Author preprint. To appear in ACM TOG 35(6).

Texture Space Caching and Reconstruction for Ray Tracing

Jacob Munkberg* Jon Hasselgren Petrik Clarberg Magnus Andersson Tomas Akenine-Möller

Intel Corporation

Figure 1: *By caching and reconstructing shading in texture space, we can significantly reduce noise in Monte Carlo ray tracing. In this example, we show the Barbarian model with environment map illumination and depth of field. Our system reduces shading noise through texture space caching and filtering. Additionally, we decouple primary visibility and denoise it separately using layered reconstruction.*

Abstract

We present a *texture space* caching and reconstruction system for Monte Carlo ray tracing. Our system gathers and filters shading on-demand, including querying secondary rays, directly within a filter footprint around the current shading point. We shade on local grids in texture space with primary visibility decoupled from shading. Unique filters can be applied per material, where any terms of the shader can be chosen to be included in each kernel. This is a departure from recent screen space image reconstruction techniques, which typically use a single, complex kernel with a set of large auxiliary guide images as input. We show a number of high-performance use cases for our system, including interactive denoising of Monte Carlo ray tracing with motion/defocus blur, spatial and temporal shading reuse, cached product importance sampling, and filters based on linear regression in texture space.

Keywords: ray tracing, reconstruction, shading reuse, global illumination, real-time rendering

Concepts: •**Computing methodologies** → **Ray tracing; Graphics processors;**

1 Introduction

Over the past five years or so, novel reconstruction and filtering techniques have shown great promise in assisting both stochastic rasterization and path tracing to substantially reduce total image generation time. Very briefly, a sparsely sampled image together with auxiliary data, e.g., surface normals and depth, are fed into a reconstruction algorithm that attempts to reduce the noise in the input image. The importance of all this research can be seen in that screen space reconstruction techniques have recently gained traction in the offline rendering community. However, most of these techniques take seconds or even minutes per frame and require several full-screen auxiliary buffers to guide the reconstruction process. In addition, due to an underlying assumption of noise-free guide images, these approaches struggle when camera effects, such as defocus and motion blur, are applied. For temporal consistency, the initial sampling level needs to be substantially increased compared to reconstructing a single frame. Otherwise, temporal, low frequency artifacts remain.

While systems for accelerating visibility queries, such as Embree [Wald et al. 2014] and OptiX [Parker et al. 2010] help reduce the total rendering time, modern workloads have complex shaders implying that only a fraction of the total time is spent tracing rays [Eisenacher et al. 2013]. Hence, efficient shading reuse for ray tracing is highly desirable.

We present a *system* for shading reuse and reconstruction. The key difference from other reconstruction methods is that ours operates directly in texture space in order to alleviate some of the previous limitations. Our system filters and caches shading on the surfaces of objects in the scene, which enables temporal and spatial shading reuse. Moreover, the local reconstruction can be material-specific, so knowledge of the current material can be exploited to guide reconstruction. Filter kernels are applied in texture space and do not need to handle inter-object visibility, nor do they require (potentially noisy) screen space guide images. As a direct consequence, we can often use very simple filters. Primary visibility is evaluated separately and may reuse the filtered texture space shading at an object hit. We show how efficient primary visibility filters are applied on top of the filtered texture space shading to obtain noise-free defocus and motion blur.

Our goal is a flexible system for generating noise-free ray traced images at modest ray budgets. We shade on quantized grids in texture space, and the results, although close to reference renderings, are thus generally neither unbiased nor consistent. However, we believe this may be an important trade-off to reach low noise levels in future real-time ray tracing scenarios. In Section 6, we explore a variety of important applications of our system, including stereo/VR rendering, interactive pre-visualization, spatial and temporal shading reuse, cached product importance sampling, and a comparison against state-of-the-art screen space reconstruction.

*e-mail:jacob.munkberg@intel.com

SA '16 Technical Papers,, December 05 - 08, 2016, , Macao
ISBN: 978-1-4503-4514-9/16/12
DOI: http://dx.doi.org/10.1145/2980179.2982407

2 Previous Work

Reconstruction There is a vast body of recent work in Monte Carlo reconstruction algorithms in the research community. Since 2010, there has been over 20 ACM TOG papers in this area and we refer to the recent survey by Zwicker et al. [2015] for an extensive overview.

A common trait in many recent screen space methods is to use a set of *guide* images, e.g., normals, depths, distance to closest occluders, etc., which are used to locally control the filter extents in order to preserve sharp edges. Cross-bilateral filters based on such guides can be evaluated in real time [Gastal and Oliveira 2012]. The selection of guide images and their respective weights are highly scene-dependent. For example, recent work applied machine learning to train a large set of weights (for 7 primary and 36 secondary features) [Kalantari et al. 2015]. In the presence of defocus and motion blur, the guide images are noisy, which limit their applicability. In contrast, we apply filters in the texture space of each object and can extract noise-free object-specific guides by querying the intersected object without the need of full-screen buffers. Furthermore, once a texture space region is filtered, this information does not have to be persistent in memory, which enables reconstruction within a small memory footprint.

To filter renderings with sparsely sampled global illumination and defocus blur, a recent technique [Bauszat et al. 2015] reconstructs shading per sample before approximating the lens integral over each pixel. Both steps use adaptive manifolds [Gastal and Oliveira 2012] based on guide images stored at sample rate. We similarly separate the reconstruction of shading and visibility, but filter the former on-demand in texture space using material-specific filters. This avoids the significant cost of storing multiple sample rate guide images. As a result of treating visibility separately, we can use any domain-specific reconstruction method for primary visibility, including motion and defocus blur, for example, Lehtinen et al.'s algorithm [2011] or recent filters based on layered scene representations [Munkberg et al. 2014; Vaidyanathan et al. 2015].

These layered filters are inspired by recent work in frequency analysis, which provides bandwidth estimates for specific effects, including soft shadows, indirect illumination, depth of field, and motion blur [Egan et al. 2009; Egan et al. 2011b; Egan et al. 2011a; Belcour et al. 2013]. The main challenge is to formulate efficient implementations of these high-dimensional filters. With intermediate 4D storage, a fast, approximate formulation of a sheared 4D filter was recently presented [Yan et al. 2015]. Interactive reconstruction has been formulated using coarser, axis-aligned filters [Mehta et al. 2012; Mehta et al. 2013; Mehta et al. 2014; Mehta et al. 2015]. An important advantage of shading in texture space is that we can apply unique filters per material and can directly apply these bandwidth estimates in texture space kernels.

Application of separate filters for different effects has previously been used in ray path decompositions [Zimmer et al. 2015], where reflectance and irradiance are stored in separate buffers at each bounce. Unique filters can be tailored for each effect, e.g., one filter for direct and another for indirect illumination. Our system gives the additional freedom to fine-tune this separation per material and avoids the problem of primary visibility noise.

Shading reuse Path space filtering [Keller et al. 2014] locally stores a vertex per path, typically the first non-specular hit. When evaluating shading, a kNN-search is performed in a radius around the path vertex, gathering radiance values from nearby vertices. Shading is evaluated as a weighted sum of the colors of the vertices within the search radius, which bears some resemblance to photon mapping [Jensen 2001]. Hou and Zhou [2010] use a simpler 2D kNN-search for micropolygon ray tracing for motion and defocus blur. Meyer and Anderson [2006] use statistical filtering for spatial and temporal denoising, and suggest storing indirect illumination at the same object space positions in each frame, either in texture maps or in a point cloud. Our approach is somewhat similar to these methods, but we explicitly store shading values in texture space to avoid the expensive kNN-search and corresponding spatial data structure.

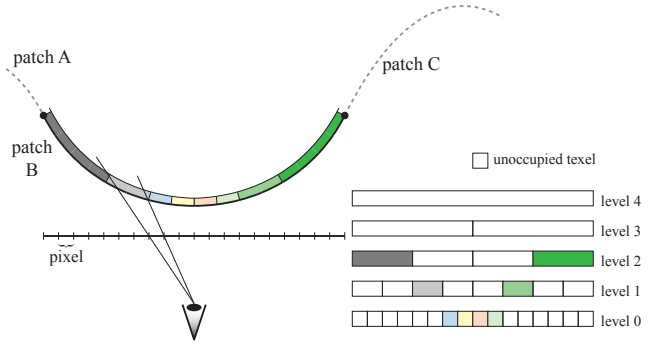


Figure 2: *Our per-face texture space shading structure visualized using curves and one-dimensional mipmap textures. The footprint of a pixel on a surface determines the mip level of the shading grid.*

Radiosity rendering methods were likely first to store shading in textures. Heckbert [1990] used per-face textures to perform density estimation for diffuse radiosity and recomputed specular effects separately. Bastos et al. [1997] improved radiosity reconstruction in texture space using bicubic filtering. Texture atlases can also be used for radiosity computations [Ray et al. 2003]. Luksch et al. [2013] precompute light maps using clustering of virtual point lights. The idea of decoupling shading from visibility is also common in Z-buffer techniques, and is at the core of the Reyes system [Cook et al. 1987] where shading is performed on micropolygon vertices. More recently, similar decoupling has been applied also to multi-resolution and micropolygon ray tracing [Djeu et al. 2011; Hou and Zhou 2011]. Similarly, we shade at a per-face grid in texture space. In contrast, we apply reconstruction filters directly on the grids and do not rely on a tessellation matching our shading resolution.

Our texture space caching system is inspired by decoupled shading caches [Burns et al. 2010; Ragan-Kelley et al. 2011; Gribel et al. 2011] and recent techniques for evaluating and caching shading in texture space on patches [Clarberg et al. 2014] and triangle meshes [Andersson et al. 2014; Hillesland and Yang 2016]. Although those methods were designed for rasterization pipelines, we apply a similar concept in a ray tracing setting where shading is triggered on demand when a ray hits an object. We extend these approaches with an additional level of caching to reconstruct and reuse filtered shading values on-demand.

