The Real-Time Reprojection Cache

Diego Nehab[1] Pedro V. Sander[2] John R. Isidoro[2]
[1]Princeton University [2]ATI Research Inc.

Abstract

Real-time pixel shading techniques have become increasingly complex, and consume an ever larger share of the graphics processing budget in applications such as games. This has driven the development of optimization techniques that either attempt to simplify pixel shaders, or to cull their evaluation when possible. In this paper, we follow an alternative strategy: reducing the number of shading computations by exploiting spatio-temporal coherence.

We describe a simple and inexpensive method that uses the graphics hardware to cache and track surface information through time. The Real-Time Reprojection Cache stores surface information in screen space, thereby avoiding complex data-structures and bus traffic. When a new frame is rendered, reverse mapping by reprojection gives each new pixel access to information computed during the previous frame.

Using this idea, we show how to modify a variety of real-time rendering techniques to efficiently exploit spatio-temporal coherence. We present examples that vary as widely as stereoscopic rendering, motion blur, depth of field, shadow mapping, and environment-mapped bump mapping. Since the overhead of a reprojection cache lookup is small in comparison to the required per-pixel processing, the cached algorithms show significant cost and/or quality improvements over their plain counterparts, at virtually no extra implementation overhead.

1 Introduction

Over the past few years, a clear tendency in real-time rendering applications has been the steady increase in pixel shading complexity. As GPUs gain in power and flexibility, sophisticated per-pixel rendering effects are becoming prevalent. Researchers are therefore starting to investigate general techniques for the optimization of pixel shading, such as automatic shader simplification [Olano et al. 2003; Pellacini 2005]. In this work, we introduce the Real-Time Reprojection Cache (RRC), a method applicable in the optimization of a wide range of pixel shading techniques.

Most real-time rendering applications exhibit a considerable amount of spatio-temporal coherence (see figure 1). High frame rates lead to small time steps, which in turn result in little change between consecutive frames. Camera motions, object animations, and lighting variations are all modest. Accordingly, projected visible surface areas and their properties are nearly unchanged. This coherence can be exploited if, in the process of computing a new frame, we can efficiently access values computed in the previous frame.

The underlying concept in the RRC is that of reverse mapping by reprojection (section 3). We use frame-buffers to cache surface information, thereby avoiding complex data structures and bus traffic between the CPU and GPU. As each pixel is generated in the new frame, we know the surface point from which it originated. We also know where this surface point was, in 3D space, at the time the previous frame was rendered. Therefore, we can easily find where

Figure 1: *Real-time rendering applications exhibit a considerable amount of spatio-temporal coherence. This is true for camera motions (top) as well as for animated object (bottom). The snapshots of the Parthenon, Heroine, and Ninja sequences illustrate this fact. Newly visible surface points are rendered in red, whereas the vast majority (shown in green) were previously visible. This paper introduces a real-time method that exploits this coherence by caching and tracking visible surface information.*

it previously projected to, and whether it was visible at that time. We can then fetch whatever surface information we stored in the previous frame's buffers, and use it while rendering the new frame.

In the real-time world, the raw cost of reprojecting a pixel has traditionally been comparable to—or higher than—that of shading the pixel anew. However, the recent popularity of sophisticated pixel shading techniques has made reprojection a comparatively cheap operation. This opens the door for a series of reprojection-based optimizations that would otherwise be disadvantageous. The RRC is a tool that greatly simplifies the implementation of such optimizations.

Consider, for example, the possibility of caching shaded surface colors. We can usually reuse cached values directly while rendering a new frame, computing new colors only for cache misses. This can lead to significantly higher frame rates at the same visual quality. Alternatively, we can compute a full new frame, but merge older samples into it. The results are higher quality, super-sampled frames, whose costs have been amortized across two or more frames.

Naturally, we are not limited to caching color information. We can also cache the results of expensive operations, such as multiple texture fetches, procedural texture computations, shadow map tests etc. (section 4). Caching allows us to decouple the application refresh rate from the rate at which certain computations are performed. We can cache partial results, to be completed during the rendering of future frames. Alternatively, we can cache full results over alternating subsets of all pixels. In fact, we can combine these two ideas in a variety of ways.

2 Related Work

Although the cost of reprojection has only recently become cheap relative to standard real-time pixel shading techniques, reprojection-based optimizations have been used extensively in

other scenarios. For example, high-quality rendering techniques such as ray-tracing or path-tracing have always been considerably more expensive than reprojection. Additionally, given high enough scene complexity, image based rendering techniques can run substantially faster than rasterization, even in a low-quality setting. Finally, especially designed hardware can make reprojection advantageous by reducing its cost.

Expensive renderers: Badt [1988] introduced reprojection as a technique to exploit temporal coherence in the off-line generation of ray-traced animation sequences. Samples from the previous frame are forward-mapped into the new frame, by reprojection, to account for camera motion. Besides handling object motion, the technique presented by Adelson and Hodges [1995] also guarantees exact results. Further attempts to bring interactivity to ray-tracing, such as the Radiance Interpolants [Bala et al. 1999] and the Render Cache [Walter et al. 1999], resulted in very similar ideas.

One disadvantage of forward reprojection is that it leads to reconstruction challenges. Reprojection does not, in general, yield a one-to-one correspondence between pixels in the two projection planes. Holes and overlaps must be efficiently detected and dealt with. Suggested solutions include carefully choosing the order in which pixels are reprojected [McMillan and Bishop 1995], preserving previous pixel colors [Bishop et al. 1994], filtering the holes out [Walter et al. 1999], or fully recomputing the values within gaps [Bala et al. 1999].

A different approach is presented by the Holodeck and Tapestry systems [Ward and Simmons 1999; Simmons and Séquin 2000]. These store samples on the vertices of a dynamically tessellated triangle mesh that is placed in front of the camera. The mesh is rendered using the graphics hardware, which automatically performs the required interpolation. The Shading Cache [Tole et al. 2002] goes one step further and stores samples in object space, on the vertices of an adaptively refined representation of the scene. Rendering from a full geometric representation produces better results, especially on dynamic scenes.

Reverse reprojection seems to be the natural alternative, just as reverse mapping is the preferred choice for texture mapping. This is the approach we take. However, reverse reprojection requires depth information for the new frame, and enough computing power to reproject every pixel. Unlike our method, those that rely on using the CPU to guide an independent renderer rarely meet these conditions. Even in recent work, which has focused on using the GPU for acceleration [Dayal et al. 2005; Zhu et al. 2005], forward reprojection is still prevalent.

