Graphics Hardware (2007)
Timo Aila and Mark Segal (Editors)

Accelerating Real-Time Shading with Reverse Reprojection Caching

Diego Nehab[1] Pedro V. Sander[2] Jason Lawrence[3] Natalya Tatarchuk[4] John R. Isidoro[4]

[1]Princeton University [2]Hong Kong University of Science and Technology [3]University of Virginia [4]Advanced Micro Devices, Inc.

Abstract

Evaluating pixel shaders consumes a growing share of the computational budget for real-time applications. However, the significant temporal coherence in visible surface regions, lighting conditions, and camera location allows reusing computationally-intensive shading calculations between frames to achieve significant performance improvements at little degradation in visual quality. This paper investigates a caching scheme based on reverse reprojection which allows pixel shaders to store and reuse calculations performed at visible surface points. We provide guidelines to help programmers select appropriate values to cache and present several policies for keeping cached entries up-to-date. Our results confirm this approach offers substantial performance gains for many common real-time effects, including precomputed global lighting effects, stereoscopic rendering, motion blur, depth of field, and shadow mapping.

Categories and Subject Descriptors (according to ACM CCS): I.3.6 [Computer Graphics]: Graphics data structures and data types I.3.6 [Computer Graphics]: Interaction techniques

1 Introduction

As the power and flexibility of dedicated graphics hardware continue to grow, a clear tendency in real-time rendering applications has been the steady increase in pixel shading complexity. Today, a considerable portion of the graphics processing budget is spent evaluating pixel shaders. Recent research has therefore investigated general techniques for optimizing these computations, either by reducing their complexity [OKS03, Pel05], or by reducing the number of fragments generated [DWS*88, NBS06].

In this paper, we develop a caching strategy that exploits the inherent spatio-temporal coherence of real-time shading calculations (Figure 1). At very high frame rates, and between consecutive frames, there is usually very little difference in the camera and lighting parameters, as well as in the set of visible surface points, their properties, and final appearance. Therefore, recomputing each frame from scratch is potentially wasteful. This coherence can be exploited to reduce the average cost of generating a single frame with a caching mechanism that allows storing, tracking and retrieving the results of expensive calculations within a pixel shader between consecutive frames. Although a number of caching techniques have been developed in different rendering contexts, ours is uniquely designed for interactive applications running on commodity GPUs which places strict constraints on the computational resources and bandwidth that can be allocated to cache maintenance.

We introduce a new caching strategy designed for real-time applications based on *reverse reprojection*. As each frame is generated, we store the desired data at visible surface points in viewport-sized, off-screen buffers. As each pixel is generated in the following frame, we reproject its surface location into the last frame to determine if it was previously visible and thus present in the cache. If available, we can reuse its cached value in place of performing a redundant and potentially expensive calculation, otherwise we recompute it from scratch and make it available in the cache for the next frame. Our approach does not require complex data structures or bus traffic between the CPU and GPU, provides efficient cache access, and is simple to implement.

We demonstrate the utility of our approach by showing how it can be used to accelerate a number of common real-time shading effects. We report results for scenes that incorporate precomputed global lighting effects, stereoscopic

Graphics Hardware 2007, San Diego, CA, August 04-05, 2007.
© 2007 ACM 978-1-59593-625-7/07/0008 $5.00

(a) Parthenon *(b) Coherence* *(c) Heroine* *(d) Coherence* *(e) Ninja* *(f) Coherence*

Figure 1: *Real-time rendering applications exhibit a considerable amount of spatio-temporal coherence. This is true for camera motion, as in the (a) Parthenon sequence, as well an animated scenes such as the (c) Heroine and (e) Ninja sequences. We visualize this property in coherence maps (b, d, and f) that show newly visible surface points in red and points that were visible in the previous frame in green. Our method allows pixel shaders to associate and store values with visible surface points that can be efficiently retrieved in the following frame. For many applications, this provides substantial performance improvements and introduces minimal error into the final shading.*

rendering, motion blur, depth of field, and shadow mapping. In summary, this paper makes the following contributions:

- We introduce a new caching scheme based on reverse reprojection targeted for real-time shading calculations. It provides a general and efficient mechanism for storing, tracking, and sharing surface information through time (Section 3);
- We develop a set of guidelines for selecting what values to cache and under what circumstances (Section 4);
- We design and evaluate a variety of refresh policies for keeping cached entries up-to-date (Section 5);
- We develop a theory for amortizing the cost of stochastically estimating a quantity across multiple frames (Section 6);
- We present a working prototype system and evaluate our caching technique for a variety of common real-time rendering applications (Section 7).

2 Related work

Reusing expensive calculations across frames generated at nearby viewpoints or consecutively in animation sequences has been studied in many rendering contexts. One of the key differences in our approach is that we do not attempt to reuse visibility information: only shading computations are reused. Furthermore, we focus on exploiting coherence in real-time pixel shaders with an approach that is targeted for commodity graphics hardware. In this context, it is important to minimize the cache overhead and guarantee that all the calculations occur on the GPU (thus limiting any bus traffic between the GPU and CPU). On the other hand, there are a number of computational advantages to implementing a caching mechanism using modern graphics hardware; our approach leverages hardware-supported Z-buffering and native texture filtering.

CPU-based methods to accelerate the off-line rendering of animation sequences [Bad88, AH95, BDT99] generally *scatter* shading information from one frame into the following by means of *forward reprojection*. This produces gaps and occlusion artifacts that must be explicitly fixed, increasing the complexity of these techniques and reducing their efficiency. Similar problems caused by forward reprojection plague methods that attempt to bring interactivity to off-line renderers [BFMZ94, WDP99], even in recent GPU-accelerated revisions [DWWL05, ZWL05].

A better alternative is to use *reverse reprojection* and take advantage of hardware support for *gathering* samples into the current frame that were generated previously. This is possible if the GPU has access to scene geometry, even in simplified form. [CCC87], [WS99], [SS00], and [TPWG02] follow this approach, and store samples in world-space. Unfortunately, these techniques require complex data structures that are maintained on the CPU and introduce considerable bus-traffic between the CPU and GPU (particularly for real-time applications).

Instead of directly associating samples with geometry, it is possible to store them in textures that are mapped to static geometry [SHSS00]. Methods that replace geometry by simpler image based representations [RH94, MS95] to accelerate real-time rendering of complex environments employ a similar idea. However, maintaining and allocating these textures also requires CPU intervention. In contrast, our approach uses only off-screen buffers that are maintained entirely on the GPU. Resolving cache hits and misses is natively enforced by the Z-buffer and retrieval is handled using native texture filtering. Unlike hardware-based systems [RP94, TK96] that also exploit coherence, our caching scheme targets commodity hardware and does not require explicit control from the programmer.

