cache.

4 Implementation

The parallel render cache algorithm is implemented on a 32 processor SGI Origin 3800. While this machine has a 16 GB shared address space, the memory is physically distributed over a total of eight nodes. Each node features four 400 MHz R12k processors and one 2 GB block of memory. In addition each processor has an 8 MB secondary cache. Memory access times are determined by the distance between the processor and the memory that needs to be read or written. The local cache is fastest, followed by the memory associated with a processor's node. If a data item is located at a different node, fetching it may incur a substantial performance penalty.

A second issue to be addressed is that the SGI Origin 3800 may relocate a rendering process with a different processor each time a system call is performed. Whenever this happens, the data that used to be in the local cache is no longer locally available. Cache performance can thus be severely reduced by migrating processes.

These issues can be avoided on the SGI Origin 3800 by actively placing memory near the processes and disallowing process migration. This can, for example, be accomplished using the *dplace* library. Associated with each tile in the parallel render cache is a local point cloud data structure and an image data structure which are mapped as close as possible to the process that uses it. Such memory mapping assures that if a cache miss occurs for any of these data structures, the performance penalty will be limited to fetching a data item that is in local memory. As argued above, this is much cheaper than fetching data from remote nodes. For this reason, using a global list of tiles as mentioned in the previous section is less efficient than distributing tiles statically over the available processors.

Carefully choreographing the mapping of processes to processors and their data structures to local memory enhances the algorithm's performance. Cache performance is improved and the number of data fetches from remote locations is minimized.

Figure 3: *Test scenes. The teapot (top) consists of 32 bezier patches, while the room scene consists of 846,563 primitives and 80 point light sources.*

5 Results

Our implementation uses the original render cache code of Walter et al [8] . The main loop of the renderer consists of a number of distinct steps, each of which are measured separately. To assess scalability, the time to execute each step is measured, summed over all invocations and processors and subsequently divided by the number of invocations and processors. The result is expressed in events per second per processor, which for a scalable system should be independent of the number of processors employed. In case this measure varies with processor count, scalability is affected.

If the number of events per second per processor drops when adding processors, sublinear scalability is measured, whereas an increase indicates super-linear speed-up for the measured function. Also note that the smaller the number, the more costly the operation will be. Using this measure provides better insight into the behaviour of the various parts of the algorithm than a standard scalability computation would give, especially since only a subset of the components of the render cache algorithm is executed during each iteration.

Two test scenes were used: a teapot with 32 bezier patches[1] and one point light source, and a room scene with 846,563 geometric primitives and area light sources approximated by 80 point light sources (Figure 3). For the teapot scene, the renderer is limited by the point reprojection algorithm, while for the room scene, tracing new rays is the slowest part of the algorithm. The latter scene is of typical complexity in architectural applications and usually cannot be interactively manipulated.

In the following subsection, the different components making up the parallel render cache are evaluated (Section 5.1), the performance as function of task size is assessed (Section 5.2) and the parallel render cache is compared with other methods to speed up interactive ray tracing (Section 5.3).

5.1 Parallel render cache evaluation

The results of rendering the teapot and room models on different numbers of processors at a resolution of 512^2 and 1024^2 pixels are depicted in Figures 4 and 5.

While most of the components making up the algorithm show horizontal lines in these graphs, meaning that they scale well, the "Clear tiles" and "Add point" components show non-linear behaviour. Clearing tiles is a very cheap operation which appears to become cheaper if more processors are used. Because more processors result in each processor having to process fewer tiles, this super-linear behaviour may be explained by better cache performance. This effect is less pronounced for the 1024^2 pixel renderings, which also points to a cache performance issue as here each processor handles more data.

The "Add point" function scales sub-linearly with the number of processors. Because the total number of tiles was kept constant between runs, this cannot be explained by assuming that different numbers of points project outside their own tile and thus have to be added to neighbouring tiles. However, with more processors there is an increased probability that a neighbouring tile belongs to

[1] These bezier patches are rendered directly using the intersection algorithm from Parker et. al [6].

