Interactive Rendering using the Render Cache

Bruce Walter†, George Drettakis†, Steven Parker‡

†*i*MAGIS[1]-GRAVIR/IMAG-INRIA
B.P. 53, F-38041 Grenoble, Cedex 9, France
‡University of Utah

Abstract.

Interactive rendering requires rapid visual feedback. The *render cache* is a new method for achieving this when using high-quality pixel-oriented renderers such as ray tracing that are usually considered too slow for interactive use. The render cache provides visual feedback at a rate faster than the renderer can generate complete frames, at the cost of producing approximate images during camera and object motion. The method works both by caching previous results and reprojecting them to estimate the current image and by directing the renderer's sampling to more rapidly improve subsequent images.

Our implementation demonstrates an interactive application working with both ray tracing and path tracing renderers in situations where they would normally be considered too expensive. Moreover we accomplish this using a software only implementation without the use of 3D graphics hardware.

1   Introduction

In rendering, interactivity and high quality are often seen as competing and even mutually exclusive goals. Algorithms such as ray tracing [29] and path tracing [14] are widely used to produce high-quality, visually compelling images that include complex effects such as reflections, refraction, and global illumination. However, they have generally been considered too computationally expensive for interactive use.

Interactive use has typically been limited to lower-quality, often hardware accelerated rendering algorithms such as wireframe or scan-conversion. While these are perfectly adequate for many applications, often it would be preferable to achieve interactivity while preserving, as much as possible, the quality of a more expensive renderer. For example, it is desirable to use the same renderer when editing a scene as will be used for the final images or animation.

The goal of this work is to show how high quality ray-based rendering algorithms can be combined with the the high framerates needed for interactivity, using only a level of computational power that is widely and cheaply available today. Once achieved, this creates a compelling visual interface that users quickly find addictive and are reluctant to relinquish. In the typical visual feedback loop, as illustrated in Figure 1(a), the expense of the renderer often strictly limits the achievable framerate [2].

The *render cache* is a new technique to overcome this limitation and allow interactive rendering in many cases where this was previously infeasible. The renderer is shifted out of the synchronous part of the visual feedback loop and a new display process introduced to handle image generation as illustrated in Figure 1(b). This greatly

---

[1] *i*MAGIS is joint project of CNRS, INPG, INRIA and Université Joseph Fourier.

[2] In this paper we will use framerate to mean the rate at which the renderer or display process can produce updated images. This is usually slower than the framerate of the display device (e.g., a monitor or CRT).
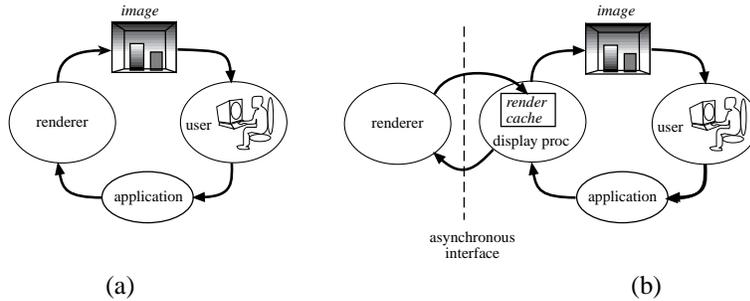
Fig. 1. (a) The traditional interactive visual feedback loop where the framerate is limited by the speed of the renderer. (b) The modified loop using the *render-cache* which decouples display framerate from the speed of the renderer.

reduces the framerate's dependence on the speed of the renderer. The display process, however, does not replace the renderer and depends on it for all shading computations.

The display process caches recent results from the renderer as shaded 3D points, reprojects these to quickly estimate the current image, and directs the renderer's future sampling. Reprojection alone would result in numerous visual artifacts, however, many of these can be handled by some simple filters. For the rest we introduce several strategies to detect and prioritize regions with remaining artifacts. New rendering samples are concentrated accordingly to rapidly improve subsequent images. Sampling patterns are generated using an error diffusion dither to ensure good spatial distributions and to mediate between our different sampling strategies.

The render cache is designed to make few assumptions about the underlying rendering algorithm so that it can be used with different renderers. We have demonstrated it with both ray tracing [29] and path tracing [14] rendering algorithms and shown that it allows them to be used interactively using far less computational power than was previously required. Even when the renderer is only able to produce a low number of new samples or pixels per frame (e.g., 1/64th of image resolution), we are able to achieve satisfactory image quality and good interactivity. Several frames taken from an interactive session are shown in Figure 2.