A final group of related methods exploit spatial coherence to efficiently generate novel views from a set of images [CW93, MB95, MMB97]. Although our method can also be interpreted as warping rendered frames, it was designed to support dynamically generated scenes, such as those found in games.

3 Reverse reprojection caching

The schematic diagram in Figure 2 illustrates the the type of single-level cache we use to improve the performance of a

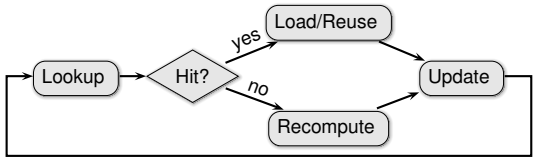


Figure 2: *Schematic diagram of a single-level cache we use to accelerate pixel-level shading calculations.*

pixel shader. As each pixel is generated the shader tests if the result of a particular calculation is available in the cache. If so, the shader can reuse this value in the calculation of the final pixel color. Otherwise, the shader executes as normal and stores the cacheable value for potential reuse during the next frame. Note that the value stored in the cache need not be the final pixel color, but can be any intermediate calculation that would benefit from this type of reuse.

Three key factors determine whether a shader modified with this type of cache is superior to its original version. First, the hit path must be executed often enough to justify its use. Second, the total cost of evaluating the shader in the case of a cache hit must be less than the unmodified shader (this includes the overhead of managing the cache). Third, the values stored in the cache must remain relevant across consecutive frames so as not to introduce significant errors into the shading.

With this criteria in mind, we propose a very simple cache policy: simply store values associated with visible surface points in viewport-sized, off-screen buffers. This permits efficient reads and writes and provides a high rate of cache hits (we discuss the third factor of maintaining relevant information in the cache in Section 4).

Figure 3 shows quantitative evidence that this policy leads to a high rate of cache hits. It plots the ratio of pixels that remain visible across two consecutive frames for the animation sequences in Figure 1. The Parthenon sequence uses static geometry and a moving camera to generate a fly-by of the temple. Note that its slow camera motion results in very high and uniform hit rates. The Heroine sequence shows an animated character with weighted skinned vertices running past the camera. The rapid increase in coherence at the beginning of the sequence is due to her entering the scene from the right. Finally, the Ninja sequence shows an animated fighter performing martial arts maneuvers. His fast kicks and movements cause the periodic dips in the corresponding plot. For all the scenes we have used to test our approach we observed hit rates typically in excess of 90%.

To meet the second criterion of providing efficient cache access, we note that our policy of maintaining cache entries exclusively for visible surface points offers a number of computational advantages:

- Using just one entry per pixel, the cache memory requirements are *output sensitive* and thus independent of scene complexity;
- Cache entries are in one-to-one correspondence with screen pixels, so no coordinate translation is needed during writes;
- The coordinate translation needed for cache lookups can be efficiently performed within the vertex shader (Section 3.1);
- The depth of each cached entry, required during the lookup process, is already available in the Z-buffer (Section 3.2);
- Native support for filtered texture lookups enables robust detection of cache hits (Section 3.2), and high-quality cache reads (Section 3.3);
- Data never leaves the GPU, thus eliminating inefficient bus-traffic with the CPU.

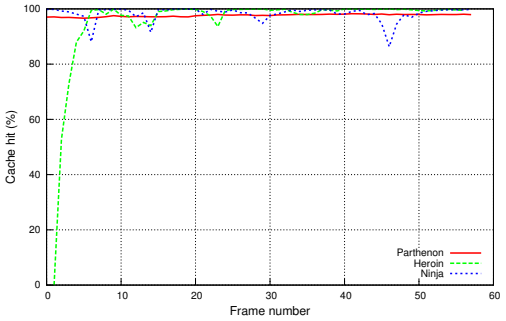


Figure 3: *Percentage of surface area that was mutually visible between consecutive frames for the animation sequences in Figure 1. These high coherence rates (generally above 90%) justifies our policy of maintaining cache entries exclusively at visible surface points.*

We next detail how a pixel shader can be modified to provide this type of caching.

3.1 Determining cache coordinates

The main computational challenge we face is to efficiently compute the location of a pixel's corresponding scene point in the previous frame. We leverage hardware support for perspective-correct interpolation [HM91] and move the bulk of this computation from the pixel level to the vertex level.

At time $t-1$, assume the result of a calculation that occurs at each pixel has been stored in a screen-space buffer (Figure 4, left). At the next frame, the homogeneous projection space coordinates $(x_t, y_t, z_t, w_t)_v$ of each vertex v at time t are calculated in the vertex shader, to which the application has provided the world, camera and projection matrices and any animation parameters (such as tween factors and blending matrices used for skinning). In our case, the application also provides the vertex program with the transformation parameters at $t-1$, allowing it to compute the projection-space coordinates of the same vertex at the previous frame $(x_{t-1}, y_{t-1}, z_{t-1}, w_{t-1})_v$. These coordinates become attributes of each transformed vertex (Figure 4, right), which in turn causes the hardware to interpolate them, automatically giving each pixel p access to the projection-space

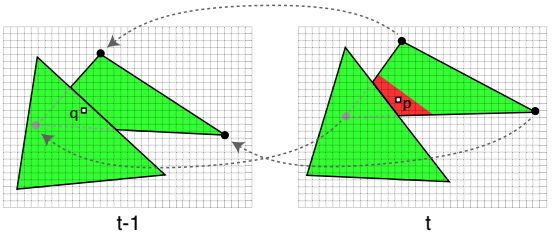


Figure 4: ***Left:*** *Shading calculations and pixel depths in frame $t-1$ are stored in screen-space buffers.* ***Right:*** *In the next frame, each vertex is also transformed by the model, camera and projection matrices (along with any animation parameters) at time $t-1$. These values become per-vertex attributes that undergo perspective-correct interpolation, giving each pixel access to its position in the cache. To detect cache misses, we compare the reprojected depth of a pixel p to the depth stored at its position in the cache at q.*

coordinates $(x_{t-1}, y_{t-1}, z_{t-1}, w_{t-1})_p$ of the generating surface point at time $t-1$. The final cache coordinates p_{t-1} are obtained with a simple division by $(w_{t-1})_p$ within the pixel shader.