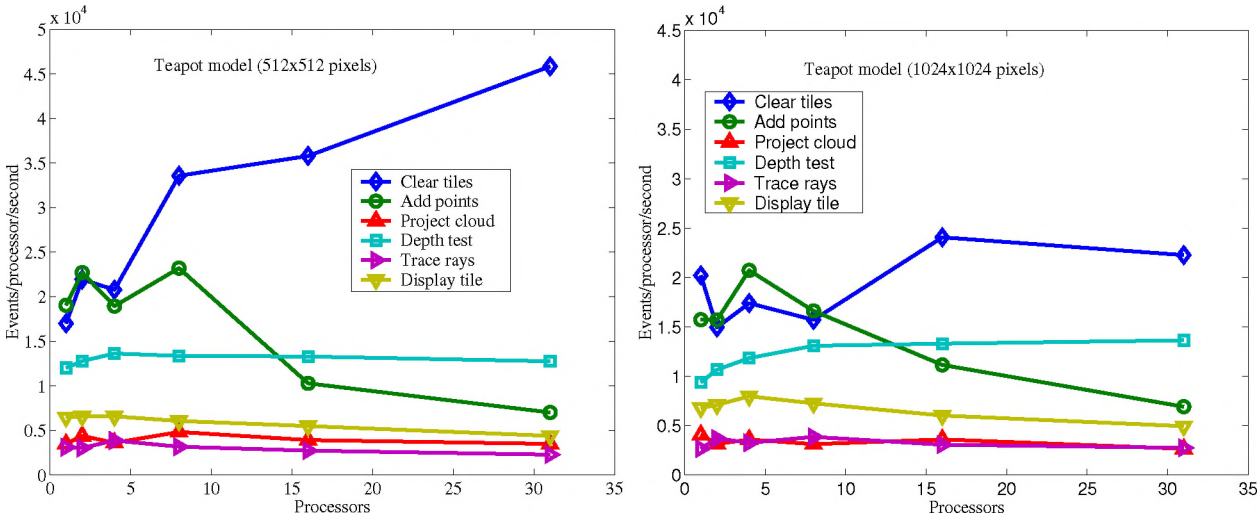

Figure 4: *Scalability of the render cache components for the teapot scene rendered at 512^2 pixels (left) and 1024^2 pixels (right). Negative slopes indicate sub-linear scalability, whereas horizontal lines show linear speed-ups.*

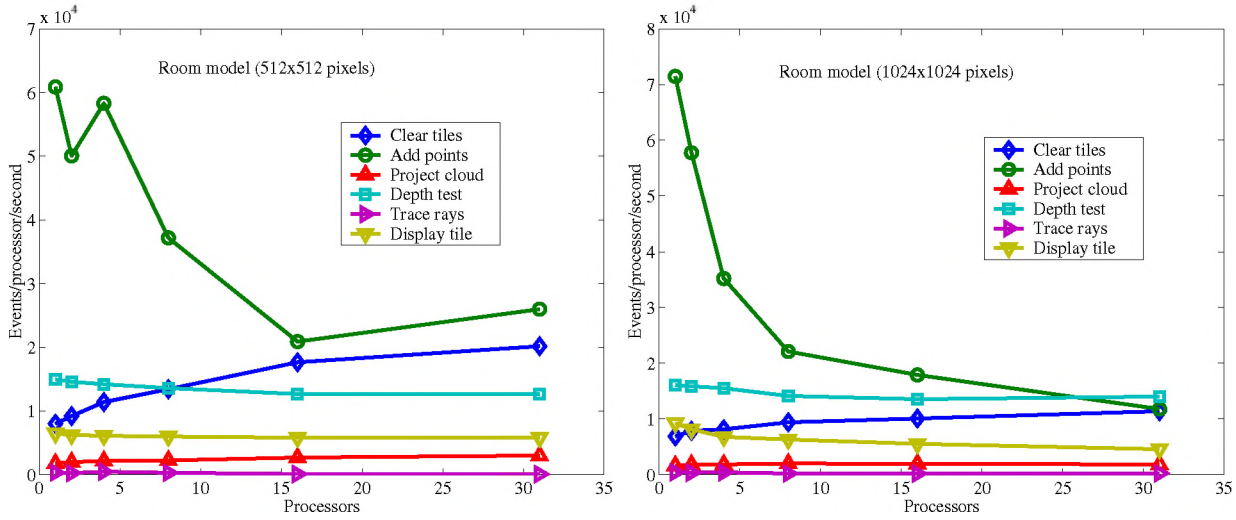

Figure 5: *Scalability for the room scene, rendered at 512^2 pixels (left) and 1024^2 pixels (right). Horizontal lines indicate linear scalability, whereas a fall-off means sub-linear scalability.*

a different processor and may therefore reside in memory which is located elsewhere in the machine. Thus projecting a point outside the tile that it used to belong to, may become more expensive for larger numbers of processors. This issue is addressed in the following section.