Image-based rendering: Most relevant to our work are 3D warping techniques which operate on a set of views and depth maps, such as those presented by Chen and Williams [1993], McMillan and Bishop [1995], and Mark et al. [1997], especially the latter. These are mainly used to generate novel views from a set of precomputed or captured images, with cost that is independent of scene complexity. Our technique, on the other hand, was designed to support animated rendering applications, such as games.

Dedicated hardware: At least two hardware architectures have been proposed that employ reprojection to speed up real-time rendering. The Address Recalculation Pipeline [Regan and Pose 1994] and the Talisman Architecture [Torborg and Kajiya 1996] achieve high frame rates by warping and compositing layered representations of a scene. In contrast, the general programmability of modern graphics processors allows us to design a caching scheme that can be easily used by other programs running on the same stock hardware.

3 The Real-Time Reprojection Cache

While rendering a given frame, an RRC application commonly accesses the cache prepared by a previous frame, and updates the cache for the frames to follow. In designing our method, our goal was to make it as flexible as possible, ensuring that these tasks can be performed in a simple, effective, and efficient way.

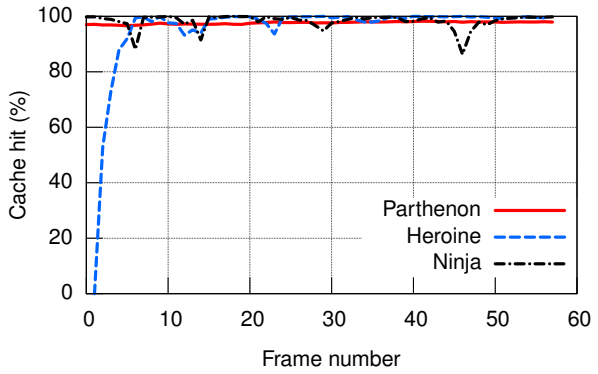


Figure 2: *The graph shows the percentage of surface area that was visible within consecutive frames for the animation sequences of figure 1. Spatio-temporal coherence causes rates to be generally above 90%. This justifies our policy of keeping cache entries for visible surface areas.*

Our description of the RRC starts with standard caching components: the eviction policy (section 3.1), data structures (section 3.2), and the lookup mechanism (section 3.3). We also discuss sampling issues that are specific to our domain (sections 3.4 and 3.5), as well as control flow strategies (section 3.6).

3.1 Eviction policy

Because real-time rendering applications exhibit considerable spatio-temporal coherence, the relevance of a cached surface entry is strongly tied to its visibility. After all, visible points are likely to remain visible, and the converse is also true. This variant of the principle of locality supports the policy of keeping cache entries for visible points. The policy is also extremely convenient: the cache can have a fixed size, i.e., one entry per pixel, and can be directly addressed by pixel coordinates.

Although the cache hit rates certainly depend on the amount of coherence in each application, our experiments show that rates in excess of 90% are typical. Figure 2 shows the observed cache hit rates for three animation sequences. The Parthenon (figure 1, top) shows a fly-through over a model of the Parthenon, with static geometry but high depth complexity. The Heroine sequence (figure 1, bottom left), shows an animated charachter with weighted skinned vertices as she runs past the camera. Finally, the Ninja sequence (figure 1, bottom right) shows an animated fighter performing typical martial arts movements. These real-world examples provide strong evidence that the eviction policy is appropriate.

3.2 Data structures

Given our eviction policy, it is natural to store cache entries in GPU-memory frame-buffers. To update the cache, the application simply renders the payload information into one or more *payload buffers*. Rendering is performed with the geometry and camera parameters current at that time (the *cache-time state*). Z-buffering automatically enforces the visibility eviction policy. In addition to the payload data, the only required information is the depth of each cached entry. This is usually available for free.

Besides the simplicity, cache operations are very efficient. In general, both the screen and the cache can be updated in a single pass on GPUs that support multiple render targets (most cards in the market do). In practice, it is often possible to exploit the alpha channel to store all required information in a single render target. Memory consumption is therefore modest, and independent of scene complexity. Furthermore, since everything remains in GPU

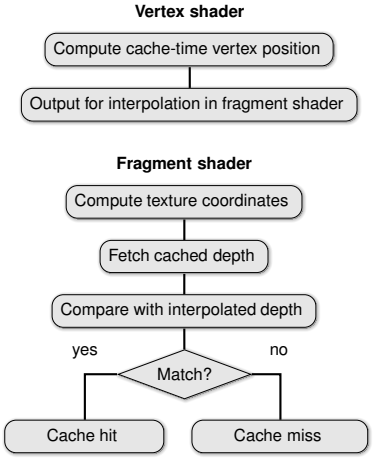


Figure 3: *Cache lookup. The vertex shader calculates the cache-time position of each vertex. The fragment shader uses the interpolated position to test the visibility of the current point in the cache frame-buffer.*

memory, there is no bus traffic between the GPU and the CPU. Finally, cache lookups conveniently reduce to texture fetches.

3.3 Cache lookup

Conceptually, the texture coordinates for a cache lookup are computed by reverse reprojection. In practice, due to the extensive information available at rendering time, the process is much simpler. Figure 3 shows a schematic description of the process.

In general, the transformed coordinates of a vertex are calculated by a vertex program, to which the application supplies the world, camera, and projection matrices, as well required animation parameters (such as tween factors and blending matrices used for skinning). If the application passes the cache-time parameters (typically for the previous frame) in addition to the current parameters, the vertex program can output both the current and cache-time transformed coordinates for each vertex.

Automatic interpolation produces the cache-time homogeneous screen coordinates associated to each fragment. Division by w within the fragment program produces the cache-time texture coordinates. These are used to fetch the depth for the cached entry. If this depth does *not* match the interpolated cache-time depth for the pixel, we have a cache miss (much like a shadow map test). If it does match, we have a cache hit. Payload data can then be found using the same texture coordinates.

Notice that simple manipulations on the cache frame-buffers allow for a series of customizations to the lookup behavior. For instance, to prevent certain objects from being cached, we can re-render them with invalid depth. It is also trivial to propagate an age field on each entry, and use it to control the life span of cached values.

3.4 Spatial resampling

Reverse reprojection transforms the problematic scattering of cached samples into a manageable gathering process. However, since reprojected pixels do not, in general, fall exactly on top of cached samples, some form of resampling is necessary. Fortunately, the uniform structure of the cache and the hardware support for texture filtering greatly simplify this task. In fact, except for depth discontinuities, cache lookups can be treated exactly as texture lookups.