3 Texture Space Shading and Caching

In this section, we give an introduction to texture space shading, before introducing our system and contributions in Section 4. The concept of evaluating and storing shading values on the surfaces of objects, i.e., in texture space, has been around for quite some time. Light mapping is an old technique for storing statically computed shading in textures, which are then applied as regular texture maps in real-time rendering of, for example, games. To handle dynamically changing environments, shading has to be computed

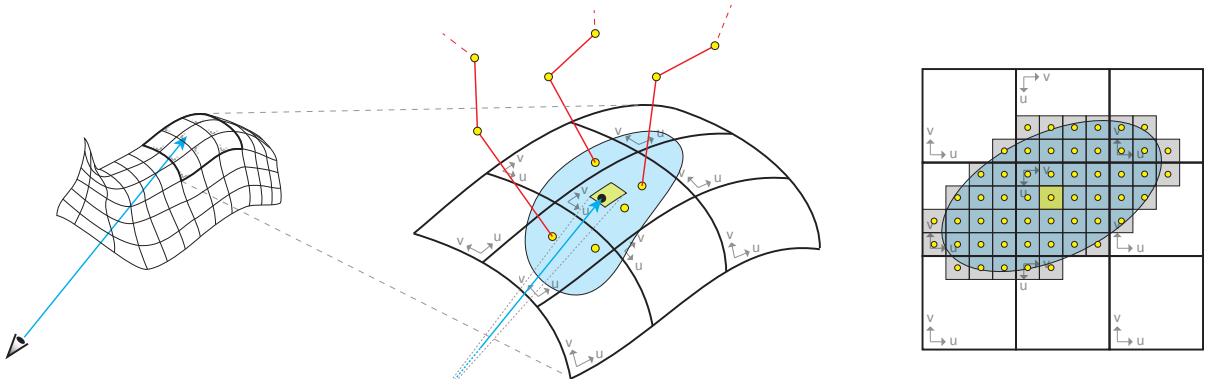


Figure 3: *Our system filters and caches shading on grids in texture space by utilizing the parameterization provided by Ptex. At the hit point (black dot), an appropriate shading grid resolution is computed based on the ray differentials. If the nearest grid cell is not already shaded, a filter footprint is determined (blue ellipse), and all not previously shaded cells (gray squares) within the footprint are queued for shading. Shading typically involves sending one or more secondary rays per cell (here one ray per cell, marked by the yellow dots), and accumulating the result. Once all required cells are available, the reconstruction filter is evaluated in texture space and the filtered color is stored at the grid cell that triggered the operation (yellow square). The system differentiates between shaded but not already filtered cells (called samples), and filtered cells (texels), keeping both types cached only for as long as necessary and/or dictated by the cache capacity.*

and cached on the fly. The core motivation is to allow costly shading computations to be amortized by reusing shading spatially and temporally. Several recent papers have shown how to do this in rasterization-based systems [Burns et al. 2010; Clarberg et al. 2014; Hillesland and Yang 2016].

Texture space shading requires a two-dimensional $\mathbf{u} = (u, v)$ parameterization of the surfaces. Tessellated geometry provides a natural parameterization as each quadrilateral patch is a $[0, 1]^2$ domain, and Ptex [Burley and Lacewell 2008] provides the necessary connectivity between patches. For non-tessellated triangle meshes, existing texture parameterizations created by artists or automated tools can be reused [Hillesland and Yang 2016]. In theory, our system is thus not limited to any particular type of assets, although we focus on the rendering of subdivision surfaces in this paper.

For a hit point $\mathbf{u}$ on a surface, the appropriate shading resolution or texture *level-of-detail* (LOD) is determined based on the screen space derivatives $\partial\mathbf{u}/\partial x$ and $\partial\mathbf{u}/\partial y$ [Gribel et al. 2011; Clarberg et al. 2014; Hillesland and Yang 2016]. In a rasterizer, these derivatives are given by finite differences, whereas the analogy in ray tracing is ray differentials [Igehy 1999]. The goal is to choose the resolution so that a texel is approximately pixel-sized. This can then be biased towards higher and lower shading resolutions, in order to trade performance versus quality. Previous work all use slightly different variations, and we compute the level-of-detail l as

$$l = \log_2 \min\left(w\|\frac{\partial\mathbf{u}}{\partial x}\|, h\|\frac{\partial\mathbf{u}}{\partial y}\|\right), \tag{1}$$

where (w, h) is the base resolution of the texture, and $l = 0$ is the base resolution and $l > 0$ smaller mip levels. Note that in practice, textures are normally arranged in a mipmap hierarchy with power-of-two resolutions, and often $w = h$. Equation 1 biases the shading towards higher resolution when the anisotropy is large, which avoids over-blurring the shading when viewing a surface at an acute angle. Figure 2 shows an example.

Once the level-of-detail has been determined, $\mathbf{u}$ is quantized to the nearest texel $(\hat{u}, \hat{v})$. We do this conservatively by selecting a mip level $i = \lfloor l \rfloor$ (again biasing towards higher resolution) and computing quantized normalized texel coordinates as

$$(\hat{u}, \hat{v}) = \left(\frac{\lfloor u\, w_i \rfloor + 0.5}{w_i}, \frac{\lfloor v\, h_i \rfloor + 0.5}{h_i}\right). \tag{2}$$

It should be noted that if the final shading is filtered by a texture filter, e.g., trilinear, bicubic, or anisotropic, multiple texels have to be selected, one per filter tap, rather than a single nearest texel.

The shading computed for a texel $(\hat{u}, \hat{v})$ can either be cached in a sparsely populated mipmap hierarchy [Gribel et al. 2011; Hillesland and Yang 2016] or in a fixed-size cache [Clarberg et al. 2014]. We have implementations of both strategies, as discussed in Section 5. In the latter case, the cache key is computed as

$$\text{key} = \text{hash}(id, i, \hat{u}, \hat{v}), \tag{3}$$

where id is a unique patch identifier, and i is the mip level. The hash function can be any standard hash, and we use a simple scanline texel ordering (see the supplemental material).

In the next section, we build on these fundamentals to introduce our novel system for caching, filtering, and reconstructing shading in a Monte Carlo ray tracer. We show that by moving state-of-the-art filters to texture space, many unique benefits are made possible. In addition to this, we believe our work is the first to analyze how well texture space shading and caching works in a ray tracing context.

4 System

We have extended the publicly available Embree path tracer [Wald et al. 2014; Benthin et al. 2015] with texture space caching and reconstruction. Our system allocates two different shading caches, each parameterized in the local uv-coordinates of the face of a subdivision mesh. These two caches temporarily hold either shading *samples*, or filtered shading colors, which we denote *texels*. Shading samples are the inputs when reconstructing texels, and may hold local attributes needed by the reconstruction filter, e.g., normals and colors. The sample and texel caches are populated on demand as part of shading and filtering. An overview of our system can be seen in Figure 3.

Our system generates a full image using the following steps:

1. Trace a set of camera rays. At each intersection, quantize to an appropriate shading grid resolution of the intersecting face and retrieve the nearest texel through the texel cache, as described in Section 3.

2. If the texel is not populated, compute the filter footprint at this intersection point. The footprint may be a function of the ray's screen space extents, i.e., ray differentials, the material at the hit point, etc.

3. For each shading sample under the footprint, check if it is already populated in the sample cache, and dispatch non-shaded samples for shading.

4. Once all samples in the footprint are shaded, apply a reconstruction filter and store the filtered color at the nearest texel.

In step 3, when shading a sample, we query the surface to obtain the shading position and normal at the quantized $(\hat{u}, \hat{v})$ position (using the Embree API call `rtcInterpolate`). In the current implementation, this function re-evaluates the subdivision surface at the chosen point. Shading then proceeds as usual by evaluating material properties, sampling texture maps, and sending secondary rays etc., depending on how the shader is written.

In steps 2 and 4, any reconstruction filter that operates over a 2D domain can be applied, and we will show examples of several state-of-the-art filters. To filter over adjacent faces, we gather samples in a local footprint with the patch connectivity information provided by Ptex [Burley and Lacewell 2008]. Since we are filtering on the surface of objects, all samples under the footprint are known to belong to the same object and the filter can exploit material-specific parameters. This a major strength compared to screen space filters, which can pull in samples from any surface.

Another strength of our system is that it builds on the same shading caching mechanisms as have been explored for rasterization-based systems in the past, and one can imagine a hybrid renderer with rasterization of primary visibility. Compared to previous work, we add a second level of caching, as the evaluation of a texel can itself trigger shading and caching of one or more samples. The samples are lazily populated in their own shading cache, and can later be reused when reconstruction filters are applied at neighboring texels. This gives our system a high amount of shading reuse, and makes it possible to apply reconstruction filters on demand as part of shading.

To summarize, there are two main benefits of our system: 1) shading can be reused between rays, and the shading rate is decoupled from the visibility rate and tessellation, and 2) material-specific reconstruction kernels are applied on-demand in texture space, without storing feature buffers to disk. In Section 6, we will evaluate these advantages in detail.

Example: Cached and Filtered Indirect Illumination It is easy to adapt a renderer to use texture space caching. Replace the shader term that you want to filter with a lookup in the texel cache, and implement the corresponding callback functions for missing texels and samples. These callbacks can be seen as the programmable filter shader and surface shader of the material, similar to how many current renderers support programmable surface shaders. Below, we demonstrate our system on a simple path tracing example, where the indirect illumination is cached and filtered in texture space.

```
// main render loop in standard path tracer
L = ComputeDirectLighting(...)
L += ComputeIndirect(...)
// our main render loop
L = ComputeDirectLighting(...)
L += TexelCallback(ray, ddx, ddy, filterRadius)
```

The render loop is divided in direct and indirect components. We lazily evaluate, filter, and cache indirect lighting for each mesh face.