1.1 Previous Work

The render cache utilizes many different techniques and ideas to achieve its goal of interactivity including progressive refinement for faster feedback, exploiting spatio-temporal image coherence, using reprojection to reuse previously results, image space sparse sampling heuristics, decoupling rendering and display framerates, and parallel processing. The contribution of the render cache is to show how these ideas can be adapted and used simultaneously in a context where *interactivity* is considered of paramount importance, in a way that is both novel and effective.

One way to provide faster visual feedback and enhanced interactivity is to provide the user with approximate intermediate results rather than waiting for exact results to be available. This is often known as progressive refinement and has been used by many researchers (e.g., the "golden thread" of [3] or progressive radiosity [9]).

Many researchers have explored ways to exploit spatio-temporal image plane coherence to reduce the computational costs in ray tracing sequences of images. With a fixed
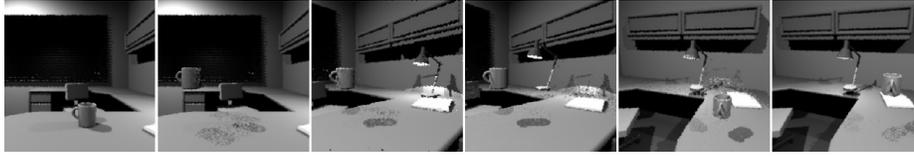
Fig. 2. Some frames from an interactive editing session using the render cache. The user is given immediate feedback when changing the viewpoint or moving objects such as the mug and desk lamp in this ray traced scene. While there are some visual artifacts, object positions are rapidly updated while other features such as shadows update a little slower. The user can continue to edit and work without waiting for complete updates. On a single processor, this session ran at ∼5fps and lasted about one minute. No graphics hardware acceleration was used. See Appendix for larger color images.

camera, changing materials or moving objects can be accelerated by storing partial or complete ray trees (e.g., [25, 13, 6]).

Sequences with camera motions can be handled by storing the rendered points and reprojecting onto the new camera plane. There are several inherent problems with reprojection including that the mapping is not a bijection (some pixels have many points map to them and some have none), occlusion errors (if the reprojected point is actually behind an occluding surface in the new view), and non-diffuse shading (a point's color may change when viewed from a different angle). Many different strategies for mitigating these problems have been proposed in the image-based literature (e.g., [8, 18, 17, 16, 26]), which relies heavily on reprojection.

Reprojection has also been used in ray tracing to accelerate the generation of animation sequences (e.g., [2, 1]). These methods save considerable computation by reprojecting data from the previous frame and only recomputing pixels which are potentially still incorrect. At a high level their operations are similar to those of the render cache but their goal (i.e. computing exact frames) is different. Our goal of interactivity requires the use of fast reconstruction heuristics that work reasonably well even in the presence of inexact previous frames and a prioritized sparse sampling scheme to best choose the limited number of pixels that can be recomputed per frame.

Sparse or adaptive image space sampling strategies (e.g., [21, 19, 11, 5]) can greatly reduce ray tracing costs. While most work has concentrated on generating single images, some researchers have also considered animations (e.g., using a uniform random sampling to detect changed regions [2] or uniform deterministic sampling in an interactive context [4]). The render cache introduces a new sampling strategy that combines several different pixel update priority schemes and uses an error diffusion dither [10] to mediate between our conflicting goals of a uniform distribution for smooth image refinement and concentrating samples in important regions for faster image convergence.

Parallel processing is another way to accelerate ray tracing and global illumination rendering (e.g., see [24] for one survey). Massive parallel processing can be used to achieve interactive ray tracing (e.g., [20, 22]), but this is an expensive approach. A better alternative is to combine parallel processing with intelligent display algorithms. For example, Parker et. al. [22] who used frameless rendering [4] to increase their framerate, could benefit from the render cache which produces better images and requires significantly fewer rays per frame.

The Post-Rendering 3D Warp [16] is an alternative intelligent display process. It displays images at a higher framerate than that of the underlying renderer by using
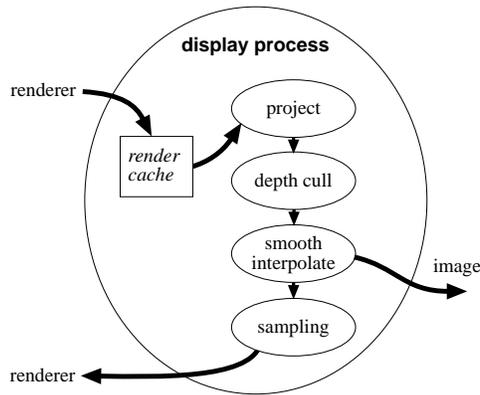
3

Fig. 3. The display process: The render cache receives and caches sample points from the renderer, which are then projected onto the current image plane. The results are filtered by the depth culling and interpolation steps to produce the displayed image. A priority image is also generated which is used to choose which new samples will be requested from the renderer.

image warping to interpolate from neighboring (past and future) rendered frames. One drawback is that the system must predict far enough into the future to render frames before they are needed for interpolation. This is trivial for a pre-scripted animation, but extremely difficult in an interactive context.