The best choice for texture filtering depends on the data being cached and on the use the application makes of it. Nearest neighbor filtering is appropriate when cached data varies smoothly, or when results of cache lookup are post-filtered by the application. On the other hand, considerable variation between adjacent cache samples might justify bilinear filtering, especially if lookup results are to be directly reused.

Reconstruction can potentially fail near depth discontinuities. However, since we are dealing with cache lookups, we can simply detect and reject problematic requests. Although it is possible to be perfectly conservative, most applications are less restrictive. An efficient heuristic that works well in practice is to use bilinear filtering when fetching cached depths. Near discontinuities, interpolation across significant depth variations will not match the depth value received from the vertex shader. Lookup will therefore fail automatically. Notice that the same argument applies to multisampled frame-buffers.

Depth discontinuities pose a considerable challenge to the use of trilinear or anisotropic filtering, which could accidentally integrate across spatially unrelated data. Fortunately, since there is little change between cache-time and lookup, we have no reason to expect significant distortions in the reprojection map. Consequently, the area of a current screen pixel covers a similar area in the cache, and it makes little sense to use trilinear or anisotropic filtering.

3.5 Amortized super-sampling

A common approach to eliminate aliasing artifacts from high-quality renderings is the use of stochastic sampling [Dippé and Wold 1985; Cook 1986]. Each pixel holds a weighted average of a number of samples, and estimates the value of an integral over its area. When the sampling process is unbiased, the expected value of the samples matches the value of the integral. The quality of the estimate is given by its variance, and depends on a series of factors.

Increasing the number of samples is the simplest variance reduction strategy, but usually entails a corresponding increase in computational cost. Fortunately, because the RRC tracks surface information through time, we can amortize the cost of the sampling process across several frames. For instance, we can use a moving average over the past n estimates for a given surface point. Since the estimates are independent, this effectively multiplies the variance by $1/n$. A serious disadvantage is that this process requires keeping n cache entries for each pixel.

To eliminate the storage requirement, we can use a recursive low-pass filter instead. Let C_{f-1} represent the contents of the cache at frame $f-1$, and let s_f be the value for the newly computed sample. The recursive filter updates the cache to hold $C_f = \lambda C_{f-1} + (1-\lambda)s_f$, for $\lambda \in (0,1)$. Notice that the relative contribution of a given frame to the current estimate falls-off exponentially, with time constant given by $\tau = -1/\ln\lambda$. Notice further that the recursive filter preserves the expected value of the sampling, but multiplies its variance by $(1-\lambda)/(1+\lambda) < 1$.

Figure 4 shows the effect of the parameter λ on fall-off and variance reduction. The *memory* of the system is defined as the time, in frames, until a value is scaled by 1/256 (i.e., completely lost in 8-bits of precision). The trade-off is between reducing the variance and keeping the system responsive to change. For example, choosing a value of $\lambda = 3/5$ reduces the variance to $1/4$ the original by effectively considering information on the last 10 frames (see figures 9b and 9d). Reducing the variance by $1/8$ requires setting $\lambda = 7/9$, and pushes the complete fall-off to 22 frames. In practice, convergence happens smoothly and much sooner (the memory, as defined above, is a worst-case measure), and each application can find the highest acceptable value for λ.

3.6 Control flow

In order to take advantage of the caching mechanism, the application must be able to control the execution flow towards different code paths for the cache-hit and cache-miss cases. We refer to these code paths as the *hit shader* and *miss shader*, respectively.

Many factors can influence the choice between different methods for control flow in graphics hardware [Sander et al. 2005]. Once

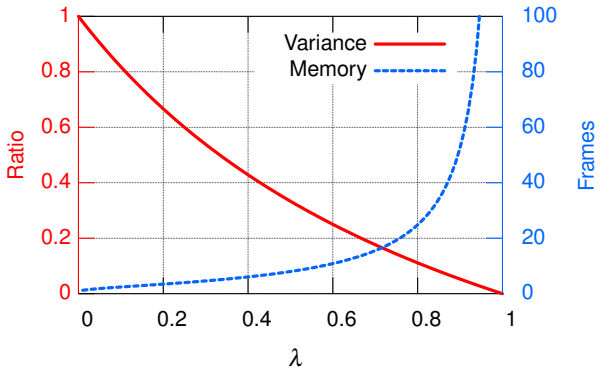


Figure 4: *When performing amortized super-sampling with a recursive filter, there is a trade-off between the amount by which the variance is reduced (the variance curve), and the number of frames that effectively contribute to the current estimate (the memory curve). This trade-off is controlled by the parameter λ. Values between 0.6 and 0.7 worked best in our tests.*

again the best option ultimately depends on the application at hand. The relative cost of the hit and miss shaders is an important factor. The complexity of the scene also plays an important role. Finally, hardware limitations might require a specific solution. We describe two options, to be used in different scenarios.

The approach described in figure 5a is adequate when either the hardware supports dynamic flow control, or when the cost for the hit and miss shaders is comparable. The first pass simply primes the Z-buffer. On the second pass, early Z-culling ensures that the fragment shader will only be executed on visible pixels. Cache lookup results are then used to branch between the hit and miss shaders. If the hardware supports dynamic flow control, the cost of execution will depend on the branch taken. The spatial coherence of lookup results ensures that lock-step execution on adjacent pixels is not an issue. Otherwise, if the cost of both branches is similar, this is irrelevant.

If the miss shader is much more expensive than the hit shader and dynamic flow control is not available, figure 5b describes an alternative: the cache lookup can be moved to the first pass. On a hit, the hit shader is executed. On a miss, the pixel is simply depth-shifted to prime the Z-buffer. On the second pass, early Z-culling ensures that the miss shader will only be executed on those pixels, and only once per pixel. Notice that, in current hardware, the depth-shift operation prevents the use of early Z-culling on the first pass. However, since we are assuming the hit shader is relatively cheap, this should not be a problem.

Other approaches are possible, for instance, using more than two passes. Depending on the application, these might be justifiable. In our tests, the options described above proved to be adequate.

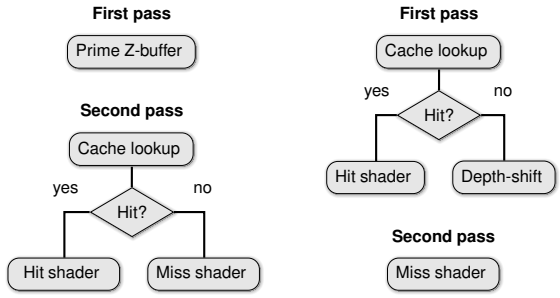
4 Applications

In the previous section, we described the RRC as a general mechanism for caching surface information across frames. In this section, we present a series of concrete examples that use the RRC to exploit spatio-temporal coherence in a variety of rendering tasks.