Note also that despite the poor scalability of "Add points", in absolute terms its cost is rather low, especially for the room model. Hence, the algorithm is bounded by components that scale well (they produce more or less horizontal lines in plots) and therefore the whole distributed render cache algorithm scales well, at least up to 31 processors (see also Section 5.3). In addition, the display of the results is completely decoupled from the renderers which produce new results and therefore the screen is updated at a rate that is significantly higher than rays can be traced and is also much higher than points can be reprojected. This three-tier system of producing new rays at a low frequency, projecting existing points at an intermediate frequency and displaying the results at a high frequency (on the Origin 3800 at a rate of around 290 frames per second for 512^2 images and 75 frames per second for 1024^2 images, regardless of number of renderers and scene complexity) ensures a smooth display which is perceived as interactive, even if new rays are produced at a rate that would not normally allow interactivity.

By abandoning ray tracing altogether during camera movement, the system shows desirable behavior even when fewer than 31 processors are used. For both the room scene and the teapot model, the camera can move smoothly if 4 or more processors are used. During camera movement, the scene deteriorates because no new rays are produced and holes in the point cloud may become visible. During rapid camera movement, tile boundaries may become temporarily visible. After the camera has stopped moving, these artifacts disappear at a rate that is linear in the number of processors employed. We believe that maintaining fluid motion is more important than the temporary introduction of some artefacts, which is why the distributed render cache is organised as described above.

For those who would prefer a more accurate display at the cost of a slower system response, it would be possible to continue trac-

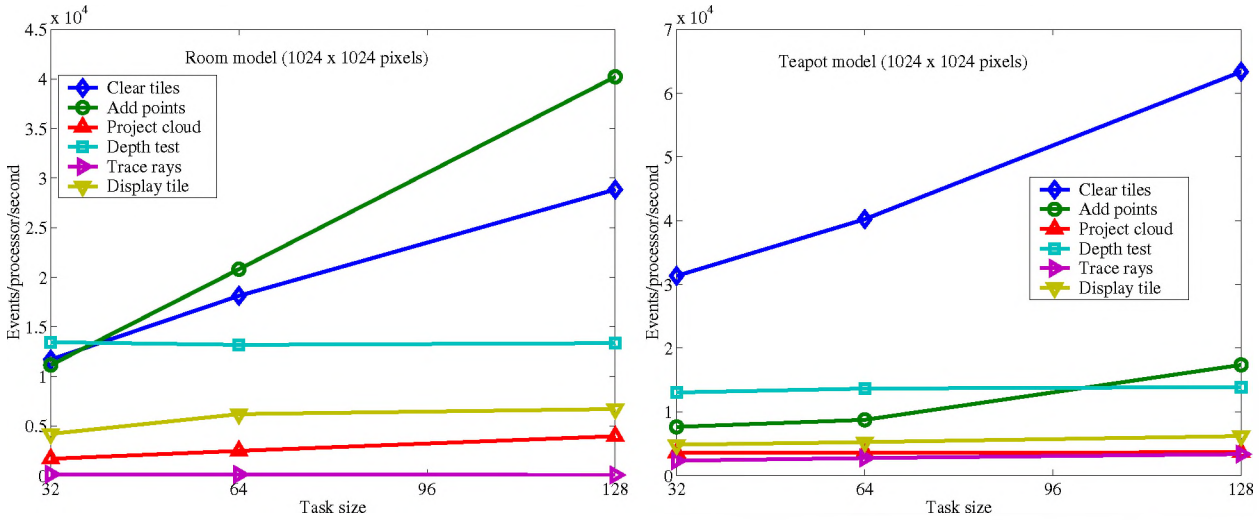

Figure 6: *Scalability for the room model (left) and teapot scene (right) as function of tile size (32^2, 64^2 and 128^2 pixels per tile). The image size is 1024^2 pixels and for these measurements 31 processors were used. These graphs should be interpreted the same as those in Figures 4 and 5.*

ing rays during camera movement. Although the render cache then behaves differently, the scalability of the separate components, as given in Figures 4 and 5, would not change. However, the fluidity of camera movement is destroyed by an amount dependent on scene complexity.