3.2 Detecting cache misses

A visible surface point p at time t may have been occluded at time $t-1$ by an unrelated point q (Figure 4). Except in the case of intersecting geometry, it is not possible for the depths of points p and q to match at time $t-1$. We therefore compare the depth of p at time $t-1$, which was computed along with its cache coordinates, to the value in the depth buffer at time $t-1$ (much like a shadow map test). If the cached depth is within ε of the expected depth for p, we report a cache hit. Otherwise, we report a cache miss. We use bilinear interpolation to reconstruct the depth stored at the previous frame. The weighted sum of values across significant depth variations will not match the reprojected depth received from the vertex shader, therefore automatically resulting in a cache miss. As a result, this greatly reduces the chance of reporting a false cache hit and improperly reconstructing values near depth discontinuities. To further improve robustness, we set ε to the smallest Z-buffer increment (this value could also be calculated using a technique such as [AS06]).

3.3 Cache resampling

A key advantage of our approach over those based on forward reprojection is that it transforms the problematic scattering of cached samples into a manageable gathering process. However, since reprojected pixels will not, in general, map to individual cached samples (Figure 4), some form of resampling is necessary. The uniform structure of the cache and native hardware support for texture filtering greatly simplify this task. In fact, except at depth discontinuities, cache lookups can be treated exactly as texture lookups.

The best texture filtering method depends on the data being cached and its function in the pixel shader. Nearest-neighbor interpolation is sufficient for cached data that varies smoothly, or that is further processed by the shader. On the other hand, bilinear interpolation is appropriate when there is considerable spatial variation in the cached entries, especially if the value will be displayed. Regardless of the method, however, repeatedly resampling the cache across multiple frames will eventually attenuate high-frequency content.

Although we can avoid resampling the cache near occlusion boundaries by using bilinear interpolation to reconstruct the depth (Section 3.2), this would not prevent wider reconstruction kernels (e.g., trilinear or anisotropic filters) from integrating across unrelated data. However, in practice there is little change in the scene between frames, limiting the amount of distortions in the reprojection map and eliminating the need to use more sophisticated reconstruction methods.

3.4 Control flow strategies

The fact that we can distinguish between cache hits and misses allows us to write pixel shaders that execute a different path for each of these cases. We refer to the desired code paths as the *hit shader* and *miss shader*, respectively.

The most straightforward approach is to branch between the hit and miss shaders according to the depth test. If the hardware supports dynamic flow control, the cost of execution will depend on the branch taken. However, one important feature of graphics hardware is that computations are evaluated in *lock-step*: entire blocks of the screen are executed in parallel, each at a rate proportional to its most expensive pixel. Therefore, one cache miss within a large block of pixels will penalize the execution time for that entire region. Fortunately, the spatial coherence in which pixels are visible in sequential frames (Figure 1) largely mitigates this effect, causing large contiguous regions of the screen to follow the same control path.

An alternative to dynamic flow control, and one that avoids penalties due to lock-step execution, is to rely on early Z-culling [SIM05] which tests the depth of a pixel before evaluating the associated shader. During a first pass, the cache lookup is performed and the hit shader is executed if it succeeds. On a miss, the pixel is simply depth-shifted to prime the Z-buffer. During the second pass, early Z-culling guarantees that the miss shader will only be executed on those pixels, and only once per pixel. Unfortunately, in current hardware the depth-shift operation prevents the use of early Z-culling on the first pass. However, since the hit shader is relatively cheap, this does not incur a substantial drop in performance. Most results in this paper were generated using the early Z-culling approach in order to support the fine-grained, randomized refresh strategy described in Section 5.2.

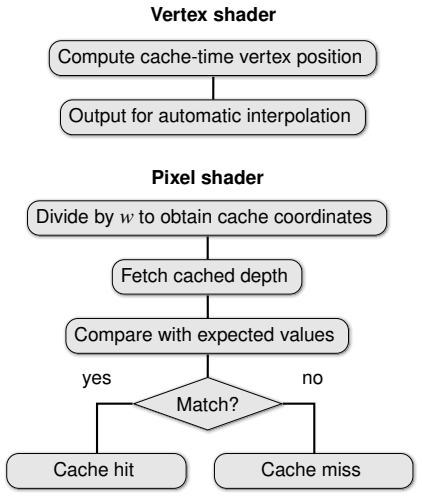


Figure 5: *The vertex shader calculates the cache-time position of each vertex and the pixel shader uses the interpolated position to test the visibility of the current point in the previous frame-buffer.*

3.5 Computational Overhead

Figure 5 shows a schematic description of the cache lookup process. One modification to the vertex shader is that the application must send it two sets of transformation parameters. Because many real-time applications often reach the hardware limit for this type of storage any increase could be problematic, although recent hardware revisions provide substantially larger limits to comply with the Direct3D® 10 system [Bly06].

Additionally, our strategy requires transforming the geometry twice: once for the current and once for the previous frame's viewing parameters. However, if the rendering cost is dominated by pixel processing, the hardware will certainly welcome this trade-off between additional vertex load and a substantial reduction in pixel load. In addition, since latest GPUs are unified architectures, the system can benefit from any significant reduction in pixel processing, even if pixel processing is not the only factor in determining the rendering cost.

At the pixel-level, our caching strategy requires one additional division and two texture lookups. One of the lookups is for the cached depths, and the other is for the payload information for that pixel. There is also overhead associated with the dynamic control flow mechanism (Section 3.4). Naturally, in order to justify caching, the computations being replaced must be more expensive than the added overhead.

With regard to off-screen buffer management, caching a single calculation at each pixel requires one additional depth and color buffer equal in size to the viewport. Caching multiple values can be done by storing entries in different color channels or within multiple buffers (for many applications, a single unused alpha channel might suffice) for which available graphics hardware supports up to eight concurrent rendering targets [Bly06].

4 Determining what to cache

Although reusing expensive shading calculations can reduce the latency of generating a single frame, this introduces error into the shading proportional to the calculation's rate of change between frames. For example, caching the final color of a highly polished object would not capture the shifting specular highlights as the camera and lighting change.

When selecting a value to cache, the programmer should seek to maximize the ratio of its associated computational effort (e.g., number of machine instructions and texture fetches) relative to the magnitude of its derivative between frames. Although we leave the final decision of what to cache to the programmer, we have identified several categories of shading calculations that meet these criteria:

- Shading models that incorporate an expensive calculation which exhibits weak directional dependence (e.g., a procedurally generated diffuse albedo);
- Multi-pass rendering effects that combine several images from nearby views (e.g., motion blur and depth of field);
- Effects that require sampling a function that is slowly varying at each pixel (e.g., jittered super-sampling shadow maps).