Perhaps the most direct application of the RRC is on stereoscopic rendering (section 4.1). By caching color values, we can easily boost the frame rates at no visual quality loss.

Effects such as motion blur (section 4.2) and depth of field (section 4.3) are also strong candidates for coherence based optimizations. Although there exist efficient approximations for these applications, the most natural method involves multiple render passes. These can be significantly optimized with the help of the RRC.

To explore the amortized super-sampling of section 3.5, we reduce aliasing in two problematic applications. In section 4.4,



(a) Dynamic flow control *(b) Explicit early Z-culling*

Figure 5: *Two control flow alternatives are presented. When the hardware supports dynamic flow control, or when the costs of the hit and miss shaders are similar, option (a) can be used. Otherwise, explicit early Z-culling is preferable (b).*

we super-sample environment-mapped bump mapping to eliminate aliasing artifacts from motion. In section 4.5, we super-sample shadow-map lookups to produce significantly higher quality shadow boundaries.

Results are presented within each application section. As usual, frame rate figures depend on a series of factors, including the system used to run the tests and the resolution being used. When comparing RRC methods to their plain counterparts, we instead focus on the trade-off between quality and performance. Similar results should apply to applications having an equivalent balance between pixel shading and geometry processing costs. In any case, all our results were produced on a P4 3.2GHz with an ATI X800 graphics card.

4.1 Stereoscopic rendering

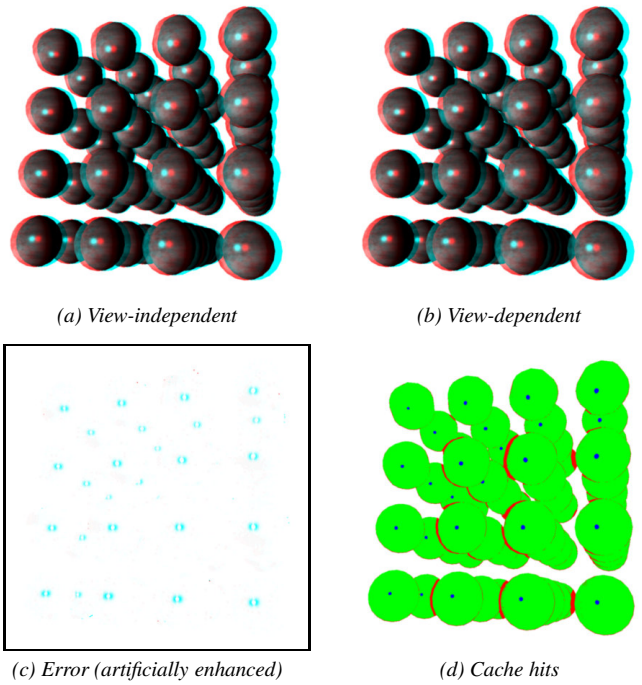
The idea of using reprojection to speed up the computation of stereoscopic images has been explored by Adelson and Hodges [1993] and by McMillan and Bishop [1995], respectively in the context of ray-tracing and head-tracked displays. Both report substantial increases in frame rate due to the extensive coherence present in nearby views.

We describe how to use the RRC to render anaglyph stereo images (see Dubois [2001] for a good review), but the same idea applies to other stereoscopic rendering techniques. On anaglyph images, the red channel is taken from the left eye view, and the green and blue channels are taken from the right eye view. Using glasses with color filters, each eye is exposed to the appropriate view, and the images appear to have three dimensions.

We proceed in two passes. On the first pass, we render the right eye view, caching the results. On the second pass, we render the scene using the left eye camera parameters, and perform one cache lookup per pixel. The hit shader simply copies the value read from the right eye. The miss shader computes the pixel color from scratch. Finally, we composite the results of the first and second passes, preserving the appropriate color channels.

If rendering each pixel is expensive, copying the values from one view to the other can lead to substantial performance improvements. Although results might not be exact on view-dependent scenes, artifacts are rarely distracting. Furthermore, if added precision is required, it is usually possible to cache only the expensive view-independent information, and add view-dependent components *after* cache lookup.

This is especially simple when the view-dependent components are additive, such as specular highlights or reflections. On the second pass, cache-time view-dependent information can be recomputed and subtracted from the cached value. The correct view-dependent information can then be added on its place. Saturated values can be easily detected and treated as cache misses.



(a) View-independent *(b) View-dependent*

(c) Error (artificially enhanced) *(d) Cache hits*

Figure 6: *When using the RRC with stereographic rendering, a view-independent treatment of cached values (a) can result in incorrect images (c). Although results are perfectly acceptable in this example, errors can be eliminated by adding view-dependent effects* after *cache lookup (b). (d) In that case, we can force cache misses over saturated specular highlights (shown in blue), in addition to the regular misses (shown in red).*

Figure 6a was generated with a view-independent treatment of the scene. As shown in figure 6c, the comparison against ground truth reveals the expected errors over the specular highlights. In figure 6b, on the other hand, the highlights were recomputed after cache lookup, completely eliminating errors. Notice the view-dependent forced cache misses in figure 6d.

The model in figure 6 has 2k triangles and uses a Perlin noise pixel shader that requires 215 instructions per pixel (expensive, but not unreasonable). Brute-force stereographic rendering happens at 28fps on our system. The view-dependent RRC method runs at 39fps, and the simpler view-independent version runs at 44fps. In other words, the RRC results in a 57% frame rate increase with negligible implementation overhead and quality loss.

4.2 Motion blur

When film is exposed for an extended interval of time, any object, camera, or shutter motion can result in a blurry image. This effect, known as motion blur, can be exploited to convey the idea of motion in static photography, or to eliminate strobing from motion pictures. The simulation of motion blur is therefore an important step in the creation of realistic synthetic images.

Satisfactory results can be obtained, for example, with temporal super-sampling [Korein and Badler 1983], stochastic sampling [Cook et al. 1984; Dippé and Wold 1985], or in the frequency domain [Potmesil and Chakravarty 1983]. In general, the high frame rate demands of real-time rendering applications restrict the range of viable approaches to coarser approximations, such as silhouette extrusion [Wloka and Zeleznik 1996]. Although graphics hardware support for accumulation buffers makes the implementation of temporal super-sampling extremely simple [Haeberli and Akeley 1990], the naïve approach tends to be overly slow. Fortunately, spatio-temporal coherence within time samples allows us to use reprojection to speed up the rendering process. This idea has been explored by Chen and Williams [1993], and by Havran et al. [2003], respectively in the context of image based rendering and ray-tracing of animations.