5.2 Task size

In section 3 it was argued that the task size, i.e. the size of the tiles, is an important parameter which defines both speed and the occurrence of visual artefacts. The larger the task size, the better artefacts become visible. However, at the same time, the reprojections that cross tile-boundaries are less likely to occur, resulting in higher performance. In Figure 6 the scalability of the parallel render cache components as function of task size is depicted. Task sizes range from 32^2 pixels to 128^2 pixels and the measurements were all obtained using 31 processors on 1024^2 images. Larger tile sizes are thus impossible, as the total number of tasks would become smaller than the number of processors. Task sizes smaller than 32^2 pixels resulted in unreasonably slow performance and were therefore left out of the assessment.

As in the previous section, the "Add points" and "Clear tile" components show interesting behavior. As expected, for larger tasks, the "Add points" function becomes cheaper. This is because the total length of the tile boundaries diminishes for larger task sizes, and so the probability of reprojections occuring accros tile boundaries is smaller.

The "Clear tile" component also becomes less expensive for larger tiles. Here, we suspect that resetting one large block of memory is less expensive than resetting a number of smaller blocks of memory.

Although Figure 6 suggests that choosing the largest task size as possible would be appropriate, the artefacts visible for large tiles are more unsettling than for smaller task sizes. Hence, for all other experiments presented in this paper, a task size of 32^2 pixels is used, which is based on an assessment of both artefacts and performance.

5.3 Comparison with other speed-up mechanisms

In this section, the parallel render cache is compared with other state-of-the-art rendering techniques. All make use of the interactive ray tracer of Parker et. al. [6], either as a back-end or as the main algorithm. The comparison includes the original render cache algorithm [8], the parallel render cache algorithm as described in this paper, the interactive ray tracer (rtrt) without reprojection techniques and the interactive ray tracer using the frameless rendering concept [6]. In the following we will refer to the original render cache as "serial render cache" to distinguish it from our parallel render cache implementation. All renderings were made using the teapot and room models (Figure 3) at a resolution of 1024^2 pixels.

The measurements presented in this section consist of the number of new samples produced per second by each of the systems and the number of points reprojected per second (for the two render cache algorithms). These numbers are summed over all processors and should therefore scale with the number of processors employed. The results for the teapot model are given in Figure 7 and the results for the room model are presented in Figure 8.

The graphs on the left of these figures show the number of samples generated per second. All the lines are straight, indicating scalable behaviour. In these plots, steeper lines are the result of higher efficiency and therefore, the real-time ray tracer would be most efficient, followed by the parallel render cache. The frameless rendering concept looses efficiency because randomising the order in which pixels are generated destroys cache coherence. The parallel render cache does not suffer from this, since the screen is tiled and tasks are based on tiles. The serial render cache appears to perform well for complex scenes and poorly for simple scenes. For scenes that lack complexity, the point reprojection front-end becomes the bottleneck, especially since the image size chosen causes the point cloud to be quite large. Thus, the render cache front-end needs to reproject a large number of points for each frame and so constitutes a bottleneck.

Although the parallel render cache does not produce as many new pixels as the real-time ray tracer by itself does, this loss of efficiency is compensated by its ability to reproject large numbers of points, as is shown in the plots on the right of Figures 7 and 8. The point reprojection component of the parallel render cache shows good scalability, and therefore the goal of parallelising the render cache algorithm is reached. The point reprojection part of the serial render cache does not scale because it is serial in nature.