Interactive shading techniques that incorporate indirect (or global) lighting effects often fall in the first category. Indeed, a scene's global shading component typically exhibits low-frequency directional dependence [NKGR06], but simulating these effects can be computationally intensive. The simplest example is the diffuse component in local lighting models which is entirely independent of the viewing angle and (ignoring the cosine fall-off) the direction of incident lighting. In cases where this is modeled as a complex procedure [Per85], reusing this value across multiple frames will improve the rendering performance without affecting the accuracy of the shading (Figure 6). Other examples include methods that precompute the global transfer of light within a scene for interactive display [SKS02]. These require expensive local shading calculations that often result in values with low-frequency directional dependence. In Section 7, we apply our technique to accelerate a method for rendering the single-scattering component of translucent objects under complex illumination [WTL05].

Multi-pass effects that combine several images rendered from nearby views can also be accelerated by caching expensive calculations performed during the first pass which are then reused in subsequent passes. Because the camera motion between passes is known and fixed, the error introduced with our approach is bounded. Furthermore, the proximity of the combined views relaxes the requirement that the cached calculation exhibit limited directional dependence. We present results for motion blur, depth-of-field, and stereoscopic effects in Section 7.

Our caching infrastructure also supports amortizing the cost of stochastically sampling a function over multiple

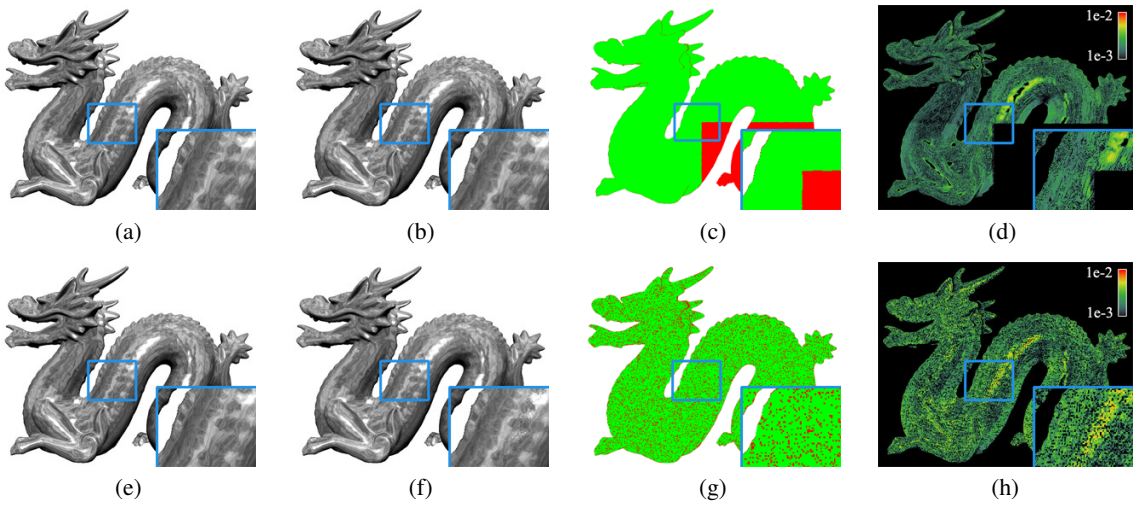


Figure 6: *Comparison of refresh policies. (a and e) One frame from an interactive sequence of the dragon model shaded with a Perlin noise diffuse layer and a specular layer as the user adjusts the position of the camera and a point light source (the same image is reproduced for side-by-side comparisons). (b) Result of caching the final pixel color using four tiled refresh regions. (c) Coherence map showing which pixels are recomputed (red) and which are retrieved from the cache (green). (d) False-color visualization of the shading error at each pixel. Note that the error is zero at pixels inside the tile that is being refreshed. (f,g,h) Result of using randomly distributed refresh regions for the same scene along with associated coherence maps showing the distribution of cache hits and the shading error. Both policies refresh entries once every four frames and provide performance gains of nearly* 100% *(i.e., 35fps for conventional rendering vs. 60fps and 67fps for tile-based and random, respectively).*

frames (Section 6). This is best suited for sampling functions that are stationary (or slowly varying) at each pixel. In Section 7.3, we describe how to improve the performance of a popular technique for rendering antialiased shadow edges from shadow maps.

5 Refreshing cached values

Scene motion, varying surface parameters and repeated resampling will eventually degrade the accuracy of cached entries, and they must be periodically refreshed. We can control the shading error introduced by reusing a calculation if we set its refresh rate proportional to its rate of change between frames. Of course, predicting its change *a priori* is not always possible as it might depend on scene motion due to unknown user input. We instead rely on the programmer to select values according to the guidelines in Section 4 and manually set an appropriate refresh rate.

We can guarantee the entire cache is refreshed at least once every n frames by updating a different region of size $1/n$ at each frame. This distributes the computational overhead evenly in time and results in smoother animations. We compare two strategies for partitioning the screen into refresh regions.

5.1 Tiled refresh regions

We partition the screen into a grid of n non-overlapping tiles and maintain a *global clock* t that is incremented at each frame, and pass it to the pixel shader as a uniform attribute. As each pixel is generated, the pixel shader adds its tile index i to the clock, triggering a refresh (i.e., executing the miss shader *even on a cache hit*) on the condition:

$$(t+i) \mod n = 0. \tag{1}$$

This has the effect of refreshing each tile in turn. Figure 6 analyzes the effect of this refresh strategy on a simple shader that combines a Perlin noise diffuse layer with a Blinn-Phong specular layer. As the user interactively adjusts the camera and point light source, we cache the final color and refresh its value once every four frames. Because accessing the cache is considerably less expensive than performing this calculation, we double the performance at negligible error.

5.2 Randomly distributed refresh regions

We have also experimented with refresh regions that form a randomly distributed pattern across the entire screen (see Figure 6). We have found these patterns produce less perceptually objectionable artifacts, exchanging sharp discontinuities at tile boundaries for high-frequency noise that is evenly distributed across the image. However, this strategy can degrade performance if naively implemented in modern graphics hardware that executes neighboring pixels in lockstep (Section 3.4). In these cases, it is important to use early Z-culling to provide flow control which allows updating randomly distributed regions as small as 2×2 pixels.