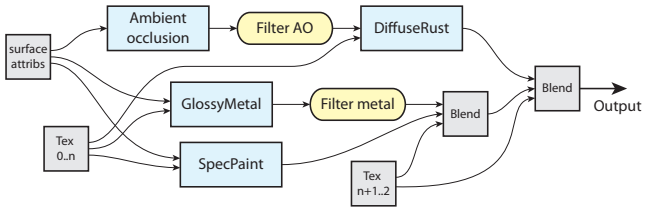


Figure 4: *Example shader graph for an artist-created material, in this case a rusty, painted metal. With our system, reconstruction filters can be applied at any point in the graph to denoise shader components. These exploit locality by operating in texture space, and samples are only kept locally as long as needed.*

The `TexelCallback` function first quantizes the intersection point to find the nearest texel at the chosen mip. Next, the texel cache is queried at the nearest texel, and if not in cache, shading is lazily triggered for all samples within the filter footprint. Finally, the shaded samples are weighted using a filter kernel of choice, and the filtered result is stored in the current texel. Subsequent requests to the same location will reuse the cached texel.

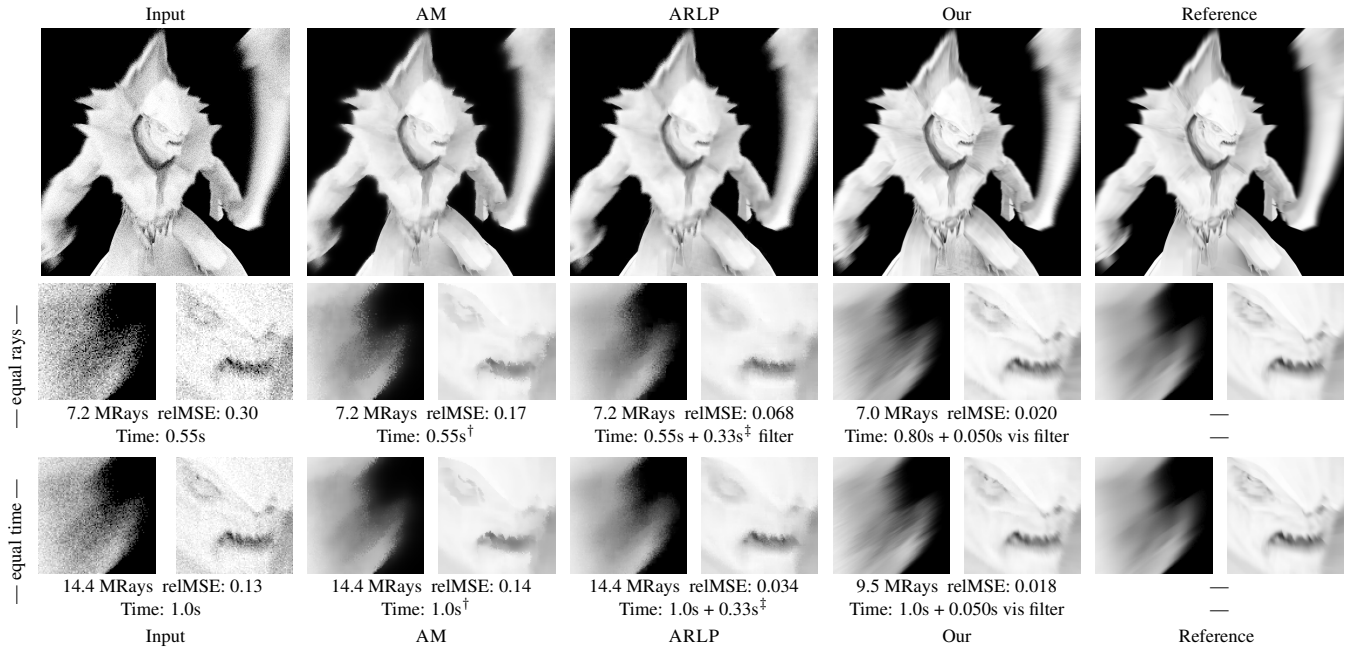
```
float3 TexelCallback(ray, ddx, ddy, filterRadius):
   faceID = ray.faceID;
   mip = ComputeMip(faceID, ddx, ddy)
   uv = ClosestTexel(faceID, ray.u, ray.v, mip)
   e = texelCache->GetTexel(faceID, uv, mip)
   if !e.filtered: // gather taps
      for s in Footprint(mip, filterRadius):
         fs = SampleCallback(s.faceID, s.uv, mip)
         w = FilterWeight(...)
         cacc += fs.color * w
         wacc += w
      e.color = cacc / wacc
      e.filtered = true
      texelCache->setTexel(faceID, uv, mip, e)
   return e.color
```

The `SampleCallback` function queries the cache, and if no shaded sample is found at the requested position, the indirect lighting is evaluated. Each sample may contain additional attributes needed by the filter kernel, e.g., surface normals.

```
sample SampleCallback(faceID, uv, mip):
   s = sampleCache->GetSample(faceID, uv, mip)
   if !s.shaded:
      attribs = GetAttribsAt(faceID, uv, mip)
      s.color = ComputeIndirect(...)
      s.shaded = true
      sampleCache->setSample(faceID, uv, mip, s)
   return s
```

For clarity in this example, the cache requests are explicitly written out and the reconstruction filter is a simple weighted average. In general, the user is free to choose any texture space reconstruction filter, and can freely decide which shader terms to cache. It is also possible to hide the caching logic from the shader author.

Per-Material Filters We have the freedom to apply unique texture space reconstruction filters for each material or shading term, as illustrated by the example in Figure 4. This mechanism is more flexible than the approach commonly taken in offline rendering, i.e., dump a large set of intermediate render targets, filter each image channel, and composite them together. We obtain the same effect as part of shading, and avoid the additional memory traffic. Operating locally in texture space, our filters can be highly specialized and take different per-material inputs. Doing this with screen space fil-



† filtering time excluded. ‡ filtering time from our multi-threaded CPU implementation. A GPU implementation may reduce this to about 50%.

Figure 5: *Comparison with motion blur and ambient occlusion. All images were rendered at 768×768 pixels, where the first row of insets (equal rays) were rendered with approximately 7 MRays and the second row (equal time) were rendered in approximately one second. Due to motion blur,* AM *and* ARLP *fail to faithfully reconstruct noise-free visibility at low sample rates, which is revealed by relMSE. Note that this is true for the equal time renderings, even if filtering times for* AM *and* ARLP *are excluded. Please zoom in to clearly see the differences.*

ters would quickly lead to a combinatorial explosion, as there may be thousands of shaders in a scene. For artistic control it is desirable to store and composite different image channels. In that case, we can store already filtered channels, but do not have to dump all intermediate data.

Furthermore, we can exploit knowledge of the material filter when gathering samples. For example, if a filter discards taps with a normal difference over a threshold, we do not need to trigger shading in locally back-facing regions. As a consequence of filtering locally, our filters can often be made much simpler than screen space filters, which have to deal with visibility discontinuities/occlusions and a mix of different materials. This is a benefit both in terms of development complexity and execution time. In addition, our kernels directly query the needed surface properties and do not rely on guide images that may be corrupted by noisy primary visibility.

Primary Visibility Filter When using motion blur or depth of field, we may optionally apply a filter to reduce visibility noise. Examples include the TLFR algorithm by Lehtinen et al. [2011] and filters based on layered representations [Vaidyanathan et al. 2015; Munkberg et al. 2014]. This is an attractive feature of our system, i.e., that we can first apply a material-specific filter in texture space to reduce shading noise and then separately filter visibility.

5 Implementation

We have implemented our framework in C++. The code uses the Embree ray tracing kernels [Wald et al. 2014] and Ptex [Burley and Lacewell 2008] for per-face textures on subdivision meshes. Our code is multi-threaded using OpenMP.

Our filters are executed on-demand as part of shading by weighting samples in a local region on the intersected object. Currently, filtering is dispatched for individual texels, so we directly evaluate an $O(n^2)$ kernel per texel, where n is the width of the filter. Alternatively, one could dispatch filters for all texels in a patch and exploit common filter optimizations over a region, using, e.g., separable filters. However, this may restrict the choice of kernels and increases buffering requirements. Since the kernels in texture space can often be made quite simple and of local extent, we have found the flexibility/performance trade-off of directly evaluating simpler kernels to be favorable, although this is an area for future investigation.

As mentioned earlier, the shading caches for texels and samples, store values directly at the accessed mip levels of the shading grids. Cache lookups are performed very frequently and must be efficient. We therefore use simple direct mapped caches, rather than more complex replacement strategies, and the cache is designed to minimize thread synchronization and potential stalls. When a filter is evaluated, we first iterate through the footprint and determine if each sample is either uninitialized, in process by another thread, or shaded. The current thread will shade uninitialized samples, and defer samples in process by another thread. The deferred samples are only revisited after the entire footprint has been processed, by which time they are likely finished by the other threads. This ensures that threads processing overlapping regions will not run in lock-step, and efficiently limits thread synchronization to once per filter kernel. In the current system, performance is largely limited by the performance of the Ptex library and ray tracing.

In scenes with motion and defocus blur, we apply an open source filter for fast layered 5D reconstruction [Hasselgren et al. 2015] as a final step. This filter runs on the GPU with full-screen color, depth, and motion vector buffers as input, while the rest of the system runs asynchronously on the CPU. The uploading of these buffers to the GPU currently consumes tens of milliseconds per frame, but these memory transfers can easily be overlapped by the ray tracing and shading of the next frame in interactive previews.

6 Results

Next, we show a set of use cases for our system. After that, we investigate the scaling properties and performance of our algorithm.

There has been a vast amount of work in Monte Carlo reconstruction algorithms, which makes an exhaustive comparison difficult. One of the most promising approaches is the work by Moon et al. [2014; 2015] and we refer to their extensive comparisons for an overview of expected image quality and run times for recent screen space filters. When appropriate, we primarily compare to *adaptive rendering with linear predictions* (ARLP) [Moon et al. 2015] and to *adaptive manifolds* (AM) [Gastal and Oliveira 2012]. ARLP has proven to be very competitive and supports both an advanced filter and adaptive sampling. AM is included since it has one of the fastest implementations of screen space bilateral filtering. Both methods are run on the CPU with multi-threaded code due to the lack of publicly available GPU implementations. We have optimized the parameters for both AM and ARLP as best as possible by trying many different variations. Further comparisons against some other screen-space filters are included in Section 6.3.