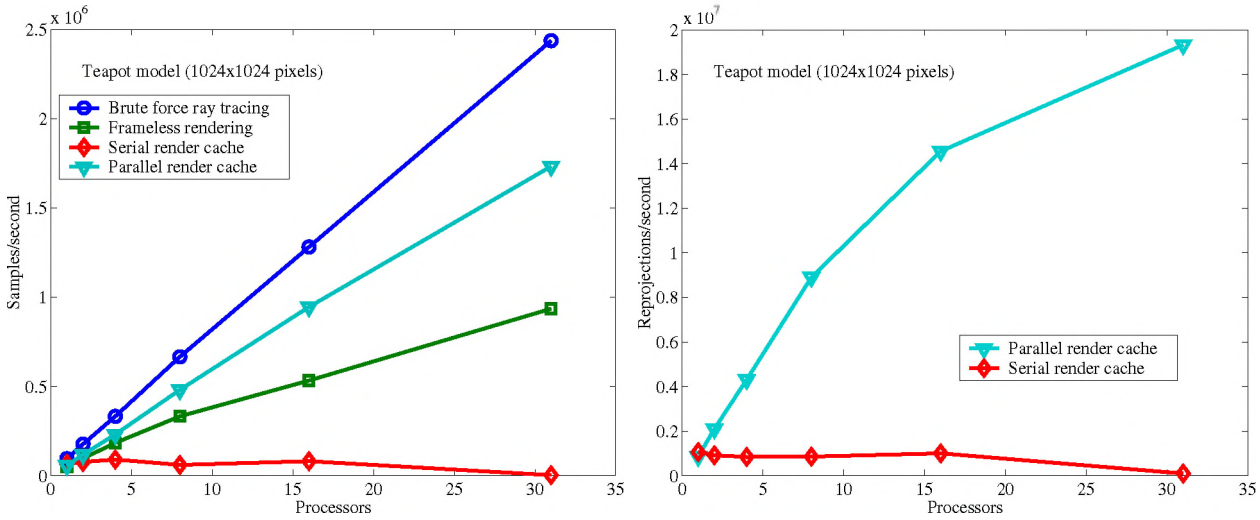

Figure 7: *Samples per second (left) and point reprojections per second (right) for the teapot model.*

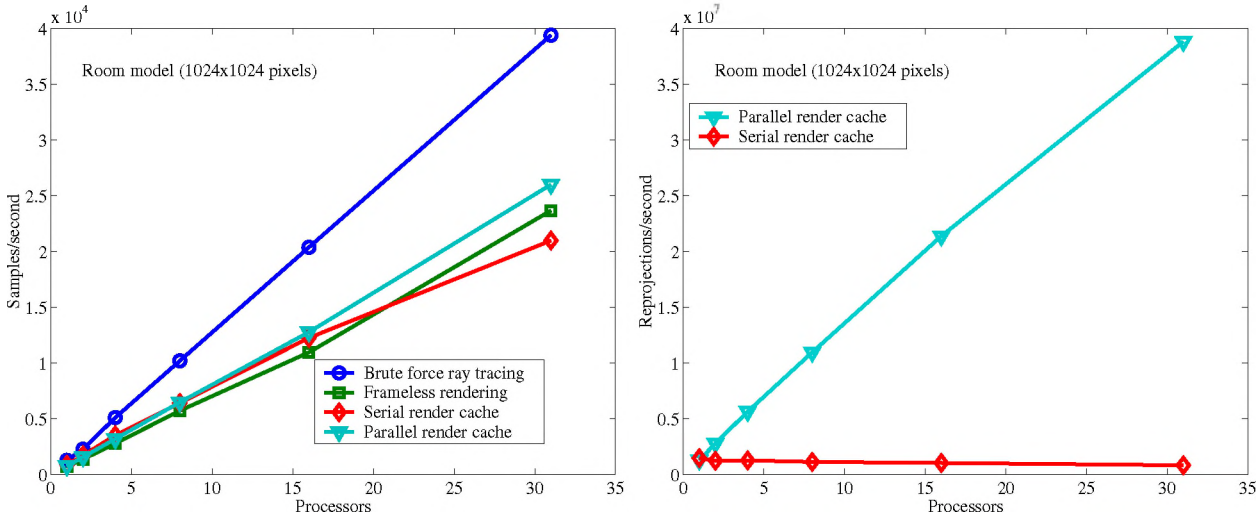

Figure 8: *Samples per second (left) and point reprojections per second (right) for the room scene.*

6 Discussion

While it is true that processors get ever faster and multi-processor machines are now capable of real-time ray tracing, scenes are getting more and more complex while at the same time frame sizes still need to increase. Hence, Moore's law is not likely to allow interactive full-screen brute-force ray tracing of highly complex scenes anytime soon.