Another example is the "holodeck" [15] which was designed as a more interactive front end for Radiance [28]. It combines precomputation, reprojection, and online uniform sampling to greatly increase interactivity as compared to the Radiance system alone. Unlike the render cache though, it is not designed to handle dynamic scenes or long continuous camera motions and uses a less sophisticated sampling strategy.

2 Algorithm Overview

Our display process is designed to be compatible with many different renderers. The main requirement is that the renderer must be able to efficiently compute individual rays or pixels. Thus, for example, ray tracing like renderers are excellent candidates while scan conversion renderers are not. The display process provides interactive feedback to the user even when the renderer itself is too slow or expensive to produce complete images at an interactive rate (though the number of visual artifacts will increase if the renderer would take more than a few seconds to produce a full image).

There are several essential requirements for our display process. It must rapidly generate approximations to the current correct image based on the current viewpoint and the data in the render cache. It must control which rays or samples are rendered next to rapidly improve future images. It also must manage the render cache by integrating newly rendered results, and discarding old data when appropriate or necessary.

Image generation consists of projection, depth culling, and interpolation/smoothing steps. Rendered points from the cache are first projected onto the current view plane. We try to enforce correct occlusion both within a pixel by using a z-buffered projection, and among neighboring pixels using the depth culling step. The interpolation step fills in small gaps in the often sparse point data and produces the displayed image.
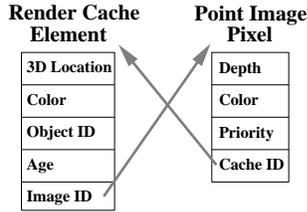
4

Fig. 4. The fields in the render cache and point image. Each point or element in the render cache contains a 3D location, a color, and an object id all provided by the renderer. They also have an age which is incremented each frame and an image id field which tells which pixel (if any) this point currently maps to. Each pixel in the point image contains a depth, a color, a priority, and the cache id of the point (if any) that is currently mapped to this pixel.
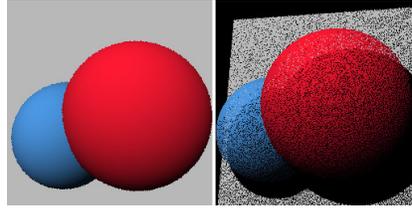
Fig. 5. Results of z-buffered point projection for a simple scene containing a white plane behind two diffuse spheres. The points generated for one viewpoint (left) are projected on the image plane for a new viewpoint (right). Notice that there are many gaps where no point projected onto a particular image (shown as black) and some points of the lighter plane are showing through gaps in the darker sphere points which should be occluding them.

Simultaneously, the display process also builds a *priority image* to guide future sampling. Because we expect that only a small subset of pixels can be rendered per frame, it important to direct the rendered sampling to maximize their benefit. Each pixel is given a priority value based on the relative value of rendering a new sample at that pixel. We then use an error diffusion dither [10] to choose the new samples to request. Using a dither both concentrates more samples in important regions and ensures that the samples are well spaced and distributed over the entire image. A diagram of the display process steps is shown in Figure 3.

3 Image Generation

Images are generated in our display process by projecting rendered points from the render cache onto an augmented image plane called *point image* (see Figure 4 for corresponding data fields). The projection step consists of a transform based on the current camera parameters as specified by the application program and z-buffering to handle the cases when more than one point maps to the same pixel. Whenever a point is mapped to a pixel, their corresponding data fields (see Figure 4) are updated appropriately including writing the point's color, depth, and a priority based on its age to the pixel.

The raw results of such a projection will contain numerous artifacts as illustrated in Figure 5. We handle some of the simpler kinds of artifacts using some filters while relying our sampling algorithm and newly rendered samples to resolve the more difficult artifacts in future frames. We have deliberately chosen to use only fast, simple techniques and heuristics in our system to keep its computational requirements both as light and consistent as possible.