Errors are reported in *relative mean-square error* (relMSE) [Rousselle et al. 2011], which is generated by first computing the sum of $(\mathrm{img} - \mathrm{ref})^2/(\mathrm{grayscale}(\mathrm{ref})^2 + \epsilon)$ over the color channels in each pixel. The relMSE is then the average of those values over all pixels in the image, where we have used $\epsilon = 0.001$. The input data to each reconstruction algorithm comes from path tracing with low discrepancy samples.

All measurements are done on a system with two Intel Xeon E5-2680 v2 CPUs, each having 10 cores and 20 threads. The clock frequency is 2.8 GHz with a turbo frequency of 3.6 GHz and 32 GB of memory. The system also has an NVIDIA GTX 980.

6.1 Use Cases

Motion & Defocus Blur with Noisy Illumination We explore how much we can smooth out noisy ray-traced direct illumination and ambient occlusion with defocus and motion blur by applying reconstruction at all appropriate locations of the rendering process. Figure 5 shows an animated character rendered with ambient occlusion and motion blur, and Figure 1 shows the character with defocus blur and environment lighting. Our texture space filter is trivial; we average samples with similar normals in a 13×13 window. More precisely, if θ is the angle between the sample normals, we include samples with $\cos\theta > 0.9$. Finally, we also apply a layered screen space primary visibility filter [Hasselgren et al. 2015].

We reuse shading over the shutter time and over the lens, which is a common approximation even in production rendering (e.g., Pixar's PRMan). Furthermore, the cache also enables spatial reuse. Although biased, this approach is useful for interactive preview. For AM, we have used $\sigma_s = 6.0$ and $\sigma_t = 0.14$, and for ARLP, we found relMSE to be the lowest using non-adaptive sampling at these sample rates (e.g., one pass with eight primary rays was better than two passes of adaptive sampling with an average of four primary rays in each). The ray traced input images, the AM images, and the ARLP images were rendered using 8 primary rays and 1 ambient occlusion ray per primary ray for the first row of insets. This was changed to 16/1 in the second row (equal time). Neither AM nor ARLP can reconstruct motion blur with high quality at lower sampling rates due to significant visibility noise, which is also present in the guide images, so additional primary rays proved most beneficial. In addition, AM overly smooths out the ambient occlusion noise, which can be seen in the lower row of Figure 5, where the input image has lower relMSE. At much larger number of rays, ARLP converges nicely using their adaptive sampling.

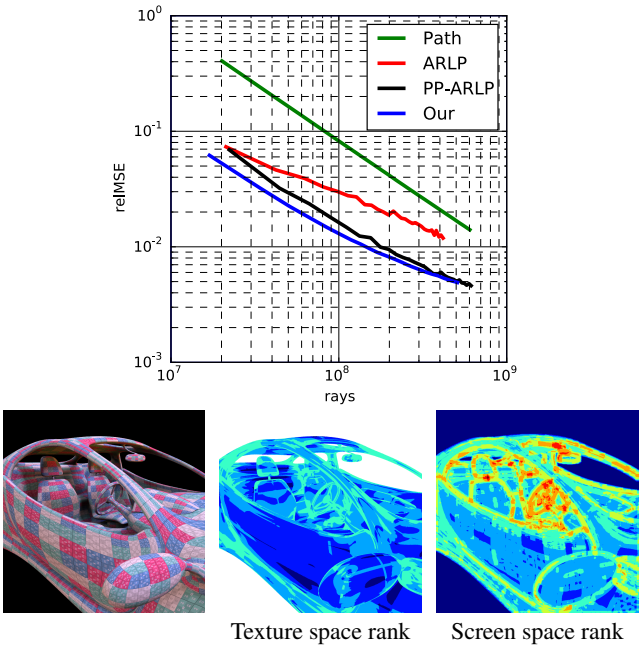


Figure 6: ***Top:*** *convergence plot comparing per-pixel linear regression in screen space (*PP-ARLP*), sparse linear regression in screen space (*ARLP*), and our linear regression in texture space. The screen space approaches use two passes with half of the ray budget each, where in the second pass, rays are adaptively distributed where needed, as suggested by Moon et al. Our texture space linear regression does not use adaptive sampling. For this scene, texture space linear regression has lower error than the screen space variants. We fixed the primary rays' screen space xy-offsets to 16 predetermined positions to isolate shading quality from primary visibility.* ***Bottom:*** *the rank of the local feature space, where the heatmap scale goes from 2 (blue) to 9 (red). Note that the texture space rank is lower overall. We used 64 spp and computed rank for each texel/pixel for these visualizations.*

For AM, we exclude filtering time altogether since the authors report GPU reconstruction times of as little as 1–7 ms at the resolutions we use. We report filtering times for ARLP using our multi-threaded CPU implementation. However, an optimized GPU implementation could likely reduce this further (Moon et al. report GPU reconstruction times of about 0.4s for an 1k×1k image). For fairness, the filtering time is separated for ARLP in the equal time comparisons, while the numbers for our algorithm include everything. Since our relMSE numbers are still lower, we claim better performance at these sampling rates. These results show that a very simple texture space filter is competitive to much more advanced filters in screen space, and the possibility to apply a separate visibility filter is an important feature of our system.

Linear Regression in Texture Space As a proof-of-concept of advanced reconstruction in texture space, we have applied the linear prediction filter with corresponding error estimate (ARLP) [Moon et al. 2015]. We use a linear model around the center texel c as

$$f(\mathbf{x}_i) \approx f(\mathbf{x}_c) + \nabla f(\mathbf{x}_c)^T(\mathbf{x}_i - \mathbf{x}_c). \quad (4)$$

The feature vector $\mathbf{x}_i$ consists of the quantized patch coordinates and surface normal. Unlike ARLP, we do not include depth or texture as features. Depth is mostly useful to handle visibility edges, which is not relevant in texture space. The surface texture can be

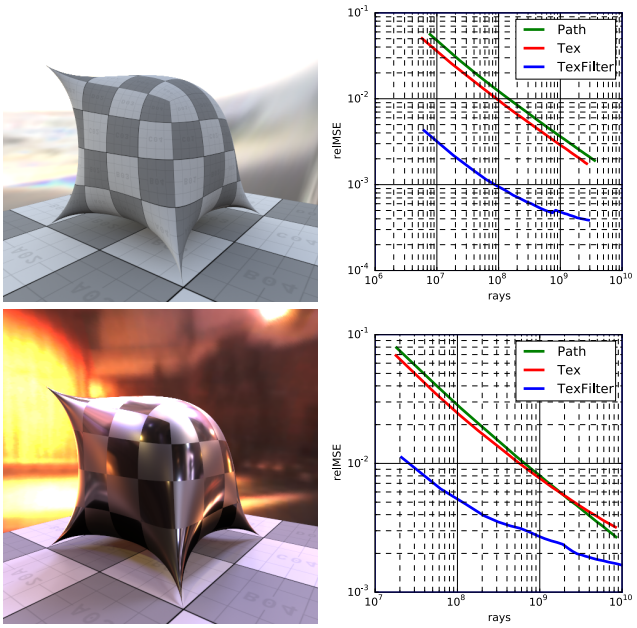


Figure 7: *We compare path tracing (Path), texture space shading without reconstruction filter (Tex), and texture space shading with linear regression (TexFilter). We fixed the primary rays' screen space xy-offsets to 16 predetermined positions to isolate shading quality from primary visibility.* ***Upper row:*** *a textured diffuse material and low-frequency environment map. By reusing shading computations in texture space, the error is lower at equal number of rays. With linear regression in texture space, the error is drastically reduced.* ***Lower row:*** *high frequency environment map and a GGX material with varying roughness. There is less shading reuse potential due to high frequency lighting and material properties. Texture space linear regression efficiently reduces the error.*

directly queried at the quantized patch coordinate $(\hat{u}, \hat{v})$ and does not need to be included in the reconstruction, i.e., we have a five-dimensional feature space, instead of the nine-dimensional feature space of ARLP. Furthermore, note that the feature normal is noise-free, as we can directly query it from the surface at $(\hat{u}, \hat{v})$. We gather all samples within a 17×17 texel footprint, and perform a truncated SVD (TSVD) [Moon et al. 2014] to construct a reduced feature space. Next, we create a set of linear models, each using $(2k+1)^2$ texels, $k \in 0, 1, \ldots, 8$, i.e., rings of radius k around c. We follow Moon et al. [2015] and compute the linear models using recursive least squares with corresponding error estimation. The linear model with smallest estimated error is chosen, and its least-square approximation $\hat{f}(\mathbf{x}_c)$ is the filtered color of texel c.

In contrast to ARLP, where linear models are adaptively placed in screen space, we perform linear regression at each texel. An adaptive approach in texel space is possible, but requires keeping more local data cached. We leave that for future work.

The complexity of computing all linear models using recursive least squares in a footprint with N texels is $\mathcal{O}(Nd^2)$, where d is the rank of the reduced feature space [Moon et al. 2015]. Linear models in texture space can be evaluated faster than the screen space equivalent due to the lower dimensionality of the feature space. If d represents the rank of the feature space, then we have $d \leq 5$ compared to ARLP with $d \leq 9$. In Figure 6, we show how the rank varies in a simple test scene, and show a convergence plot of texture and screen space linear regression.

In Figure 7, we show two versions of a scene with environment map illumination. One purely diffuse with a low-frequency environment map, the second with high frequency lighting and a microfacet GGX-material with spatially varying roughness. As shown in the convergence plots, by moving shading to texture space, the error is slightly lower, as many secondary rays can be shared. Combined with texture space linear regression, the error is drastically reduced. The texture space shading resolution governs the amount of shading reuse. Increased resolution means higher fidelity but less reuse. In this example, at very large number of rays, the chosen shading resolution in texture space caps the convergence rate. For the second (microfacet) example, we include a material-specific roughness parameter as an additional scalar feature to the linear regression. This is a trivial extension in texture space, but is not straightforward in screen space, given a scene with multiple materials.