To use the RRC in temporal super-sampling, we proceed as follows. Recall each output image represents an interval of time, and is the result of accumulating a number of time samples within that interval. We fully render the first time sample into the cache. Then, while rendering the remaining frames for the interval, we perform one cache lookup per pixel. The miss shader computes the pixel color from scratch, whereas the hit shader simply reuses the cached value for the previous frame. If an object is known to change considerably in appearance over the exposure time (through animated textures, for instance), cache misses can be forced for that object.

Given that all time samples are averaged together, the use of reprojection causes no perceptible quality loss. On the contrary, since the rendering process becomes much faster, more time samples can be used. Figure 7 shows a comparison of the results for the brute-force and the RRC accumulation-based motion blur at the same frame rates. The model shown has 2.5k triangles and uses the same Perlin noise pixel shader used in the previous section. In this setting, the RRC enables us to double the number of time samples.

4.3 Depth of field

The standard 3D graphics pipeline is based on the pinhole camera model, and produces perfectly sharp images. Real cameras (as well as our eyes), on the other hand, have lens systems with finite apertures. Only points within a certain distance from the focal plane (the depth of field) are in focus. Points that are out of focus project to an area on the film (the circle of confusion), and result in blured images. The effect is commonly used to direct user attention, and is therefore important in high-quality renderings.

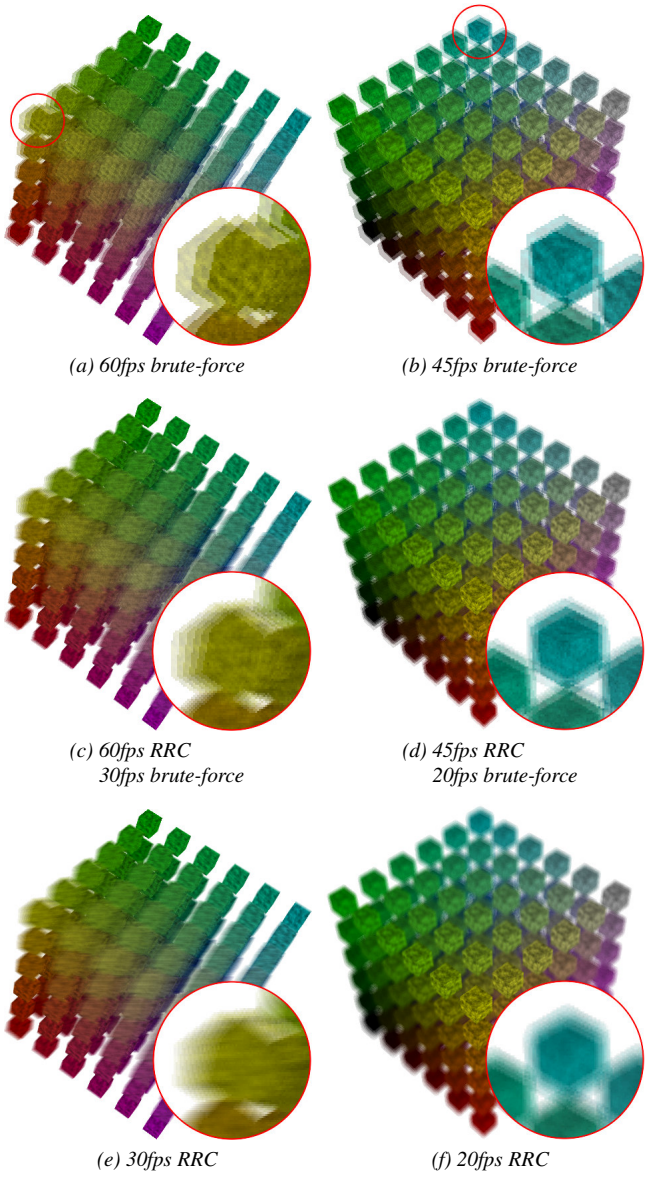
Depth of field can be simulated in a variety of ways [Demers 2004]. The most accurate methods, such as distributed ray-tracing [Cook 1986] or the accumulation buffer [Haeberli and Akeley 1990], are based on integration over the aperture extent. Post-filtering techniques, such as [Potmesil and Chakravarty 1981; Rokita 1993; Mulder and van Liere 2000; Bertalmío et al. 2004], approximate the effect by blurring a sharp image in a depth-dependent fashion. These are usually fast enough for real-time rendering, but often have problems with intensity leakage and partial occlusions. Eliminating these artifacts adds to the complexity of these methods [Scofield 1992; Riguer et al. 2004].

By far, the simplest approach is to use accumulation [Haeberli and Akeley 1990]. Several sharp images are generated under varying camera positions that sample the area of the aperture (for example, using a Poison disk pattern). Averaging the sharp images together produces the appropriate depth of field effect. Although this process is computationally intensive, all images share a considerable amount of spatial coherence and the RRC can be used to significantly reduce rendering cost.

Once the camera positions are determined, each view is generated in sequence. Using RRC lookups, values computed for the last view are reused whenever available. The high amount of coherence between nearby views results on high cache-hit ratios. Figure 7 show results for the same scene used in the motion blur test. Once again, using the RRC allows us to either substantially increase the frame rates or the quality of the renderings. This time, more than twice the number of samples can be used.