These refresh patterns can be implemented by precomputing and storing a randomly distributed integral offset with each pixel. We generate these so that 2×2 regions have

the same offset (see Figure 6). During rendering, the pixel shader accesses its offset d according to its screen position and adds this value to the global clock t, refreshing on the condition:

$$(t+d) \mod n = 0. \tag{2}$$

5.3 Implicit refresh

Many shaders that benefit from our technique do not require explicitly refreshing the cache. Effects computed in multiple rendering passes can reap the benefits of caching by reusing values only between the passes within a single frame. The cache is therefore completely refreshed in the first pass of the following frame, avoiding any overhead required for the policies described above. The same is true of our method for amortized sampling (Section 6). In this case, values are quickly and smoothly attenuated with the accumulation of newer samples.

6 Amortized sampling

Many quantities in graphics result from a stochastic process that combines a number of randomly chosen samples of a function [DW85, Coo86]. Interactive applications are limited by the maximum number of samples their computational budget can afford. Our caching infrastructure allows amortizing the cost of sampling a function over multiple frames, thereby improving the quality of these estimates at comparable frame rates. As discussed in Section 4, this method is best suited for sampling functions at each pixel that are stationary or slow-varying.

Our goal is to compute

$$\int_\Omega f(x)d\mu(x), \tag{3}$$

where $f(x)$ is the function of interest and Ω is the domain of integration (e.g., the shadow coverage within a singe pixel as in Section 7.3). Monte Carlo techniques [RC04] approximate Equation 3 as the weighted sum of n samples of $f(x)$ chosen according to a probability density $p(x)$:

$$F_n = \frac{1}{n}\sum_{i=1}^{n}\frac{f(x_i)}{p(x_i)}. \tag{4}$$

The variance of this approximation, which measures its quality, is inversely proportional to the number of samples and directly proportional to the quality of the distribution $p(x)$:

$$\mathrm{Var}[F_n] = \frac{1}{n^2}\mathrm{Var}\left[\frac{f(x)}{p(x)}\right]. \tag{5}$$

At each frame, we may replace the cached entry c_t with a weighted combination of its current value and a new sample $f(x_t)$, weighted by its probability:

$$c_{t+1} \leftarrow \lambda c_t + (1-\lambda)\frac{f(x_t)}{p(x_t)}, \text{ where } \lambda \in [0,1). \tag{6}$$

It can be easily shown that c_t is an unbiased estimator for Equation 3, with variance given by:

$$\frac{(1-\lambda)}{(1+\lambda)}\mathrm{Var}\left[\frac{f(x)}{p(x)}\right]. \tag{7}$$

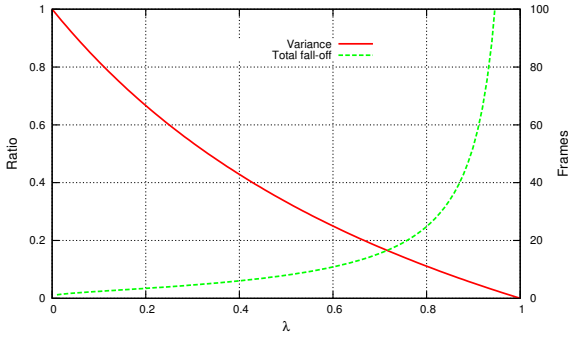


Figure 7: *When performing amortized super-sampling with a recursive filter, there is a trade-off between the amount by which the variance is reduced (the variance curve), and the number of frames that contribute to the current estimate (the total fall-off curve). This trade-off is controlled by the parameter* λ.

Note that the relative contribution of any sample to the current estimate falls off exponentially, with a time constant equal to $\tau = -1/\ln\lambda$. The value of λ therefore controls the trade-off between variance in the estimator and responsiveness to changes in the scene (i.e., changes to $f(x)$). Figure 7 quantifies this relationship where the *total fall-off* is the time, measured in frames, until a sample is scaled by $1/256$ (i.e., completely lost in 8-bits of precision). For example, choosing a value of $\lambda = 3/5$ reduces the variance to $1/4$ the original (Equation 7) and effectively combines samples from the previous 10 frames (also refer to Figures 11b and 11d). Conversely, reducing the variance by a factor of $1/8$ requires setting $\lambda = 7/9$, and increases the total fall-off to 22 frames. In practice, we determine λ empirically.

7 Results

We have used our caching strategy to accelerate several common interactive rendering techniques. These were selected according to the guidelines in Section 4 and include a shading model for translucent objects based on precomputed light transport, several common effects computed in multiple rendering passes, and a technique for rendering antialiased shadow map boundaries.

Our comparisons to conventional rendering methods focus on the trade-off between quality and performance. The trends we report would be similar for applications that exhibit a comparable balance between pixel shading and geometry processing complexity. Our results were generated using a P4 3.2GHz with an ATI X800 graphics card.

7.1 Shading model for translucent materials

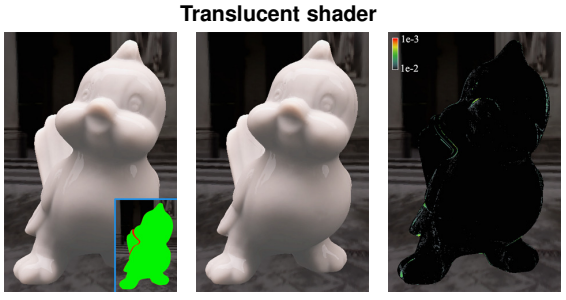
We used our method to accelerate a technique, based on precomputed light transport [SKS02], for interactively rendering translucent objects under complex all-frequency distant illumination [WTL05]. Their model considers the complete Bidirectional Surface Scattering Reflectance Distribution Function (BSSRDF) proposed by [JMLH01], which includes both single and diffuse multiple scattering. The multiple scattering term is independent of view direction and can therefore be captured with a set of per-vertex transfer vectors that are precomputed and compressed using non-linear wavelet approximation [NRH03]. The dot product of these vectors and the environment lighting represented in the same wavelet basis are computed on the CPU and become vertex attributes $\tilde{T}_d$.