We have modified Moon et al.'s error estimation computation slightly when estimating the error for a single texel (i.e., $k = 0$) to favor larger filter kernels at low sample rates. They use the sample variance of the current texel. We use a 3×3 window, and if n is the number of samples per texel, we compute a blend factor $\beta = \texttt{clamp}(1.0 - n/256, 0, 1)$, and multiply the per-texel variances with β for all but the center tap when accumulating the variances. The accumulated variance is then normalized with $1 + 8\beta$, i.e., the sum of the weights. For $n \geq 256$, $\beta = 0$ and only the variance of the current texel is included in the error estimate.

Cached Product Importance Sampling Consider direct illumination, where the outgoing radiance L_o, in a direction ω_o, is given by the integral over incoming directions, i.e.,

$$L_o(\omega_o) = \int L(\omega) B(\omega_o, \omega) V(\omega) d\omega. \tag{5}$$

L is the illumination from an environment map, B the reflectance function, i.e., the BSDF times the cosine term, and V denotes visibility. Product importance sampling techniques [Clarberg et al. 2005; Clarberg and Akenine-Möller 2008] explicitly derive an importance function of the reflectance function times the lighting $(B \times L)$, which often results in lower variance than multiple importance sampling [Veach 1998] for cases where both B and L are complex. The main drawback is the cost of building product importance sampling records (PSR), where a 2D slice of B is sampled densely at each intersection point to construct a quad tree over the samples. A quad tree of the environment map can be precomputed. Subsequently, the two quad trees are traversed in parallel to generate a discretized importance function, which is used to guide sampling [Clarberg and Akenine-Möller 2008].

To amortize the cost of building the PSR, we propose to adaptively cache these records on-demand in texture space. All shading locations within a local footprint can then reuse a single PSR. Our approach is unbiased, as the cached PSRs are only used to guide sampling. The example in Figure 8 illustrates the generality of the texture space caching system, and shows how large data structures can be cached to accelerate advanced rendering tasks. In addition, at relMSE=0.01, cached product importance sampling is 4× faster than product importance sampling and 2.5× faster than multiple importance sampling. Furthermore, a biased version of this approach would be even faster.

Diffuse Interreflection In a diffuse scene, our texture space system can be seen as a flexible form of irradiance caching (IC) [Ward et al. 1988]. Both approaches reuse expensive shading terms and decouple primary visibility from shading. Next, the most important differences are summarized. IC uses a 3D search structure, while

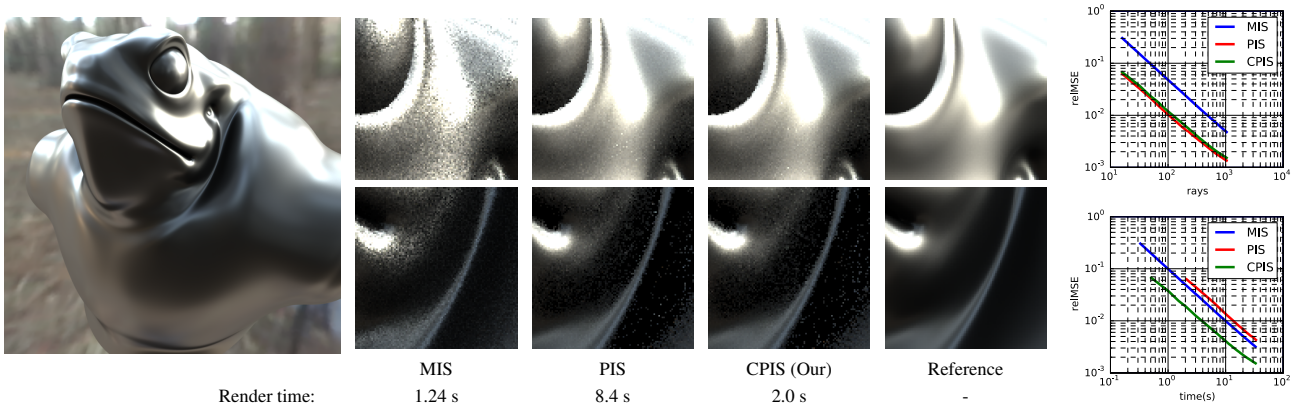


Figure 8: *The Toad scene using a brushed metal material. We show multiple importance sampling (MIS), product importance sampling (PIS), and cached product sampling (CPIS). The convergence plots show that PIS converges quickly, but is about 7× slower than MIS at equal number of rays. However, CPIS is 4× faster than PIS at similar quality and about 2.5× faster than MIS.*

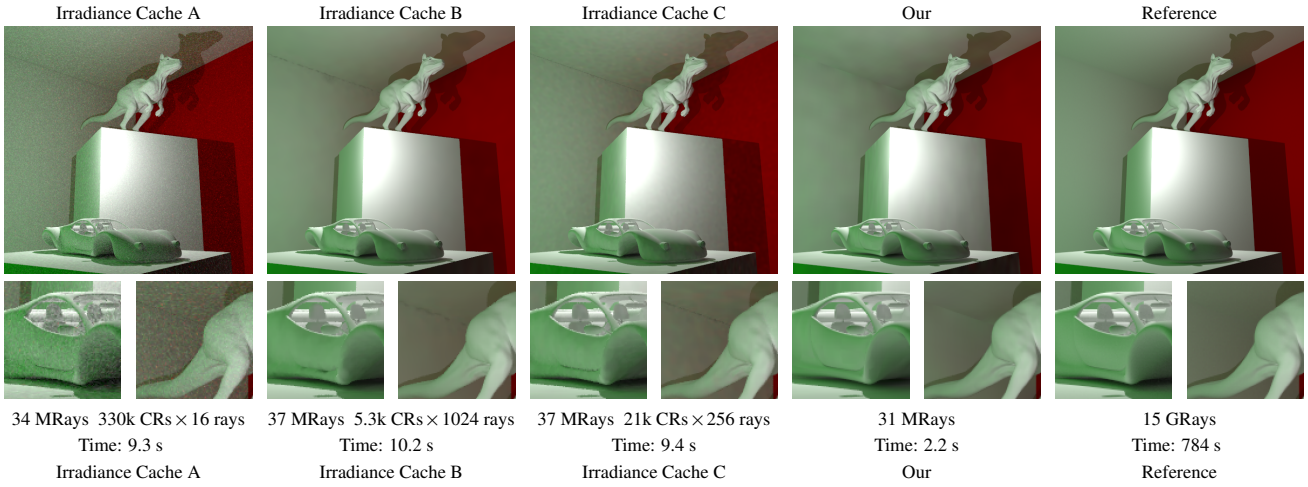


Figure 9: *Comparison against irradiance caching. Since irradiance caching is highly configurable, we present equal ray comparisons for three configurations.* ***A****: many, coarsely sampled cache records (CRs),* ***B****: few, densely sampled records, and* ***C****: a middle ground with more (decently sampled) records. As can be seen, with few rays per record, there is significant residual noise. With few records, there are visible low-frequency interpolation artifacts. At the same ray budget, our texture space caching solution has very good quality. The irradiance cache images were rendered using* `pbrt-v2`*'s implementation, while our method uses the Embree ray tracing system, so the render times are not directly comparable and the lighting is slightly different. On a dual Xeon E5-2680, a path traced version of this scene renders about 3× faster in our Embree framework than in* `pbrt-v2` *(20.2 MRays/s vs. 6.6 MRays/s). In the comparisons above, we generate similar quality output using texture space caching 4–5× faster than the irradiance caching renderer in* `pbrt-v2`*.*

our system stores filtered values in texture space and requires a surface parameterization. IC spends significant effort in placing the cache records non-uniformly in the scene, with more records in regions with complex visibility. Typically, the cache records are computed in a separate render pass. In contrast, we evaluate and filter shading over uniform grids in texture space, where the grid resolution is selected based on ray differentials. In animations, there may be scintillation artifacts from IC due to temporal visibility changes and temporally incoherent or noisy cached samples [Keller et al. 2014]. Our shading samples are instead static in object space. We show examples of temporal shading reuse later in this section.

In Figure 9, we use the irradiance caching implementation available in `pbrt-v2` [Pharr and Humphreys 2010] and compare three configurations against our texture space system at equal number of rays. As reconstruction filter for this comparison, we apply a Gaussian kernel in a 20×20 window. We only include samples where $\cos\theta > 0.9$, where θ is the angle between the sample normals. Visually, the texture space solution is smoother, but still faithfully captures the illumination changes. An equal-time comparison is difficult given the two different rendering systems, but note that the texture space system uses a single pass and avoids the kNN-search and (multi-threaded) octree updates.

Temporal Reuse Temporal reuse is greatly simplified by shading and filtering in texture space as samples remain static on the surface even if the scene animates. For a given texel, we simply accumulate the current sample to the one already residing in the cache, using a simple infinite impulse response (IIR) filter,

$$c_i = (1 - \alpha) \cdot c + \alpha \cdot c_{i-1}, \tag{6}$$

where c is the shaded color for the current frame and c_{i-1} is the accumulated color from previous frames. We use $\alpha = 0.7$ in Figure 10 and the video. Referring to the accompanying video, note that image space approaches typically struggle when objects undergo complex motion. For example, Moon et al. [2015] use a heuristic based on world space positions and similar colors to guide reprojection. This works well for camera animations, but is less useful in regions with large object motion or deformation. In those

Figure 10: *Two crops from frame 76 of the Barbarian animation. Please refer to the supplemental video for the full animated sequence. Temporal shading reuse in texture space works surprisingly well from sparsely sampled input.*

cases, the motion may cause two completely different points on the surface to map to similar world space positions in two consecutive frames. As can be seen in the video, this leads to noise or overblurring. An image comparison is also shown in Figure 10. Temporal reprojection was not a main focus of Moon et al.'s work, but we note that it is non-trivial to make screen space reprojection robust with respect to object motion. Texture space reuse removes most artifacts related to reprojection. Still, some remaining artifacts can be seen in regions with rapidly varying shading.

Temporal reuse requires that the cache from the previous frame is backed by memory, which is a reasonable requirement shared by other methods. For the animation in the accompanying video, our cache requires about 93 MB of storage for shading cache and framebuffer, while the algorithm of Moon et al. [2015] requires 83 MB to store all auxiliary buffers and temporal models. Note that cache storage is roughly proportional to the number of pixels in the rendered image as we only filter visible texels, and mip mapping will ensure we filter approximately once per pixel.