3.1 Depth Culling and Smoothing/Interpolation

Some of the points, though formerly visible, may currently lie behind an occluding surface (e.g., due to object or camera motion). Visual artifacts, such as surfaces incorrectly "showing through" other surfaces, occur if such points are not removed (see Figure 6).
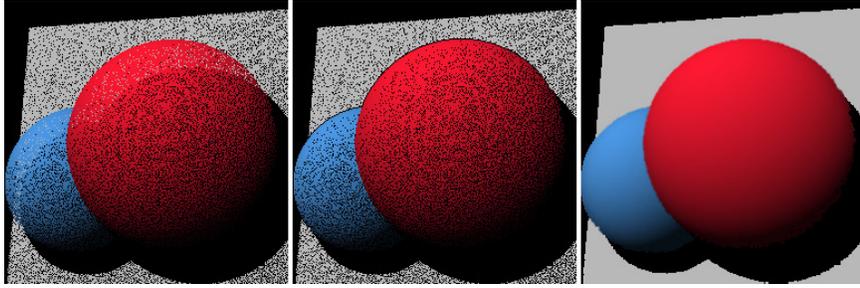
5

Fig. 6. Image reconstruction example: The raw projected points image (left) from Figure 5 is filtered by our depth-cull (middle) and interpolation (right) to produce an image that gives the correct impression that the surfaces are opaque and continuous.

Projection only removes occluded points if a point from the occluding surface maps to the same pixel. We remove more occluded points using a depth culling heuristic that searches for points whose depth is inconsistent with their neighbors.

Each pixel's 3x3 neighborhood is examined and an average depth computed, ignoring neighboring pixels without points. If the point's depth is significantly different from this average, then it is likely that we have points from different surfaces and that the nearer surface should now be occluding the farther one. Based on this assumption, we remove the point (i.e. we treat it as if no point had mapped to this pixel) if its depth is more than some threshold beyond the average depth (currently we use 10%). This heuristic both correctly removes many points which should have been occluded and falsely removes some genuinely visible points near depth discontinuity edges. Fortunately the incorrect removal artifacts are largely hidden by the interpolation step.

Next we use an interpolation and smoothing filter to fill in small gaps in the point image (see Figure 6). For each pixel, we again examine its 3x3 neighborhood and perform a weighted average[3] of the corresponding colors. The weights are 4, 2, and 1 for center, immediate neighbor, and diagonal neighbors respectively, and pixels without points receive zero weight. This average becomes the pixel's displayed color except when there are no points in the neighborhood making the average invalid. Such pixels either retain the color they had in the previous frame or are displayed as black depending on the user's preference.

The quality of the resulting image depends on how relevant the cached points are to the current view. Actions such as rapid turning or moving through a wall can temporarily degrade image quality significantly. Fortunately the sparse sampling and interpolation tend to quickly restore image quality. Typically the image quality becomes usable again by the time that just one tenth of the cache has been filled with relevant points.

4   Sampling

Choosing which samples the renderer should compute next is another essential function of the display process. Since we expect that the number of new samples computed per frame to be much smaller than the number of pixels in the displayed image (typically by a factor between 8 and 128), it is important to optimize the placement of these sparse

---

[3]As described the system performs some slight smoothing even in fully populated regions. If this is considered objectionable, smoothing could easily be disabled at those pixels that had a point map to them.
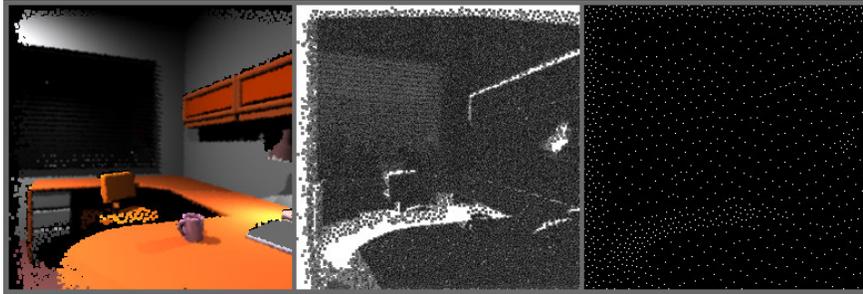
Fig. 7. A image produced by the display process (left) along with its corresponding priority image (middle) and the dithered binary image specifying which sample locations will be requested next from the renderer. In this case the user is moving toward the upper left and the high priority regions are due to previously occluded regions becoming visible. Note that the dithering algorithm causes new samples to be concentrated in these high priority regions while staying well spaced and distributed over the entire image region.

samples. Samples are chosen by first constructing a grayscale sampling *priority image* and then applying an error diffusion dither algorithm. We use several heuristics to give high priority to pixels that we suspect are likely to contain visual artifacts.