They use an approximation for the single scattering term [JB02] that allows decomposing an arbitrary phase function into the sum of K terms, each the product of two functions that depend only on the local light ω_i and view direction ω_o, respectively. They precompute K additional per-vertex colors $\tilde{T}_k$ that capture the product of the environment lighting and these light-dependent terms. The view-dependent terms h_k are stored in texture maps and evaluated at the local refracted view direction $(h_k(\omega_o'))$ during rendering, resulting in the complete shading model[†]:

$$L_o(x_o,\omega_o) = \tilde{T}_d(x_o) + \sum_k h_k(\omega_o')\tilde{T}_k(x_o). \qquad (8)$$

For optically dense materials, the outgoing radiance L_o exhibits very low-frequency directional dependence. However, evaluating Equation 8 requires K texture fetches and can be expensive for accurate decompositions. Therefore, reusing this calculation across multiple frames reduces the cost of generating a single frame at the cost of only marginal shading errors.

Figure 8 compares the performance of the original shader[‡] to the result of applying our caching strategy to reuse the result of Equation 8 at each pixel across consecutive frames. We do not explicitly refresh the cache, but only recompute at cache misses (see the cutout in Figure 8(a)). In the end, we are replacing 4 texture fetches with two (one for resolving cache hits and one for retrieving the actual payload) plus the computational overhead of maintaining the cache. For this scene we observed a 30% improvement in the frame rate. We expect to achieve even better results for more complex precomputed radiance transfer techniques.



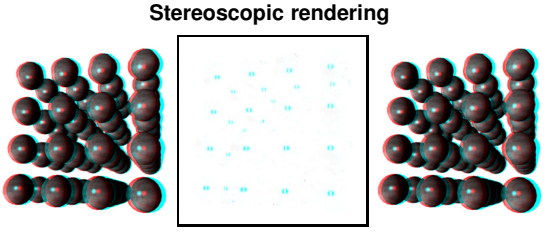
(a) Translucent Shader (b) Caching SS (30% faster) (c) Shading Error

Figure 8: *A translucent shader based on PRT [WTL05] accelerated with our caching strategy. (a) One frame from a sequence in which the user adjusts the position of the bird model under environment lighting. (b) Result of caching and reusing the single-scattering shading calculation (note the cache is never explicitly refreshed as seen in the coherence map in (a,cutout)).*

7.2 Multi-pass rendering effects

Anaglyph stereo images encode a binocular pair of views of a scene in separate color channels and can be generated in two rendering passes (see [Dub01] for a good review). The proximity of these views allows us to reuse shading information computed for one view at mutually visible points in the opposing view. Although prior work has used reprojection to accelerate this technique, it was applied in the context of ray-tracing [AH93] and head-tracked displays [MB95].

Figure 9a demonstrates the effect of reusing the final color of a shading model with a Perlin noise function that requires 512 instructions per pixel (expensive, but not unreasonable). As shown in Figure 9b, the comparison to ground truth reveals noticeable shading errors around specular highlights. In Figure 9c we cache and reuse only the expensive noise



(a) final color *(b) error (enhanced)* *(c) albedo only*

Figure 9: *Rendering stereoscopic images using our caching method to share values at mutually visible points. Caching (a) the final color leads to (b) visual errors near specular highlights. These errors can be eliminated by (c) caching only the surface albedo and recomputing the specular contribution at each frame.*

† For clarity, we omit the Fresnel term and simple surface scattering term used in [WTL05].

‡ We used a Henyey-Greenstein phase function with $g = -0.25$, $K = 4$.

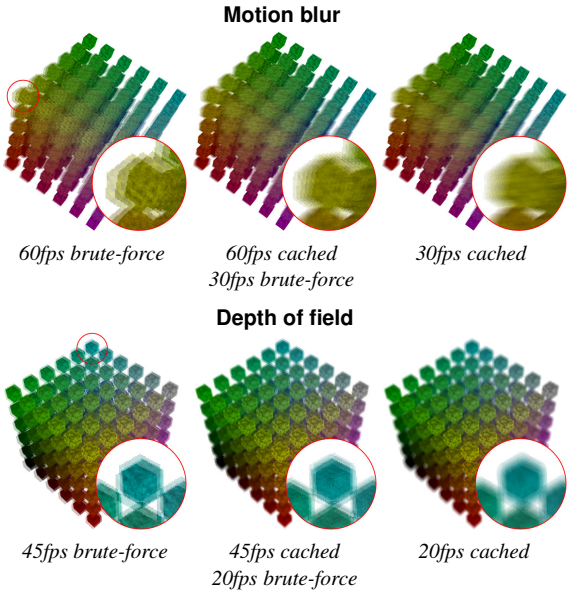


Figure 10: *Equal time/quality comparisons between brute-forced methods for rendering motion blur and depth of field effects and techniques extended to use our caching method.* ***Left:*** *At high frame rates, brute-force methods may undersample camera locations and lead to unconvincing results.* ***Middle:*** *Our caching technique lowers the cost of a single pass, allowing the accumulation of more samples and thus smoother effects at comparable frame rates.* ***Right:*** *Results obtained with cache-based methods at equal frame rates.*

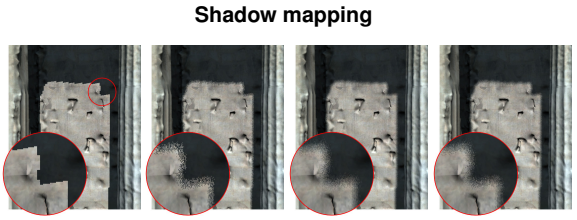


Figure 11: *Our caching strategy can be used to super-sample shadow-map tests. As seen in this close-up of the Parthenon model (a) the limited resolution of the shadow map results in aliasing artifacts along shadow boundaries. (b) Percentage Closest Filtering (PCF) exchanges aliasing for high-frequency noise by averaging the results of several shadow tests. (c) Increasing the number of samples further attenuates the noise, but can become too expensive for interactive applications. (d) Our approach allows amortizing the cost of sampling over several frames to provide improved image quality at higher frame rates.*

calculation and recompute the specular contribution anew during each pass. This example underscores the importance of selecting values to cache that change gradually between frames.

Brute-force stereographic rendering allows 28fps on our system. Caching only the diffuse component improves the frame rate to 39fps and caching the final color results in 44fps. Our method provides a 57% frame rate increase, with negligible loss in visual quality.

Motion blur and depth of field effects can be simulated by combining several images of a scene rendered at slightly different points in time or from nearby viewing angles [HA90]. Their strong spatio-temporal coherence allows reusing expensive shading calculations computed in the first pass during subsequent passes–an idea explored by [CW93] and [HDMS03] in the context of image-based rendering and ray-tracing animations, respectively. Furthermore, averaging together multiple images tends to blur shading errors and extends the use of our technique to values with stronger view-dependent effects.