Shading Reuse for Stereo/VR Our system separates primary visibility from the remainder of the ray path, so we can easily cache and reuse view-independent shading between views, and get stereo rendering at a fraction of the cost of two separate renders. For example, in a diffuse scene, only the primary rays need to be traced for the second view, plus a small amount of new texels shaded due to disocclusion.

Figure 11 shows an example of the Toad mesh rendered for a head-mounted display (HMD). In this case, view-independent ambient occlusion is evaluated, cached, and denoised in texture space, while the specular paths are re-traced in the second view to obtain correct specular reflections. Hence, the view-independent effects are effectively reused between the two views. In fact, rendering the view for the right eye only adds shading of 8.9% additional texels. Note that due to the chromatic aberrations in the optical path of the HMD's display system, we need to trace individual rays to evaluate the three primaries (red, green, blue) for each pixel. With our system these three rays are relatively inexpensive as they can share already evaluated shading to a large extent.

Soft Shadows A number of recent papers use frequency analysis to provide bandwidth estimates for specific effects, including soft shadows and indirect illumination [Egan et al. 2009; Egan et al. 2011b; Egan et al. 2011a; Belcour et al. 2013]. In texture space, we can directly apply these bandwidth estimates to specific shader terms. As a proof-of-concept, we apply a 4D soft shadow filter proposed by Mehta et al. [2012], which provides bandwidth estimates based on distance to closest occluder in a local footprint. In Figure 12, we note that this domain-specific filter in texture space is more efficient than ARLP when applied to the isolated effect of shadow filtering.

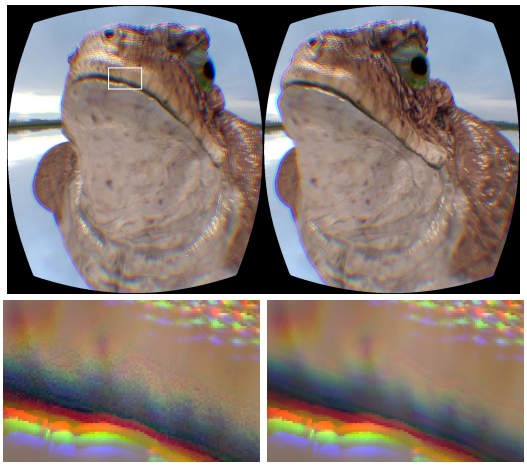


Figure 11: *The Toad model rendered for the Oculus Rift DK2 VR headset, using 12 primary rays (4×RGB) per pixel and 8 ambient occlusion rays per hit point. With caching in texture space, these primaries often reuse already cached shading, whereas a traditional path tracer would re-evaluate the shader at each hit. Another benefit is that view-independent effects are reused between the two eyes, while specular shading is re-traced for each eye. The two effects together reduce the total ray count by 4.4×, and only 8.9% additional texels are shaded to evaluate ambient occlusion for the second view.*

6.2 Scaling Properties

Cache Size In Figure 13, we illustrate the effect of the sample shading cache size. A limited cache footprint is particularly beneficial for resource-limited applications, such as ray tracing of massive models and production rendering. We therefore use high quality settings and very high resolutions to create similar rendering conditions. As a reference, the frame buffer alone requires 256 MB

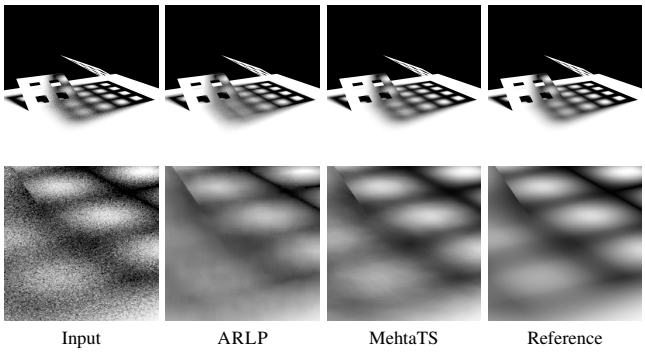


Figure 12: *Soft shadow denoising example, where we apply a domain specific shadow filter by Mehta et al. in texture space (denoted MehtaTS) and compare it with* ARLP. *All examples but the reference use approximately 19 Mrays.*

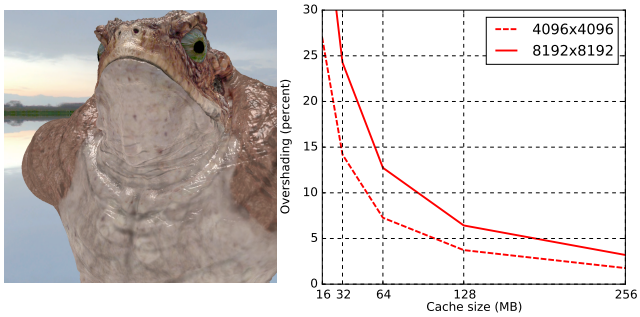


Figure 13: *We render the Toad scene at high resolutions, and vary the cache size to examine how shading rate scales. The diagram to the right shows overshading for varying cache sizes and resolutions, when compared to a reference with unlimited storage.*

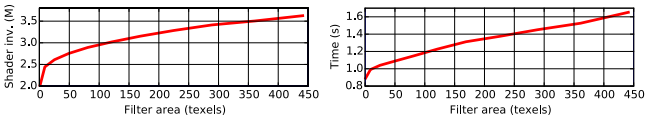


Figure 14: *The Car scene with increasing filter size.*

Time	14.0 s	6.5 s	3.5 s	2.5 s
Shader inv.	1.2M	0.49M	0.23M	0.14M
Rays	250M	110M	57M	38M
relMSE	0.005	0.012	0.026	0.060

Figure 15: *Image crops from the Car scene (Figure 6), with mip LOD biased by 1-3 levels to reduce shading rate. The shading and shadowing quality is reduced for higher LOD.*

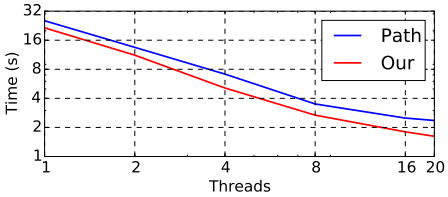


Figure 16: *Execution time for the Car scene as a function of the number of threads, up to the number of cores of our computer.*

of storage at 4K×4K resolution, i.e., not including guide images (normals, depth, etc.) which are typically required by screen space filtering approaches. As can be seen from the graph, at this resolution we shade an additional 2% to 27% when the cache size is reduced from 256 MB to 16 MB, compared to an ideal, unlimited cache. That is, the cache is efficient even when its size is reduced.

Image Size Referring to Figure 13 again, we note that our system scales well with resolution. A $4\times$ increase in resolution can be counteracted by increasing the cache size by less than $2\times$. This is expected, as higher resolution leads to more coherent cache lookups. In contrast, image-based algorithms rely on guide images with storage directly proportional to image resolution. For 4K×4K renderings, our implementations of ARLP and AM require 1088 MB and 640 MB, respectively, for guide images alone. This may be reduced by using half-float or fixed-point image formats, but this may reduce performance or degrade image quality. It can be argued that screen space tiling may alleviate this issue, but we remark that most screen space algorithms are not evaluated under such circumstances. Tiling may affect reconstruction performance, or lead to redundant shading in systems where there is no easy way to share data in tile guard bands, e.g., in a rendering system where tiles are rendered on separate nodes.

Filter Size The graphs in Figure 14 show how shader invocations and render time scale with filter size. Larger filters increase time spent in filtering computations, but even with large filters (21×21 samples), the rendering time is only doubled compared to no filtering at all. Texture space filter footprints may include samples which are occluded or backfacing from the viewer, causing the number of shader invocations to grow as the footprint increases. Similarly, in regions where the mip level changes, samples will be shaded for both mip levels in the area overlapped by the filter footprint. This effect may be reduced by propagating shading results up and down the mip map pyramid [Gribel et al. 2011], but we leave such investigations to future work.

Shader Level of Detail As previously shown by Clarberg et al. [2014], we can use the texture space shading cache to perform shader level of detail (LOD) through subsampling. Low-frequency shading components, such as diffuse indirect lighting and ambient occlusion, may be evaluated at a rate that is lower than once per texel without significantly reducing image quality.

We have experimented with a global LOD parameter, which bias the mip level computation of the shading cache. In Figure 15, we show performance for varying degrees of subsampling. Generous speedups are obtained with reasonable image quality. This simple scheme is beneficial in our pre-visualization application, as we may increase frame rate further while still keeping shading quality at a sufficient level. We demonstrate this in the accompanying video.

Thread Scaling In Figure 16, we show execution time as a function of the number of threads for the Car scene. Our system shows very similar behavior to that of a path tracer. Hence, it is reasonable to assume that our parallel cache implementation is efficient from a threading perspective.

6.3 Comparisons Against Screen Space Filters

We compare against screen space filters in Figure 17, namely robust denoising using feature and color information (RDFC) [Rousselle et al. 2013], weighted linear regression (LWR) [Moon et al. 2014], adaptive rendering using linear prediction (ARLP) [Moon et al. 2015], and a learning based filter (LBF) [Kalantari et al. 2015]. We used the available source code, and adjusted the settings to get approximately equal number of rays. RDFC, LWR, ARLP, and LBF each use a modified version of `pbrt-v2` combined with CUDA-kernels. The other algorithms are implemented in our Embree ray tracing system (CPU only). Therefore, the render times are not directly comparable and the shading is slightly different. The source code for ARLP is not publicly available, but the comparison image was generated by the ARLP authors on our request.

RDFC was configured with two adaptive passes and the `wnd_rad` parameter set to 10 (default settings). For LWR and ARLP, four passes with an average of 4 spp in each iteration was used. For LBF we used the trained weights accompanying their source code.