4.4 Environment-mapped bump mapping

While the previous applications used the RRC in order to avoid rerendering portions of the scene, section 3.5 describes how the RRC can be used to reduce the variance of super-sampling results. The strategy can be used directly on pixel colors to produce better results at reduced computational cost. We illustrate the technique



(a) 60fps brute-force *(b) 45fps brute-force*

(c) 60fps RRC 30fps brute-force *(d) 45fps RRC 20fps brute-force*

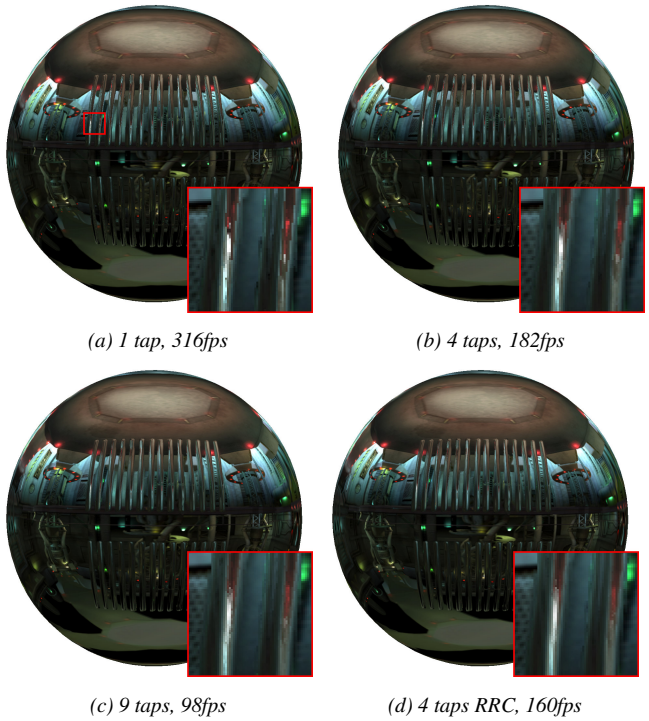
(e) 30fps RRC *(f) 20fps RRC*

Figure 7: *The RRC can be used to optimize motion blur and depth of field rendering. Results of running the brute-force accumulation method at high frame rates are usually unacceptable (top). At the same frame rate, the RRC produces much better results (middle). Matching RRC quality causes the brute-force method to drop the frame rate. Naturally, at these lower frame rates, the RRC produces even higher-quality results (bottom).*

with a difficult problem in real-time computer graphics: the anti-aliasing of bump-mapped environment mapping.

The complication stems from the fact that bump maps can cause the reflection vectors emanating from nearby points on the object to span large regions in the environment map. Since the derivative computation used in mip-level selection is based on finite differences that span one entire pixel, adjacent pixels may end up selecting wildly different mip-levels. The resulting aliasing artifacts can be extremely distracting in animations, particularly for slow motions, which cause the lack of temporal coherence in the aliasing to becomes evident as a shimmering effect. Naturally, smoothing the bump map can defeat the purpose of using it.

A possible solution is to generate a roughness map [Schilling 1997], which precomputes the distribution of normal vectors for each region of the bump map. Unfortunately, this distribution can



(a) 1 tap, 316fps *(b) 4 taps, 182fps*

(c) 9 taps, 98fps *(d) 4 taps RRC, 160fps*

Figure 8: *Bump-mapped environment mapping can result in severe aliasing artifacts (a), especially in animations. In order to eliminate the problem, many samples are required (b, c), which has a negative impact on the frame rate. Using the RRC, we can amortize the super-sampling costs and substantially increase the frame rates (d).*

be highly anisotropic, and current hardware does not have the ability to anisotropically filter across cube map faces.

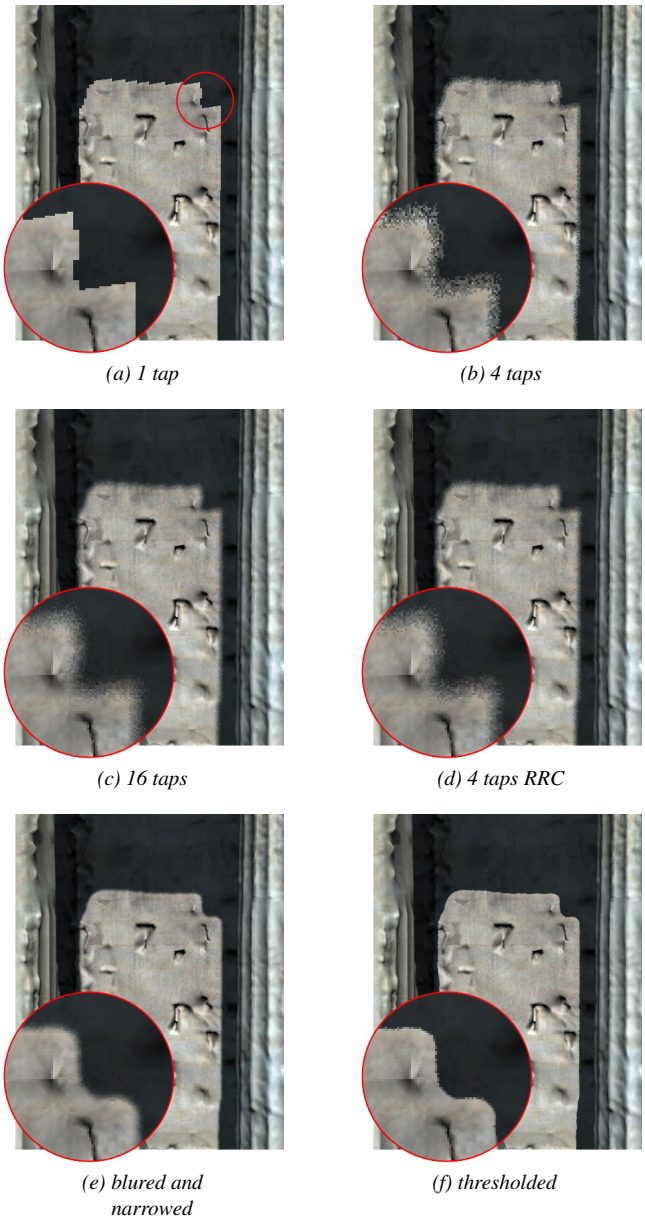
The simplest solution is to super-sample the environment map lookups. In order to do this, we generate interpolated bump-mapped normals for a number of sub-pixel samples within each pixel. We then compute associated reflection vectors, perform an environment map lookup for each one, and average the resulting colors. Due to the severity of the aliasing, many samples are required. Fortunately, it is simple to use the RRC and a recursive filter to accumulate the contribution of several frames. The resulting variance reduction allows us to generate fewer new samples per frame (thus increasing frame rates), while maintaining an acceptable visual quality.

Figure 8a depicts the aliasing artifacts resulting from using only a single texture fetch from the environment map. Figures 8b and 8c show the same object with $4\times$ and $9\times$ super-sampling of the environment map lookups. The reduction in aliasing artifacts come at the cost of a significant drop in frame rates. Figure 8d shows the results using the RRC to combine $4\times$ super-sampling with a $\lambda = 0.6$ recursive filter. The resulting quality surpasses that of $9\times$ super-sampling (it is roughly equivalent to $16\times$), but renders considerably faster.

4.5 Shadow mapping

Shadows not only make synthetic images much more realistic, but also provide important visual cues on the relative position of objects and light sources. For these reasons (and because current graphics hardware is powerful enough), shadow casting has become a requirement in modern real-time rendering applications.