Figure 10 compares brute-force techniques for rendering motion blur and depth of field effects to results obtained with caching. The model shown has 2.5k triangles and the same shading as Figure 9. Our technique allows rendering this scene approximately twice as fast at equal quality or, conversely, combining twice the number of samples at an equal frame rate.

7.3 Antialiased shadow map boundaries

Shadow maps [Wil78] have become an indispensable tool for displaying shadows at interactive rates. The scene is first rendered from the center of projection of each light source and the contents of the Z-buffer are stored in textures called shadow maps. As the scene is rendered from the observer's point of view, each pixel tests its location within each map and accumulates the contribution of visible sources.

Because the sampling pattern of pixels at the camera are different from the light sources, this simple technique is often plagued by aliasing problems (Figure 11a). One solution is to increase the effective resolution of the shadow map [FFBG01, SD02]. Alternatively, Reeves et al. [RSC87] introduced Percentage Closer Filtering (PCF) as a way to reduce these artifacts by approximating partial shadow coverage with the average over a number of stochastic samples within each pixel (see Figure 11b). In our experiments, PCF typically requires as few as 16 samples to resolve acceptable shadow boundaries (Figure 11c).

Our amortized sampling method described in Section 6 is well suited to optimize PCF. Figure 11d shows the result of generating 4 samples at each frame by randomly rotating a fixed sampling pattern and recursively accumulating these samples in the cache using a weighting factor of $\lambda = 3/5$. This reduces the variance of our estimator by a factor of $1/4$ (Equation 7), providing images of comparable quality to the original method using 16 samples per frame (compare Figures 11c and 11d).

8 Conclusions

We have introduced a simple technique for caching and reusing expensive shading calculations that can improve the performance and quality of many common real-time rendering tasks. Based on reverse reprojection, our method allows consecutive frames to efficiently share shading information, avoids maintaining complex data structures and limits the traffic between the CPU and GPU. We have also provided a set of guidelines for selecting calculations appropriate for reuse and measured the benefit of our method in several real-world applications.

Limitations: Our method is appropriate only for applications with significantly larger per-pixel shading costs than geometry processing costs. It is also important to reuse calculations that exhibit low-frequency light- and view-dependent effects in order to avoid noticeable errors in the shading. Section 4 provides a set of guidelines for identifying appropriate applications.

Future work: We are interested in exploring alternative parameterizations of cached values. Currently we store entries over visible surfaces, but parameterizations designed to expose the symmetry of local reflectance models [Rus98] might allow more aggressive caching of highly directionally-dependent scenes.

Another area of future work involves using our technique to guide automatic per-pixel selection of level-of-detail. Because a side-effect of our technique is a dense and exact motion field, we can estimate the speed of objects and use this information to dynamically select an appropriate level within a set of automatically or manually generated shaders [OKS03, Pel05].

Acknowledgements

The authors wish to thank Rui Wang and David Luebke for generously sharing their subsurface scattering code and the many reviewers for their helpful comments.

References

[AH93] ADELSON S. J., HODGES L. F.: Stereoscopic ray-tracing. *The Visual Computer 10*, 3 (1993), 127–144.

[AH95] ADELSON S. J., HODGES L. F.: Generating exact ray-traced animation frames by reprojection. *IEEE Computer Graphics and Applications 15*, 3 (1995), 43–52.

[AS06] AKELEY K., SU J.: Minimum triangle separation for correct z-buffer occlusion. In *Proc. of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware* (2006), pp. 27–30.

[Bad88] BADT JR. S.: Two algorithms for taking advantage of temporal coherence in ray tracing. *The Visual Computer 4*, 3 (1988), 123–132.

[BDT99] BALA K., DORSEY J., TELLER S.: Radiance interpolants for accelerated bounded-error ray tracing. *ACM Transactions on Graphics 18*, 3 (1999), 213–256.

[BFMZ94] BISHOP G., FUCHS H., MCMILLAN L., ZAGIER E. J. S.: Frameless rendering: Double buffering considered harmful. In *Proc. of ACM SIGGRAPH 94* (1994), ACM Press/ACM SIGGRAPH, pp. 175–176.

[Bly06] BLYTHE D.: The Direct3D® 10 system. *ACM Transactions on Graphics (Proc. of ACM SIGGRAPH 2006) 25*, 3 (2006), 724–734.

[CCC87] COOK R. L., CARPENTER L., CATMULL E.: The REYES image rendering architecture. *Computer Graphics (Proc. of ACM SIGGRAPH 87) 21*, 4 (1987), 95–102.

[Coo86] COOK R. L.: Stochastic sampling in computer graphics. *ACM Transactions on Graphics 5*, 1 (1986), 51–72.

[CW93] CHEN S. E., WILLIAMS L.: View interpolation for image synthesis. In *Proc. of ACM SIGGRAPH 93* (1993), ACM Press/ACM SIGGRAPH, pp. 279–288.

[Dub01] DUBOIS E.: A projection method to generate anaglyph stereo images. In *ICASSP* (2001), vol. 3, IEEE Computer Society Press, pp. 1661–1664.

[DW85] DIPPÉ M. A. Z., WOLD E. H.: Antialiasing through stochastic sampling. *Computer Graphics (Proc. of ACM SIGGRAPH 85) 19*, 3 (1985), 69–78.

[DWS*88] DEERING M., WINNER S., SCHEDIWY B., DUFFY C., HUNT N.: The triangle processor and normal vector shader: a VLSI system for high performance graphics. In *Computer Graphics (Proc. of ACM SIGGRAPH 88)* (1988), ACM Press/ACM SIGGRAPH, pp. 21–30.

[DWWL05] DAYAL A., WOOLLEY C., WATSON B., LUEBKE D.: Adaptive frameless rendering. In *Eurographics Symposium on Rendering* (2005), Rendering Techniques, Springer-Verlag, pp. 265–275.

[FFBG01] FERNANDO R., FERNANDEZ S., BALA K., GREENBERG D. P.: Adaptive shadow maps. In *Proc. of ACM SIGGRAPH 2001* (2001), ACM Press/ACM SIGGRAPH, pp. 387–390.

[HA90] HAEBERLI P., AKELEY K.: The accumulation buffer: hardware support for high-quality rendering. *Computer Graphics (Proc. of ACM SIGGRAPH 90) 24*, 4 (1990), 309–318.