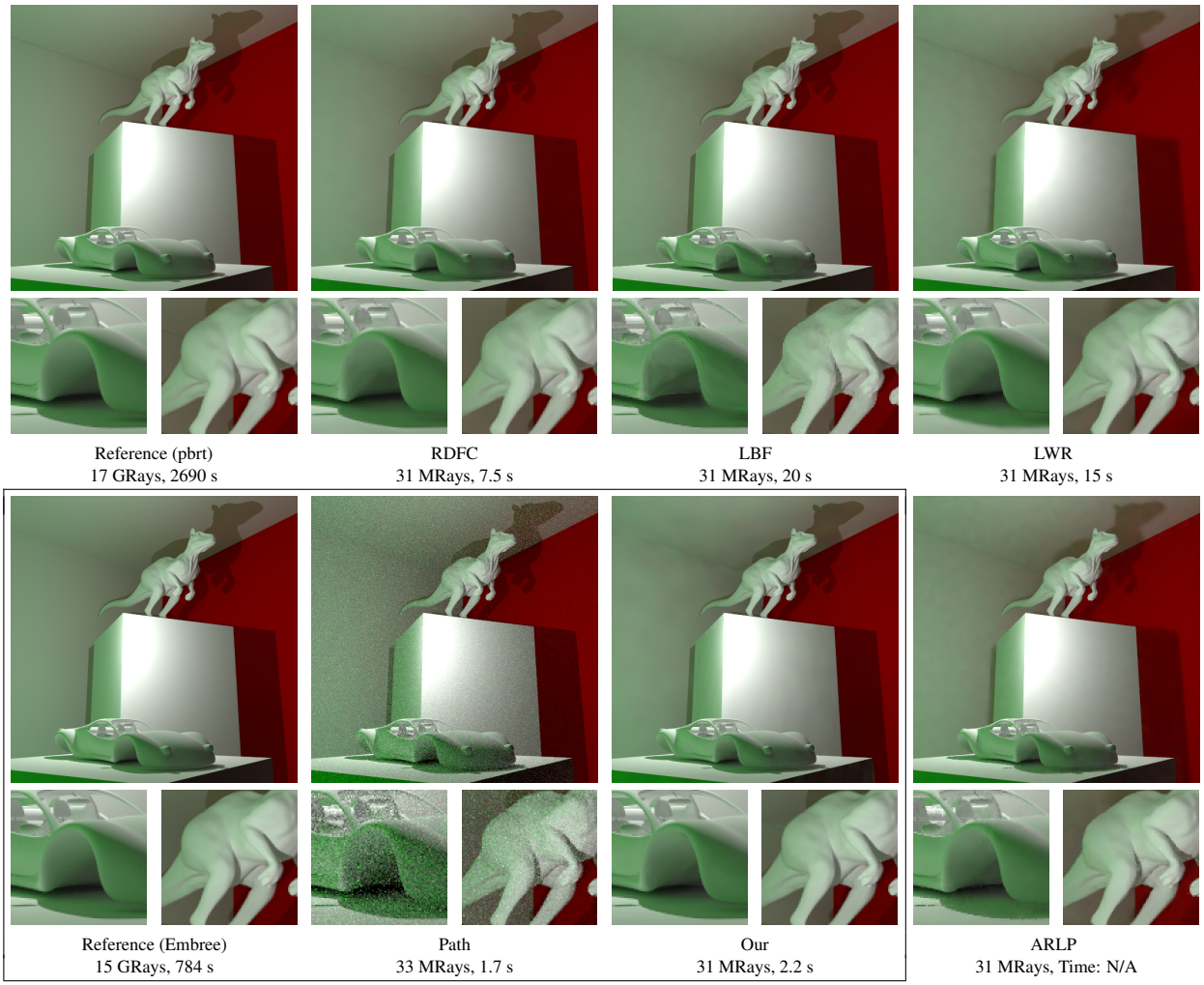


Figure 17: *Comparison against a variety of screen space filtering techniques rendered at 512×512 resolution, with approximately 16 samples per pixel (31 MRays). We compare against robust denoising using feature and color information (RDFC), a learning based filter (LBF), weighted linear regression (LWR) and adaptive rendering using linear prediction (ARLP). All methods on the upper row, and ARLP use modified versions of* `pbrt-v2` *combined with CUDA-accelerated filter kernels. The algorithms in the box in the lower row use our (CPU-only) Embree-based ray tracing system. Therefore, the render times are not directly comparable and the shading is slightly different.*

Our algorithm applies a simple texture space kernel for the indirect illumination. As reconstruction filter, we apply a Gaussian kernel in a 20×20 window. We include samples if $\cos\theta > 0.9$, where θ is the angle between the sample normals.

All algorithms efficiently reduce the residual noise. At close inspection, LBF produces some banding artifacts on the curved objects. LWR and ARLP overly smooth the hard shadow borders, which may be due to the lack of visibility as a feature. RDFC faithfully captures the shadow borders, but on the car hood, some shading transitions look overly sharp. Our method uses a very simple texture space filter and contains some residual noise on the planar surfaces, but handles the shadow borders and shading on the curved objects nicely. In terms of execution time, ARLP (as reported in their paper) and our method have modest overhead compared to a path traced input, while the other methods have significantly larger overheads. However, keep in mind that the ray tracing engines are different in this comparison and that some methods are GPU accelerated. Note also that RDFC, ARLP, and LWR all use adaptive sampling, while this is left for future work for our method.

7 Limitations

Texture space shading requires a surface parameterization on a per-patch or per-mesh level, which is often readily available, e.g., we use Ptex (mostly quad-patches) on subdivision and quad meshes. For triangle meshes, mesh colors [Yuksel et al. 2010] or existing texture parameterizations [Andersson et al. 2014; Hillesland and Yang 2016] could be used. We also need the ability to query the intersected surface for the position and normal at a texture coordinate. Care must be taken to ensure that the result corresponds to what an intersecting ray reports, taking tessellation and displacement mapping into account. Previous solutions involve caching the tessellated triangles and performing a barycentric-to-triangle lookup [Clarberg et al. 2014]. This is not a major hurdle since triangles are already cached for ray tracing [Wald et al. 2014], although our current implementation does not yet do this.

One potential disadvantage is that invisible points may be shaded, adding overhead compared to screen space methods. An example is shown in Figure 18. One way to avoid this is to trace back to the camera to exclude occluded points, which may be reasonable

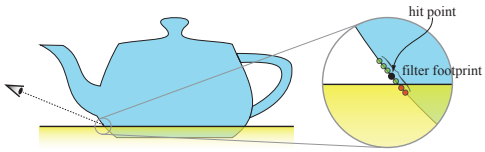


Figure 18: *A point on the teapot, which intersects the yellow ground plane, is hit and several shading values are requested in order to filter shading at the hit point. Some of the shading points are below the ground plane and may contribute to incorrect shading since the red samples are not visible from the eye. One may argue, however, that the object has been positioned in an invalid position.*

if shading is expensive. Similarly, backfacing points may be excluded. Another aspect is that our system can perform accumulation over frames in either texture space or in screen space, but not simultaneously. The size of the texture space kernels should ideally shrink with the number of samples for consistency, but that would require retaining all samples across multiple frames, which is memory intensive. Path space filtering [Keller et al. 2014] faces a similar difficulty and propose to hold as many vertices as fits in memory.

8 Conclusion

We have demonstrated the feasibility of a texture space caching system within a ray tracing setting, and shown several applications of shading reuse and filtering. These include interactive rendering of ambient occlusion with motion blur and depth of field, VR ray tracing, linear regression in texture space, cached product importance sampling, and temporal reuse. In addition, we have shown that the shading rate can be easily controlled and that the rate determines the level of shading reuse. Our system also provides filtering over adjacent faces/patches, which the streaming nature of rasterization-based systems cannot handle. The decoupling of visibility and shading means that (noisy) visibility from defocus and/or motion blur does not corrupt the cached shading, and as a consequence, existing screen space filters can be applied to reduce visibility noise as a post-process. We have only explored a small set of filtering techniques, and we believe that there is a plethora of work in novel material filters to be done. Adaptive sampling in texture space is an interesting avenue for future work, and we believe that techniques from global, adaptive sampling strategies could be applied.

Acknowledgements Thanks to Robert Toth and Jim Nilsson for help with implementation and video, Charles Lingle and David Blythe for supporting this research, and the ARLP authors for help generating the comparison image in Figure 17. The Toad King is courtesy of Craig Barr, 2008, Polymorphic3D. The Turtle Barbarian model is courtesy of Autodesk (Jesse Sandifer is the original artist). The Killeroo subdivison model is courtesy of Headus (metamorphosis) Pty Ltd. The car model is part of the OpenSubdiv SDK.

References

ANDERSSON, M., HASSELGREN, J., TOTH, R., AND AKENINE-MÖLLER, T. 2014. Adaptive Texture Space Shading for Stochastic Rendering. *Computer Graphics Forum, 33*, 2, 341–350.

BASTOS, R., GOSLIN, M., AND ZHANG, H. 1997. Efficient Radiosity Rendering using Textures and Bicubic Reconstruction. In *Symposium on Interactive 3D Graphics*, 71–74.

BAUSZAT, P., EISEMANN, M., JOHN, S., AND MAGNOR, M. 2015. Sample-Based Manifold Filtering for Interactive Global Illumination and Depth of Field. *Computer Graphics Forum, 34*, 1, 265–276.

BELCOUR, L., SOLER, C., SUBR, K., HOLZSCHUCH, N., AND DURAND, F. 2013. 5D Covariance Tracing for Efficient Defocus and Motion Blur. *ACM Transactions on Graphics, 32*, 3, 31:1–31:18.

BENTHIN, C., WOOP, S., NIESSNER, M., SELGRAD, K., AND WALD, I. 2015. Efficient Ray Tracing of Subdivision Surfaces Using Tessellation Caching. In *High-Performance Graphics*, 5–12.

BURLEY, B., AND LACEWELL, D. 2008. Ptex: Per-Face Texture Mapping for Production Rendering. In *Eurographics Symposium on Rendering*, 1155–1164.

BURNS, C. A., FATAHALIAN, K., AND MARK, W. R. 2010. A Lazy Object-Space Shading Architecture with Decoupled Sampling. In *High-Performance Graphics*, 19–28.