The priority image is generated simultaneously with image reconstruction. Each point in the render cache has an age which starts at zero and is incremented each frame. When a point in the render cache maps to a pixel, that pixel's priority is set based on the point's age. This reflects the intuition that it is more valuable to recompute older samples since they are more likely to have changed. The priority for other pixels is set during the interpolation step based on how many of their neighbors had points map to them. Pixels with no valid neighbors receive the maximum possible priority while pixels with many valid neighbors receive only a medium priority. The intuition here is that it is more important to sample regions with lower local point densities first.

Choosing sampling locations from the priority image is equivalent to turning a grayscale image into a binary image. We want our samples to have a good spatial distribution so that the image visually refines in a smooth manner by avoiding the clumping of samples and ensuring that they are distributed over the whole image. We also want to concentrate more samples in high priority regions so that the image converges more quickly. A uniform distribution would not properly prioritize pixels, and a priority queue would not ensure a good spatial distribution. Instead we utilize a simple error diffusion dithering algorithm [10] to create the binary sample image (see Figure 7). The dithering approach nicely mediates between our competing sampling goals at the cost of occasionally requesting a low priority pixel.

In our implementation, scanlines are scanned in alternating directions and the priority at each pixel compared to a threshold value (total priority / # samples to request). If above threshold, this pixel is requested as a sample and the threshold subtracted from its priority. Any remaining priority is then propagated, half to the next pixel and half to the corresponding pixel on the next scanline.

4.1 Premature Aging

By default, points age at a constant rate, but it is often useful to prematurely age points that are especially likely to be outdated or obsolete. Premature aging encourages the

system to more quickly recompute or discard these points for better performance.

A good example is our color change heuristic. Often a new sample is requested for a pixel which already contains a point. We consider this a *resample*, record the old point's index in the sample request, and ensure that the requested ray passes exactly through the 3D location[4] of the old point. We can then compare the old and new colors of resampled pixels to detect changes (e.g., due to changes in occlusion or lighting). If there is a significant change, then it is likely that nearby pixels have also changed. Therefore, we prematurely age any points that map to nearby pixels. In this way we are able to automatically detect regions of change in the image and concentrate new samples there. Another example is that we prematurely age points which are not visible in the current frame since it is likely that they are no longer useful.

4.2   Renderer and Application Supplied Hints

While we want our display process to work automatically, we also want to provide ways for the renderer and application to optionally provide hints to increase the display process' effectiveness. For example, the renderer can flag some points as being more likely to change than others and thus should be resampled sooner. The display process then ages these points at a faster rate. Some possible candidates are points on a moving objects, points in their shadows, or points which are part of a specular highlight. Together with the resample color change optimization, this can greatly improve the display process's ability to track the changes in such features.

The noise inherent in Monte Carlo renderers can cause the display process to falsely think that a sample's color has changed significantly during resampling. Falsely triggering the color change heuristic can prematurely age still valid points and wastefully concentrate samples in this region. To avoid this, the renderer can provide a hint that specifies the expected amount of noise in a result. This helps the display process to distinguish between significant color changes and variation simply due to noise.

We have also added a further optimization to help with moving objects. The application can provide rigid body transforms (e.g., rotation or translation) for objects. The display process then updates the 3D positions of points in the render cache with the specified object identifiers. This significantly improves the tracking of moving objects as compared to resampling alone though resampling is still necessary.

4.3   Cache Management Strategy

We use a fixed size render cache that is slightly larger than the number of pixels to be displayed. Thus each new point or sample must overwrite a previous one in the cache. The fixed size cache helps keep the computational cost low and constant. In dynamic environments, this also ensures that any stale data will eventually be discarded.

New points or samples that are resamples of an old point (see above) simply overwrite that point in the cache. Since the old point is highly likely to be either redundant or outdated, this simple strategy works well.

For other new points, we would like to find some no longer useful point to overwrite. One strategy is to replace the oldest point in the cache as it is more likely to be obsolete. However, we decided that doing this exactly would be unnecessarily expensive. Instead we examine a subset of the cache (e.g., groups of 8 points in a round robin order) and replace the oldest point found.

---

[4]Otherwise requested rays are generated randomly within the pixel to reduce aliasing and Moie´ patterns.

5 Implementation and Results

We have implemented our display process and a simple test application that allows us to change viewpoints and move objects. The display process communicates with renderers using a simple abstract broker interface. This abstract interface allows us to both easily work with different renderers (two ray tracers and two path tracers so far) and to utilize parallel processing by running multiple instances of the renderers simultaneously. The broker collects the sample requests from the display process, distributes them to the renderers when they need more work and gathers rendered results to be returned to the display process. It is currently written for shared memory parallel processing using threads, though a message passing version for distributed parallel processing is also feasible.