[HDMS03] HAVRAN V., DAMEZ C., MYSZKOWSKI K., SEIDEL H.-P.: An efficient spatio-temporal architecture for animation rendering. In *Eurographics Symposium on Rendering* (2003), Rendering Techniques, Springer-Verlag, pp. 106–117.

[HM91] HECKBERT P., MORETON H.: Interpolation for polygon texture mapping and shading. In *State of the Art in Computer Grpahics: Visualization and Modeling*, Rogers D., Earnshaw R., (Eds.). Springer-Verlag, 1991, pp. 101–111.

[JB02] JENSEN H. W., BUHLER J.: A rapid hierarchical rendering technique for translucent materials. In *Proc. of ACM SIGGRAPH 2002* (2002), ACM Press.

[JMLH01] JENSEN H. W., MARSCHNER S. R., LEVOY M., HANRAHAN P.: A practical model for subsurface light transport. In *Proc. of ACM SIGGRAPH 2001* (2001), ACM Press.

[MB95] MCMILLAN L., BISHOP G.: Head-tracked stereoscopic display using image warping. In *SPIE* (1995), Fisher S., Merritt J., Bolas B., (Eds.), vol. 2049, pp. 21–30.

[MMB97] MARK W. R., MCMILLAN L., BISHOP G.: Post-rendering 3D warping. In *Symposium on Interactive 3D Graphics* (Apr. 1997), pp. 7–16.

[MS95] MACIEL P. W. C., SHIRLEY P.: Visual navigation of large environments using textured clusters. In *SI3D'95* (1995), ACM Press, pp. 95–102.

[NBS06] NEHAB D., BARCZAK J., SANDER P. V.: Triangle order optimization for graphics hardware computation culling. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2006), pp. 207–211.

[NKGR06] NAYAR S. K., KRISHNAN G., GROSSBERG M. D., RASKAR R.: Fast separation of direct and global components of a scene using high frequency illumination. *ACM Transactions on Graphics (Proc. of ACM SIGGRAPH 2006) 25*, 3 (2006), 935–944.

[NRH03] NG R., RAMAMOORTHI R., HANRAHAN P.: All-frequency shadows using non-linear wavelet lighting approximation. In *Proc. of ACM SIGGRAPH 2003* (2003), ACM Press.

[OKS03] OLANO M., KUEHNE B., SIMMONS M.: Automatic shader level of detail. In *Proc. of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware* (2003), Eurographics Association, pp. 7–14.

[Pel05] PELLACINI F.: User-configurable automatic shader simplification. *ACM Transactions on Graphics (Proc. of ACM SIGGRAPH 2005) 24*, 3 (2005), 445–452.

[Per85] PERLIN K.: An image synthesizer. In *Proc. of ACM SIGGRAPH 85* (1985), ACM Press/ACM SIGGRAPH, pp. 287–296.

[RC04] ROBERT C. P., CASELLA G.: *Monte Carlo Statistical Methods*. Springer, 2004.

[RH94] ROHLF J., HELMAN J.: Iris performer: a high performance multiprocessing toolkit for real-time 3d graphics. In *Proc. of ACM SIGGRAPH 94* (1994), ACM Press/ACM SIGGRAPH, pp. 381–394.

[RP94] REGAN M., POSE R.: Priority rendering with a virtual reality address recalculation pipeline. In *Proc. of ACM SIGGRAPH 94* (1994), ACM Press/ACM SIGGRAPH, pp. 155–162.

[RSC87] REEVES W. T., SALESIN D. H., COOK R. L.: Rendering antialiased shadows with depth maps. *Computer Graphics (Proc. of ACM SIGGRAPH 87) 21*, 4 (1987), 283–291.

[Rus98] RUSINKIEWICZ S.: A new change of variables for efficient BRDF representation. In *Eurographics Workshop on Rendering* (1998).

[SD02] STAMMINGER M., DRETTAKIS G.: Perspective shadow maps. *ACM Transactions on Graphics (Proc. of ACM SIGGRAPH 2002) 21*, 3 (2002), 557–563.

[SHSS00] STAMMINGER M., HABER J., SCHIRMACHER H., SEIDEL H.-P.: Walkthroughs with corrective texturing. In *Eurographics Workshop on Rendering* (2000), Rendering Techniques, Springer-Verlag, pp. 377–388.

[SIM05] SANDER P. V., ISIDORO J. R., MITCHELL J. L.: Computation culling with explicit early-z and dynamic flow control. In *GPU Shading and Rendering*. ACM SIGGRAPH Course 37 Notes, 2005, ch. 10.

[SKS02] SLOAN P.-P., KAUTZ J., SNYDER J.: Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. In *Proc. of ACM SIGGRAPH 2002* (2002), ACM Press.

[SS00] SIMMONS M., SÉQUIN C. H.: Tapestry: A dynamic mesh-based display representation for interactive rendering. In *Eurographics Workshop on Rendering* (2000), Rendering Techniques, Springer-Verlag, pp. 329–340.

[TK96] TORBORG J., KAJIYA J. T.: Talisman: commodity realtime 3D graphics for the PC. In *Proc. of ACM SIGGRAPH 96* (1996), ACM Press/ACM SIGGRAPH, pp. 353–363.

[TPWG02] TOLE P., PELLACINI F., WALTER B., GREENBERG D. P.: Interactive global illumination in dynamic scenes. *ACM Transactions on Graphics (Proc. of ACM SIGGRAPH 2002) 21*, 3 (2002), 537–546.

[WDP99] WALTER B., DRETTAKIS G., PARKER S.: Interactive rendering using the render cache. In *Eurographics Workshop on Rendering* (1999), Rendering Techniques, Springer-Verlag, pp. 19–30.

[Wil78] WILLIAMS L.: Casting curved shadows on curved surfaces. *Computer Graphics (Proc. of ACM SIGGRAPH 78) 12*, 3 (1978), 270–274.

[WS99] WARD G., SIMMONS M.: The holodeck ray cache: an interactive rendering system for global illumination in nondiffuse environments. *ACM Transactions on Graphics 18*, 4 (1999), 361–368.

[WTL05] WANG R., TRAN J., LUEBKE D.: All-frequency interactive relighting of translucent objects with single and multiple scattering. *ACM Transactions on Graphics (Proc. of ACM SIGGRAPH 2005) 24*, 3 (2005), 1050–1053.

[ZWL05] ZHU T., WANG R., LUEBKE D.: A GPU accelerated render cache. In *Pacific Graphics (short paper)* (2005).