For a recent survey on shadow casting algorithms, see Hasenfratz et al. [2003]. Here we concentrate on an increasingly popular approach: Shadow Mapping [Williams 1978]. The idea is to render the scene twice. On the first pass, the scene is rendered from the point of view of the light source, and depth values are stored



(a) 1 tap *(b) 4 taps*

(c) 16 taps *(d) 4 taps RRC*

(e) blured and narrowed *(f) thresholded*

Figure 9: *The RRC can be used to super-sample shadow-map tests. The images show a closeup of the Parthenon. (a) When the resolution of the shadow map is not high enough, aliasing effects are clearly visible. (b) PCF turns aliasing into high-frequency noise by averaging the results of several taps. (c) Increasing the number of taps makes the noise barely visible, but can be too expensive. (d) Amortized super-sampling can eliminate the additional cost. (e) Shadow boundaries can be blured and narrowed in screen space for added quality. (f) Approximate, alias-free hard shadows can be obtained by thresholding.*

in a shadow map. On the second pass, the scene is rendered from the observer's point of view. While each pixel is generated, it is transformed into the light source's reference frame, and tested for visibility against the shadow map. Failure means the pixel is in shadow.

Although it is extremely simple and general, shadow mapping is plagued by aliasing problems, because the sampling densities on the screen and on the shadow map can be considerably different (see figure 9a). One solution is to increase the effective resolution of the shadow map [Fernando et al. 2001; Stamminger and Drettakis 2002]. A simpler alternative is to use the Percentage Closer Filtering (PCF) of Reeves et al. [1987] (see figure 9b). The idea is to integrate the result of the shadow tests over a neighborhood of the shadow map. The integration is performed stochastically, with a Poisson disk sampling pattern, which transforms aliasing into high-frequency noise. The noise becomes barely visible when 16 taps into the shadow map are averaged together (figure 9c).

This sampling process is directly amenable to optimization by the amortized super-sampling method of section 3.5 (see figure 9d). We compute PCF results at each frame, randomly rotating the sampling patterns each time (to make them independent). Using the RRC and a recursive filter with $\lambda = 3/5$, the variance is reduced to $1/4$ the original. This effectively renders a 4-tap PCF as good as a much more expensive 16-tap PCF (contrast figures 9c and 9d).

To reduce the amount of noise even further, we can apply a screen space Gaussian blur to the cached PCF values, by rendering a full-screen quadrilateral. The accumulation process then causes the contribution of older cached values to be progressively smoother. Finally, the width of the shadow transitions can be narrowed by remapping the PCF values with a smooth step function. Figure 9e shows the result of these two extra steps. Noise levels are so small that the shadow boundaries can be thresholded to produce approximate, alias-free hard shadows (see figure 9f). The method runs at the same speed as the rotated 4-tap PCF, but produces substantially better results.

5 Conclusions

In this paper, we presented the Real-Time Reprojection Cache, a simple, efficient, and effective technique to cache surface information across frames. This information can be used to improve the quality, amortize the cost, or increase the rendering speed of subsequent frames. We demonstrated the effectiveness of the RRC by presenting a variety of concrete examples.

Limitations: The main underlying assumption in the use of the RRC is that reprojection is essentially free. This is true whenever the cost of shading a pixel is high. Conversely, applications dealing with high geometric complexity and low pixel shading costs might not benefit at all from the technique. This problem can be aggravated if the per-vertex transformations are expensive, since at each frame these operations have to be repeated with the cache-time parameters.

A limitation of the amortized super-sampling is the inertia introduced by the memory of the recursive filter. The effect is visible when surface properties are changing with time. In that case, choosing high values for λ (above 0.7) can cause a trailing effect, not unlike motion blur. If frame rates are not high-enough, this can become unacceptable. In that case, lower values of λ usually solve the problem, at the expense of variance reduction.

Future work: Naturally, we have not explored all applications for the RRC. We are experimenting with a technique which we call *amortized tiled rendering*. The idea is to alternatingly render half of each frame from scratch, and use the previous frame as a cache while rendering the other half. Preliminary results show that this technique can increase the effective frame rate by almost a factor of two, with little noticeable quality loss. Naturally, the idea could be pushed even further, by rerendering only $1/3$ or $1/4$ of the pixels every frame.

It would be also interesting to use our technique to guide an automatic per-pixel selection of shader level-of-detail. A set of progressively cheaper shaders for the same effect could be produced automatically [Olano et al. 2003; Pellacini 2005] or by hand. Notice that reprojection gives the application access to the exact *motion field* for the animation sequence. The speed at which a surface point moves on screen could be used to dynamically select among the shaders, including reusing the result of a cache lookup. This

could potentially result in higher frame rates at no perceptible quality loss, especially if motion blur is involved.

References

ADELSON, S. J. and HODGES, L. F. 1993. Stereoscopic ray-tracing. *The Visual Computer*, 10(3):127–144.

ADELSON, S. J. and HODGES, L. F. 1995. Generating exact ray-traced animation frames by reprojection. *IEEE Computer Graphics and Applications*, 15(3):43–52.

BADT, JR., S. 1988. Two algorithms for taking advantage of temporal coherence in ray tracing. *The Visual Computer*, 4(3):123–132.

BALA, K., DORSEY, J., and TELLER, S. 1999. Radiance interpolants for accelerated bounded-error ray tracing. *ACM Transactions on Graphics*, 18(3):213–256.

BERTALMÍO, M., FORT, P., and SÁNCHEZ-CRESPO, D. 2004. Real-time, accurate depth of field using anisotropic diffusion and programmable graphics cards. In *3DPVT*, pages 767–773.

BISHOP, G., FUCHS, H., MCMILLAN, L., and ZAGIER, E. J. S. 1994. Frameless rendering: Double buffering considered harmful. In *Proc. of ACM SIGGRAPH 94*, ACM Press/ACM SIGGRAPH, pages 175–176.

CHEN, S. E. and WILLIAMS, L. 1993. View interpolation for image synthesis. In *Proc. of ACM SIGGRAPH 93*, ACM Press/ACM SIGGRAPH, pages 279–288.

COOK, R. L. 1986. Stochastic sampling in computer graphics. *ACM Transactions on Graphics*, 5(1):51–72.

COOK, R. L., PORTER, T., and CARPENTER, L. 1984. Distributed ray tracing. *Computer Graphics (Proc. of ACM SIGGRAPH 84)*, 18(3):137–145.