The render mismatch ratio is a useful measure of the effectiveness of the render cache and our display process. We define this ratio as the number of pixels in a frame divided by the number of new samples or pixels produced by the renderer per frame. It is render cache's ability to handle higher mismatch ratios that allows us to achieve interactivity while using more expensive renderers and/or less computational power.

Working with a mismatch ratio of one is trivial; render and display each frame. Mismatch ratios of two to four can easily be handled using existing techniques such as frameless rendering [4]. The real advantage and contribution of the render cache is its ability to effectively handle higher mismatch ratios. In our experience, the render cache works well for mismatch ratios up to 64 and can be usable at even higher ratios. In many cases this allows us to achieve much greater interactivity with virtually no modification to the renderer. Performance in particular cases will of course depend on many factors including the absolute framerate, scene, renderer, and user task.

5.1 Results

Our current implementation runs on Silicon Graphics workstations using software only (i.e. we do not use any 3D graphics hardware). Our experience shows that the render cache can achieve interactive ray tracing even on single processor systems whose processing power to equivalent to that of today's PC computers. Specialized or expensive hardware is not required, though we can also exploit the additional rendering power of parallel processing when available.

Timings for the display process running on a 195Mhz R10000 processor in an SGI Origin 2000 are shown in Table 1. The display process can generate a 256x256 frame in 0.07 seconds for a potential framerate of around 14 frames per second. In a uniprocessor system, the actual framerate will be lower because part of the processors time must also be devoted to the renderer. In this case, we typically split the processors time evenly between the display process and the renderer for a framerate of around 7 fps. Even on a multiple processor machine it may be desirable to devote less than a full processor to the display process in order to increase the number of rendered samples produced.

Using larger images is trivial though it reduces the framerate. The time to produce each frame scales roughly linearly with the number of pixels to be displayed since all the data structures sizes and major operations are linear in the number of pixels.

We have tested the render cache in various interactive sessions using both ray tracing [29] and path tracing [14] renderers and on machines ranging from one to sixty processors. Some images from example sessions are shown in Figures 2 and 8 (see color plates in Appendix) and videos are available on our web page [5].

---

[5]http://www-imagis.imag.fr/Publications/walter

Initialize buffers	0.0046 secs
Point projection	0.0328 secs
Depth cull	0.0085 secs
Interpolation	0.0139 secs
Display image	0.0027 secs
Request new samples	0.0053 secs
Update render cache	0.0027 secs
Total time	0.0705 secs

Table 1. Timings for the display process' generation of a 256x256 image produced on a single 195Mhz R10000 processor. The display process is capable of producing about 14 frames per second in this case, though the actual framerate may be slower if part of the processors time is also devoted to renderering.

In all cases tested, the render cache provides a much more interactive experience than any other method using the same renderers that we are aware of (e.g., [22, 4]). The reprojection correctly tracks motion and efficiently reuses relevant previously rendered samples. While there are visual artifacts in individual frames, the prioritized sparse sampling smoothly refines the images and allows us to quickly recover from actions that make the previous samples irrelevant (e.g., walking through a wall). We still rely on the renderer for all shading calculations and need it to produce an adequate number of new samples per frame. Compared to previous methods though, we require far fewer new samples per frame to maintain good image quality.

All the sessions shown in Figure 8 used 320x320 resolution and ran at around 8 fps. The first three sessions used ray tracing and between two and four R10000 processors. An image from a sequence where the user walks through a door in Greg Larson's cabin model is shown in the upper left. In the upper right, an ice cream glass has just been moved in his soda shoppe model, and its shadows are in the process of being updated. In the lower left, the camera is turning to the right in a scene with many ray traced effects including extensive reflection and refraction.

The lower right of Figure 8 shows a path tracing of Kajiya's original scene. Path tracing simulates full global illumination and is much more expensive. The four processor version (shown) is no longer really adequate as too few new samples are rendered per frame resulting in more visual artifacts. Nevertheless, interactivity is still much better than it would be without the render cache. We have demonstrated good interactivity even in this case when using a sixty processor machine.

6   Conclusions

The render cache's modular nature and generic interfaces allows it to be used with a variety of different renderers. It uses simple and fast algorithms to guarantee a fast consistent framerate and is designed for interactivity even when rendered samples are expensive and scarce. Reprojection and filtering intelligently reuse previous results to generate new images and a new directed sampling scheme tries to maximize the benefits of future rendered results.