Interactive manipulation of complex models is still not possible without the use of sophisticated algorithms that can efficiently exploit temporal coherence. The render cache is one such algorithm that can achieve this. However, for it not to become a bottleneck itself, the render cache functionality needs to be distributed over the processors that produce new samples. The resulting algorithm, presented in this paper, shows superior reprojection capabilities that enables smooth camera movement, even in the case where the available processing power is much lower than would be required in a brute force approach. It achieves this for scene complexities and image resolutions that are not feasible using any of the other algorithms mentioned in the previous section.

While smoothness of movement is an important visual cue, our algorithm necessarily produces other artifacts during camera motion. These artifacts are deemed less disturbing than jerky motion and slow response times. The render cache attempts to fill small holes after point reprojection. For larger holes, this may fail and unfilled pixels may either be painted in a fixed color, or can be left unchanged from previous reprojections. Either approach causes artefacts inherent to the algorithm and is present both in the original render cache and in our parallel implementation of it.

The parallel render cache produces additional artefacts due to the tiling scheme employed. During camera movement, tile boundaries may temporarily become visible, because there is some latency between points being reprojected from neighbouring tiles and this reprojection becoming visible in the current tile. A further investigation to minimize these artifacts is in order, which we reserve for future work. Currently, the parallel render cache algorithm is well suited for navigation through highly complex scenes to find appropriate camera positions.

It has been shown that even with a relatively modest number of processors, the distributed render cache can produce smooth camera movement at resolutions typically sixteen times higher than the

original render cache. The system as presented here scales well up to 31 processors. Its linear behavior suggests that improved performance is likely beyond 31 processors, although if this many processors are available, it would probably become sensible to devote the extra processing power to produce more samples, rather than increase the speed of reprojection.

Acknowledgements

The authors are extremely grateful to Bruce Walter, George Drettakis and Steven Parker for making their render cache source code available to us. We would also like to thank John McCorquodale for memory and processor placement discussions. This work was partially supported by NSF grants 97-96136, 97-31859, 98-18344, 99-77218, 99-78099 and by the DOE AVTC/VIEWS.

References

[1] K. BALA, J. DORSEY, AND S. TELLER, *Radiance interpolants for accelerated bounded-error ray tracing*, ACM Transactions on Graphics, 18 (1999), pp. 213–256.

[2] G. BISHOP, H. FUCHS, L. MCMILLAN, AND E. J. SCHER ZAGIER, *Frameless rendering: Double buffering considered harmful*, in Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994), A. Glassner, ed., July 1994, pp. 175–176.

[3] R. A. CROSS, *Interactive realism for visualization using ray tracing*, in Proceedings Visualization '95, 1995, pp. 19–25.

[4] G. W. LARSON AND M. SIMMONS, *The holodeck ray cache: An interactive rendering system for global illumination in non-diffuse environments*, ACM Transactions on Graphics, 18 (October 1999), pp. 361–368.

[5] M. J. MUUSS, *Towards real-time ray-tracing of combinatorial solid geometric models*, in Proceedings of BRL-CAD Symposium, June 1995.

[6] S. PARKER, W. MARTIN, P.-P. SLOAN, P. SHIRLEY, B. SMITS, AND C. HANSEN, *Interactive ray tracing*, in Symposium on Interactive 3D Computer Graphics, April 1999.

[7] M. SIMMONS AND C. SÉQUIN, *Tapestry: A dynamic mesh-based display representation for interactive rendering*, in Proceedings of the 11th Eurographics Workshop on Rendering, Brno, Czech Republic, June 2000, pp. 329–340.

[8] B. WALTER, G. DRETTAKIS, AND S. PARKER, *Interactive rendering using the render cache*, in Rendering Techniques '99, D. Lischinski and G. W. Larson, eds., Eurographics, Springer-Verlag Wien New York, 1999, pp. 19–30.

[9] E. S. ZAGIER, *Defining and refining frameless rendering*, Tech. Rep. TR97-008, UNC-CS, July 1997.