DAYAL, A., WOOLLEY, C., WATSON, B., and LUEBKE, D. 2005. Adaptive frameless rendering. In *Eurographics Symposium on Rendering*, Rendering Techniques, Springer-Verlag, pages 265–275.

DEMERS, J. 2004. *Depth of Field: A Survey of Techniques*, chapter 23, pages 375–390. GPU Gems. Addison-Wesley Professional.

DIPPÉ, M. A. Z. and WOLD, E. H. 1985. Antialiasing through stochastic sampling. *Computer Graphics (Proc. of ACM SIGGRAPH 85)*, 19(3):69–78.

DUBOIS, E. 2001. A projection method to generate anaglyph stereo images. In *ICASSP*, volume 3, IEEE Computer Society Press, pages 1661–1664.

FERNANDO, R., FERNANDEZ, S., BALA, K., and GREENBERG, D. P. 2001. Adaptive shadow maps. In *Proc. of ACM SIGGRAPH 2001*, ACM Press/ACM SIGGRAPH, pages 387–390.

HAEBERLI, P. and AKELEY, K. 1990. The accumulation buffer: hardware support for high-quality rendering. *Computer Graphics (Proc. of ACM SIGGRAPH 90)*, 24(4):309–318.

HASENFRATZ, J.-M., LAPIERRE, M., HOLZSCHUCH, N., and SILLION, F. 2003. A survey of real-time soft shadows algorithms. *Computer Graphics Forum*, 22(4):753–774.

HAVRAN, V., DAMEZ, C., MYSZKOWSKI, K., and SEIDEL, H.-P. 2003. An efficient spatio-temporal architecture for animation rendering. In *Eurographics Symposium on Rendering*, Rendering Techniques, Springer-Verlag, pages 106–117.

KOREIN, J. and BADLER, N. 1983. Temporal anti-aliasing in computer generated animation. *Computer Graphics (Proc. of ACM SIGGRAPH 83)*, 17(3):377–388.

MARK, W. R., MCMILLAN, L., and BISHOP, G. 1997. Post-rendering 3D warping. In *Symposium on Interactive 3D Graphics*, pages 7–16.

MCMILLAN, L. and BISHOP, G. 1995. Head-tracked stereoscopic display using image warping. In S. Fisher, J. Merritt, and B. Bolas, editors, *SPIE*, volume 2049, pages 21–30.

MULDER, J. D. and VAN LIERE, R. 2000. Fast perception-based depth of field rendering. In *Proc. of the ACM Symposium on Virtual Reality Software and Technology*, ACM Press, pages 129–133.

OLANO, M., KUEHNE, B., and SIMMONS, M. 2003. Automatic shader level of detail. In *Proc. of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, Eurographics Association, pages 7–14.

PELLACINI, F. 2005. User-configurable automatic shader simplification. *ACM Transactions on Graphics (Proc. of ACM SIGGRAPH 2005)*, 24(3):445–452.

POTMESIL, M. and CHAKRAVARTY, I. 1981. A lens and aperture camera model for synthetic image generation. *Computer Graphics (Proc. of ACM SIGGRAPH 81)*, 15(3):297–305.

POTMESIL, M. and CHAKRAVARTY, I. 1983. Modeling motion blur in computer-generated images. *Computer Graphics (Proc. of ACM SIGGRAPH 83)*, 17(3):389–399.

REEVES, W. T., SALESIN, D. H., and COOK, R. L. 1987. Rendering antialiased shadows with depth maps. *Computer Graphics (Proc. of ACM SIGGRAPH 87)*, 21(4):283–291.

REGAN, M. and POSE, R. 1994. Priority rendering with a virtual reality address recalculation pipeline. In *Proc. of ACM SIGGRAPH 94*, ACM Press/ACM SIGGRAPH, pages 155–162.

RIGUER, G., TATARCHUK, N., and ISIDORO, J. 2004. *Real-time depth of field simulation*, pages 529–556. ShaderX^2. Wordware Publishing, Inc.

ROKITA, P. 1993. Fast generation of depth of field effects in computer graphics. *Computers & Graphics*, 17(5):593–595.

SANDER, P. V., ISIDORO, J. R., and MITCHELL, J. L. 2005. Computation culling with explicit early-z and dynamic flow control. In *GPU Shading and Rendering*, chapter 10. ACM SIGGRAPH Course 37 Notes.

SCHILLING, A. G. 1997. Antialiasing of bump-maps. Technical Report WSI-97-15, Wilhelm-Schickard-Institut für Informatik.

SCOFIELD, C. 1992. *$2\frac{1}{2}$-D Depth-of-Field Simulation for Computer Animation*, chapter 1.8, pages 36–38. Graphics Gems III. Morgan Kaufmann.

SIMMONS, M. and SÉQUIN, C. H. 2000. Tapestry: A dynamic mesh-based display representation for interactive rendering. In *Eurographics Workshop on Rendering*, Rendering Techniques, Springer-Verlag, pages 329–340.

STAMMINGER, M. and DRETTAKIS, G. 2002. Perspective shadow maps. *ACM Transactions on Graphics (Proc. of ACM SIGGRAPH 2002)*, 21(3):557–563.

TOLE, P., PELLACINI, F., WALTER, B., and GREENBERG, D. P. 2002. Interactive global illumination in dynamic scenes. *ACM Transactions on Graphics (Proc. of ACM SIGGRAPH 2002)*, 21 (3):537–546.

TORBORG, J. and KAJIYA, J. T. 1996. Talisman: commodity realtime 3D graphics for the PC. In *Proc. of ACM SIGGRAPH 96*, ACM Press/ACM SIGGRAPH, pages 353–363.

WALTER, B., DRETTAKIS, G., and PARKER, S. 1999. Interactive rendering using the render cache. In *Eurographics Workshop on Rendering*, Rendering Techniques, Springer-Verlag, pages 19–30.

WARD, G. and SIMMONS, M. 1999. The holodeck ray cache: an interactive rendering system for global illumination in nondiffuse environments. *ACM Transactions on Graphics*, 18(4):361–368.

WILLIAMS, L. 1978. Casting curved shadows on curved surfaces. *Computer Graphics (Proc. of ACM SIGGRAPH 78)*, 12(3):270–274.

WLOKA, M. M. and ZELEZNIK, R. C. 1996. Interactive real-time motion blur. *The Visual Computer*, 12(6):283–295.

ZHU, T., WANG, R., and LUEBKE, D. 2005. A GPU accelerated render cache. In *Pacific Graphics (short paper)*.