Our prototype implementation has shown that we can achieve interactive framerates using software only for low but reasonable resolutions. We have also shown that it can enable satisfactory image quality and interactivity even when the renderer is only able to produce a small fraction of new pixels per frame (e.g., between 1/8 and 1/64 of

the pixels in a frame). We have also demonstrated it working with both ray tracing and path tracing and efficiently using parallel processors ranging from two to sixty processors. Moreover, we have shown the render cache can handle dynamic scenes including moving objects and lights.

We believe that the render cache has the potential to significantly expand the use of ray tracing and related renderers in interactive applications and provide interactive users with a much wider selection of renderers and lighting models to choose from.

6.1 Future Work

There are many ways in which the render cache can be further improved. Higher framerates and bigger images are clearly desirable and will require more processing power. With its fixed-size regular data structures and operations, the render cache could benefit from the small-scale SIMD instructions that are becoming common (e.g., AltiVec for PowerPC and SSE for Pentium III). It is also a good target for graphics hardware acceleration as its basic operations are very similar to those already performed by current graphics hardware (e.g., 3D point projection, z-buffering, and image filtering).

The lack of good anti-aliasing is one clear drawback of the render cache as presented here. Unfortunately since anti-aliasing is highly view dependent, we probably do not want to include anti-aliasing or area sampling within individual elements in the render cache [8]. This leaves supersampling as the most obvious solution though this will considerably increase the computational expense of the display process.

Although the render cache works well for renderer mismatch ratios up to 64, more work is needed to improve its performance at higher ratios. Some of the things that will be needed are interpolation over larger spatial scales, better very sparse sampling, and methods to prematurely evict obsolete points from the render cache.

Because the render cache works largely in the image plane, it is an excellent place to introduce perceptually based optimizations and improvements. Some examples include introducing dynamic tone mapping models (e.g., [27, 23]) or using perceptual based sampling strategy (e.g., [5]).

We also like to see our display process used with a wider variety of renderers such as Radiance [28], bidirectional path tracing, photon maps[12], and the ray-based gather passes of multipass radiosity methods [7].

References

1. S. J. Adelson and L. F. Hodges. Generating exact ray-traced animation frames by reprojection. *IEEE Computer Graphics and Applications*, 15(3):43–52, May 1995.
2. S. Badt. Two algorithms taking advantage of temporal coherence in ray tracing. *The Visual Computer*, 4(3):123–132, Sept. 1988.
3. L. D. Bergman, H. Fuchs, E. Grant, and S. Spach. Image rendering by adaptive refinement. In *Computer Graphics (SIGGRAPH '86 Proceedings)*, volume 20, pages 29–37, Aug. 1986.
4. G. Bishop, H. Fuchs, L. McMillan, and E. J. Scher Zagier. Frameless rendering: Double buffering considered harmful. In *Computer Graphics (SIGGRAPH '94 Proceedings)*, pages 175–176, July 1994.

5. M. R. Bolin and G. W. Meyer. A perceptually based adaptive sampling algorithm. In M. Cohen, editor, *SIGGRAPH 98 Conference Proceedings*, pages 299–310, July 1998.

6. N. Brière and P. Poulin. Hierarchical view-dependent structures for interactive scene manipulation. In *SIGGRAPH 96 Conference Proceedings*, pages 83–90, Aug. 1996.

7. S. E. Chen, H. Rushmeier, G. Miller, and D. Turner. A progressive multi-pass method for global illumination. In *SIGGRAPH 91 Conference Proceedings*, pages 165–174, July 1991.

8. S. E. Chen and L. Williams. View interpolation for image synthesis. In J. T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 279–288, Aug. 1993.

9. M. F. Cohen, S. E. Chen, J. R. Wallace, and D. P. Greenberg. A progressive refinement approach to fast radiosity image generation. *Computer Graphics*, 22(4):75–84, August 1988. ACM Siggraph '88 Conference Proceedings.

10. R. W. Floyd and L. Steinberg. An adaptive algorithm for spatial greyscale. In *Proceedings of the Society for Information Display*, volume 17(2), pages 75–77, 1976.

11. B. Guo. Progressive radiance evaluation using directional coherence maps. In M. Cohen, editor, *SIGGRAPH 98 Conference Proceedings*, pages 255–266, July 1998.

12. H. W. Jensen. Global illumination using photon maps. In *Rendering Techniques '96*, pages 21–30. Springer-Verlag/Wien, 1996.