CLARBERG, P., AND AKENINE-MÖLLER, T. 2008. Practical Product Importance Sampling for Direct Illumination. *Computer Graphics Forum (Proceedings of Eurographics 2008), 27*, 2, 681–690.

CLARBERG, P., JAROSZ, W., AKENINE-MÖLLER, T., AND JENSEN, H. W. 2005. Wavelet Importance Sampling: Efficiently Evaluating Products of Complex Functions. *ACM Transactions on Graphics, 24*, 3, 1166–1175.

CLARBERG, P., TOTH, R., HASSELGREN, J., NILSSON, J., AND AKENINE-MÖLLER, T. 2014. AMFS: Adaptive Multi-Frequency Shading for Future Graphics Processors. *ACM Transactions on Graphics, 33*, 4, 141:1–141:12.

COOK, R. L., CARPENTER, L., AND CATMULL, E. 1987. The Reyes Image Rendering Architecture. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, ACM, vol. 21, 95–102.

DJEU, P., HUNT, W., WANG, R., ELHASSAN, I., STOLL, G., AND MARK, W. R. 2011. Razor: An Architecture for Dynamic Multiresolution Ray Tracing. *ACM Transactions on Graphics, 30*, 5, 115:1–115:26.

EGAN, K., TSENG, Y.-T., HOLZSCHUCH, N., DURAND, F., AND RAMAMOORTHI, R. 2009. Frequency Analysis and Sheared Reconstruction for Rendering Motion Blur. *ACM Transactions on Graphics, 28*, 3, 93:1–93:13.

EGAN, K., DURAND, F., AND RAMAMOORTHI, R. 2011. Practical Filtering for Efficient Ray-Traced Directional Occlusion. *ACM Transactions on Graphics, 30*, 6, 180:1–180:10.

EGAN, K., HECHT, F., DURAND, F., AND RAMAMOORTHI, R. 2011. Frequency Analysis and Sheared Filtering for Shadow Light Fields of Complex Occluders. *ACM Transactions on Graphics, 30*, 2, 9:1–9:13.

EISENACHER, C., NICHOLS, G., SELLE, A., AND BURLEY, B. 2013. Sorted Deferred Shading for Production Path Tracing. *Computer Graphics Forum, 32*, 4, 125–132.

GASTAL, E. S. L., AND OLIVEIRA, M. M. 2012. Adaptive Manifolds for Real-Time High-Dimensional Filtering. *ACM Transactions on Graphics, 31*, 4, 33:1–33:13.

GRIBEL, C. J., BARRINGER, R., AND AKENINE-MÖLLER, T. 2011. High-Quality Spatio-Temporal Rendering using Semi-

Analytical Visibility. *ACM Transactions on Graphics, 30*, 4, 54:1–54:12.

HASSELGREN, J., MUNKBERG, J., AND VAIDYANATHAN, K. 2015. Practical Layered Reconstruction for Defocus and Motion Blur. *Journal of Computer Graphics Techniques (JCGT), 4*, 2, 45–58.

HECKBERT, P. S. 1990. Adaptive Radiosity Textures for Bidirectional Ray Tracing. In *Computer Graphics (Proceedings of SIGGRAPH 90)*, ACM, 145–154.

HILLESLAND, K. E., AND YANG, J. C. 2016. Texel Shading. In *Eurographics 2016 – Short Papers*.

HOU, Q., AND ZHOU, K. 2011. A Shading Reuse Method for Efficient Micropolygon Ray Tracing. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia), 30*, 6, 151:1–151:8.

HOU, Q., QIN, H., LI, W., GUO, B., AND ZHOU, K. 2010. Micropolygon Ray Tracing with Defocus and Motion Blur. *ACM Transactions on Graphics, 29*, 4, 64:1–64:10.

IGEHY, H. 1999. Tracing Ray Differentials. In *Proceedings of SIGGRAPH 1999*, ACM, 179–186.

JENSEN, H. W. 2001. *Realistic Image Synthesis Using Photon Mapping*. AK Peters Ltd.

KALANTARI, N. K., BAKO, S., AND SEN, P. 2015. A Machine Learning Approach for Filtering Monte Carlo Noise. *ACM Transactions on Graphics, 34*, 4, 122:1–122:12.

KELLER, A., DAHM, K., AND BINDER, N. 2014. Path Space Filtering for Integro-Approximation Problems. In *Eleventh International Conference on Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing*.

LEHTINEN, J., AILA, T., CHEN, J., LAINE, S., AND DURAND, F. 2011. Temporal Light Field Reconstruction for Rendering Distribution Effects. *ACM Transactions on Graphics, 30*, 4, 55:1–55:12.

LUKSCH, C., TOBLER, R. F., HABEL, R., SCHWÄRZLER, M., AND WIMMER, M. 2013. Fast Light-Map Computation with Virtual Polygon Lights. In *Proceedings of I3D*, 87–94.

MEHTA, S., WANG, B., AND RAMAMOORTHI, R. 2012. Axis-Aligned Filtering for Interactive Sampled Soft Shadows. *ACM Transactions on Graphics, 31*, 6, 163:1–163:10.

MEHTA, S. U., WANG, B., RAMAMOORTHI, R., AND DURAND, F. 2013. Axis-Aligned Filtering for Interactive Physically-based Diffuse Indirect Lighting. *ACM Transactions on Graphics, 32*, 4, 96:1–96:12.

MEHTA, S. U., YAO, J., RAMAMOORTHI, R., AND DURAND, F. 2014. Factored Axis-aligned Filtering for Rendering Multiple Distribution Effects. *ACM Transactions on Graphics, 33*, 4, 57:1–57:12.

MEHTA, S. U., KIM, K., PAJAK, D., PULLI, K., KAUTZ, J., AND RAMAMOORTHI, R. 2015. Filtering Environment Illumination for Interactive Physically-Based Rendering in Mixed Reality. In *Eurographics Symposium on Rendering (EI&I)*.

MEYER, M., AND ANDERSON, J. 2006. Statistical Acceleration for Animated Global Illumination. *ACM Transactions on Graphics, 25*, 3, 1075–1080.

MOON, B., CARR, N., AND YOON, S.-E. 2014. Adaptive Rendering Based on Weighted Local Regression. *ACM Transactions on Graphics, 33*, 5, 170:1–170:14.

MOON, B., IGLESIAS-GUITIAN, J. A., YOON, S.-E., AND MITCHELL, K. 2015. Adaptive Rendering with Linear Predictions. *ACM Transactions on Graphics, 34*, 4, 121:1–121:11.

MUNKBERG, J., VAIDYANATHAN, K., HASSELGREN, J., CLARBERG, P., AND AKENINE-MÖLLER, T. 2014. Layered Light Field Reconstruction for Defocus and Motion Blur. *Computer Graphics Forum, 33*, 4, 81–92.

PARKER, S. G., BIGLER, J., DIETRICH, A., FRIEDRICH, H., HOBEROCK, J., LUEBKE, D., MCALLISTER, D., MCGUIRE, M., MORLEY, K., ROBISON, A., AND STICH, M. 2010. OptiX: A General Purpose Ray Tracing Engine. *ACM Transactions on Graphics, 29*, 4, 66:1–66:13.

PHARR, M., AND HUMPHREYS, G. 2010. *Physically Based Rendering: From Theory to Implementation*, 2nd ed. Morgan Kaufmann.

RAGAN-KELLEY, J., LEHTINEN, J., CHEN, J., DOGGETT, M., AND DURAND, F. 2011. Decoupled Sampling for Graphics Pipelines. *ACM Transactions on Graphics, 30*, 3, 17:1–17:17.

RAY, N., ULYSSE, J.-C., CAVIN, X., AND LÉVY, B. 2003. Generation of Radiosity Texture Atlas for Realistic Real-Time Rendering. In *Eurographics 2003 – Short Papers*.

ROUSSELLE, F., KNAUS, C., AND ZWICKER, M. 2011. Adaptive Sampling and Reconstruction Using Greedy Error Minimization. *ACM Transactions on Graphics, 30*, 6, 159:1–159:12.

ROUSSELLE, F., MANZI, M., AND ZWICKER, M. 2013. Robust Denoising using Feature and Color Information. *Computer Graphics Forum 32*, 7, 121–130.

VAIDYANATHAN, K., MUNKBERG, J., CLARBERG, P., AND SALVI, M. 2015. Layered Light Field Reconstruction for Defocus Blur. *ACM Transactions on Graphics, 34*, 2, 23:1–23:12.

VEACH, E. 1998. *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis.

WALD, I., WOOP, S., BENTHIN, C., JOHNSON, G. S., AND ERNST, M. 2014. Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Transactions on Graphics, 33*, 4, 143:1–143:8.

WARD, G. J., RUBINSTEIN, F. M., AND CLEAR, R. D. 1988. A Ray Tracing Solution for Diffuse Interreflection. In *Computer Graphics (Proceedings of SIGGRAPH 88)*, ACM, 85–92.

YAN, L.-Q., MEHTA, S. U., RAMAMOORTHI, R., AND DURAND, F. 2015. Fast 4D Sheared Filtering for Interactive Rendering of Distribution Effects. *ACM Transactions on Graphics 35*, 1, 7:1–7:13.

YUKSEL, C., KEYSER, J., AND HOUSE, D. H. 2010. Mesh Colors. *ACM Transactions on Graphics, 29*, 2, 15:1–15:11.

ZIMMER, H., ROUSSELLE, F., JAKOB, W., WANG, O., ADLER, D., JAROSZ, W., SORKINE-HORNUNG, O., AND SORKINE-HORNUNG, A. 2015. Path-space Motion Estimation and Decomposition for Robust Animation Filtering. *Computer Graphics Forum, 34*, 4, 131–142.

ZWICKER, M., JAROSZ, W., LEHTINEN, J., MOON, B., RAMAMOORTHI, R., ROUSSELLE, F., SEN, P., SOLER, C., AND YOON, S.-E. 2015. Recent Advances in Adaptive Sampling and Reconstruction for Monte Carlo Rendering. *Computer Graphics Forum (Proceedings of Eurographics), 34*, 2, 667–681.