13. D. A. Jevans. Object space temporal coherence for ray tracing. In *Proceedings of Graphics Interface '92*, pages 176–183, May 1992.

14. J. T. Kajiya. The rendering equation. In D. C. Evans and R. J. Athay, editors, *Computer Graphics (SIGGRAPH '86 Proceedings)*, volume 20, pages 143–150, Aug. 1986.

15. G. W. Larson. The holodeck: A parallel ray-caching rendering system. In *Second Eurographics Workshop on Parallel Graphics and Visualisation*, Rennes, France, Sept. 1998.

16. W. R. Mark, L. McMillan, and G. Bishop. Post-rendering 3D warping. In *1997 Symposium on Interactive 3D Graphics*, pages 7–16. ACM SIGGRAPH, Apr. 1997.

17. N. Max and K. Ohsaki. Rendering trees from precomputed Z-buffer views. In *Eurographics Rendering Workshop 1995*. Eurographics, June 1995.

18. L. McMillan and G. Bishop. Plenoptic modeling: An image-based rendering system. In R. Cook, editor, *SIGGRAPH 95 Conference Proceedings*, pages 39–46, Aug. 1995.

19. D. P. Mitchell. Generating antialiased images at low sampling densities. In M. C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, pages 65–72, July 1987.

20. M. J. Muuss. Towards real-time ray-tracing of combinatorial solid geometric models. In *Proceedings of BRL-CAD Symposium*, 1995. http://ftp.arl.mil/ mike/papers/.

21. J. Painter and K. Sloan. Antialiased ray tracing by adaptive progressive refinement. In *Computer Graphics (SIGGRAPH '89 Proceedings)*, pages 281–288, July 1989.

22. S. Parker, W. Martin, P. Sloan, P. Shirley, B. Smits, and C. Hansen. Interactive ray tracing. In *Symposium on Interactive 3D Computer Graphics*, April 1999.

23. S. N. Pattanaik, J. A. Ferwerda, M. D. Fairchild, and D. P. Greenberg. A multiscale model of adaptation and spatial vision for realistic image display. In *Computer Graphics*, July 1998. ACM Siggraph '98 Conference Proceedings.

24. E. Reinhard, A. Chalmers, and F. W. Jansen. Overview of parallel photo-realistic graphics. In *Eurographics '98 State of the Art Reports*. Eurographics Association, Aug. 1998.

25. C. H. Séquin and E. K. Smyrl. Parameterized ray tracing. In J. Lane, editor, *Computer Graphics (SIGGRAPH '89 Proceedings)*, volume 23, pages 307–314, July 1989.

26. J. W. Shade, S. J. Gortler, L. He, and R. Szeliski. Layered depth images. In M. Cohen, editor, *SIGGRAPH 98 Conference Proceedings*, pages 231–242, July 1998.

27. G. Ward. A contrast-based scalefactor for luminance display. In P. Heckbert, editor, *Graphics Gems IV*, pages 415–421. Academic Press, Boston, 1994.

28. G. J. Ward. The RADIANCE lighting simulation and rendering system. *Computer Graphics*, 28(2):459–472, July 1994. ACM Siggraph '94 Conference Proceedings.

29. T. Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, June 1980.
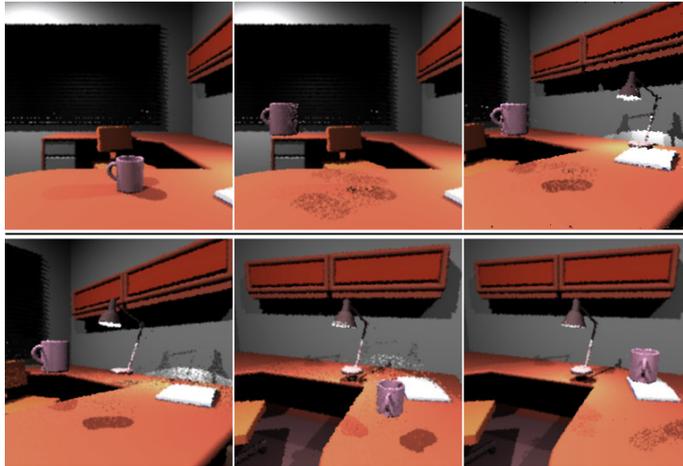
Fig. 2. Some frames from a render cache session. See main text for more detail.

Fig. 8. Some example images captured from interactive sessions. Some approximation artifacts are visible but the overall image quality is good. All scenes are ray traced except the lower right which is path traced. In the upper right image we have just moved the ice cream glass and you can see its shadow in the process of being updated on the table top.

13