Practical Layered Reconstruction for Defocus and Motion Blur

Jon Hasselgren Jacob Munkberg Karthik Vaidyanathan

Figure 1. We accelerate layered reconstruction for defocus and motion blur [Munkberg et al. 2014] (denoted LRDM) by over 4× with very little impact on image quality.

Abstract

We present several practical improvements to a recent layered reconstruction algorithm for defocus and motion blur. We leverage hardware texture filters, layer merging and sparse statistics to reduce computational complexity. Furthermore, we restructure the algorithm for better load-balancing on graphics processors, albeit at increased memory usage. We show run time reductions by 1/2 to 1/5 with minimal change in image quality vs. previous techniques, bringing this reconstruction technique to the real-time domain.

1. Introduction

Monte-Carlo sampling is a popular technique for simulating realistic camera effects, such as depth of field and motion blur. However, it requires a large number of samples to converge to a result with acceptable noise levels.

Several recent papers [Lehtinen et al. 2011; Vaidyanathan et al. 2015; Munkberg et al. 2014] focus on reconstruction algorithms for defocus and/or motion blur, without requiring a second adaptive sampling pass. They generate a higher quality image from a sparsely sampled light field with accompanying per-sample motion vectors and depth. These algorithms are biased, but produce images close to ground truth at

 ISSN 2331-7418

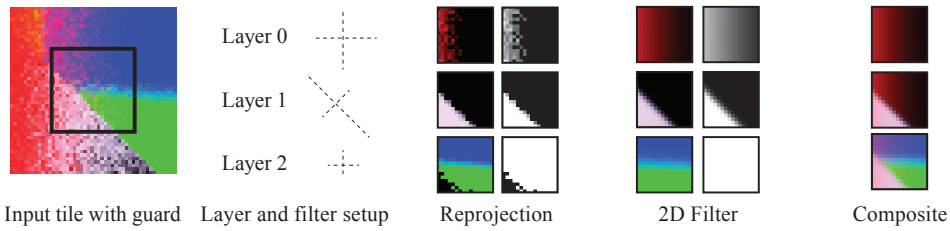


Figure 2. Algorithmic flow for LRDM [Munkberg et al. 2014].

low sample rates, and are temporally robust.

This paper presents practical improvements and optimizations to the layered defocus and motion blur reconstruction algorithm by Munkberg et al. [2014], herein denoted as LRDM. LRDM is about two orders of magnitude faster compared to offline reconstruction, e.g., Lehtinen et al. [2011; 2012], but it is still not fast enough for real time applications. In this paper, we improve the performance of LRDM by a factor $2-5\times$, bringing reconstruction filters into the real-time domain.

2. Summary of the LRDM algorithm

Layered reconstruction for defocus and motion blur (LRDM) [Munkberg et al. 2014], forms the basis for our work. LRDM divides the samples into screen space tiles, and each tile into *depth layers*. Each such combination of tile and depth layer is called a *partition*, and a sheared filter in (x,y,u,v,t)-space is derived from frequency analysis of the light field [Egan et al. 2009; Egan et al. 2011; Belcour et al. 2013] of all samples falling in that partition. Here, (x,y) denote screen space coordinates, (u,v) lens coordinates and t time.

To make the filter derivation tractable, a sample's motion is approximated to be linear within the shutter interval, and a common filter is derived per partition, based on the samples' motion vectors and depths. The LRDM algorithm can be split into three main passes, shown in Figure 2:

Filter Setup All samples in a small screen-space region, say 32×32 pixels, are partitioned into depth layers, where the depth layers are typically fixed and configured to have an even spacing by circle of confusion. For each partition, parameters for a sheared 5D filter are computed, based on the motion vectors and depth ranges of the partition. Out-of-focus partitions get larger filter kernels, sheared in xu and yv, and motion blurred partitions have anisotropic kernels in xyt.

Reprojection To evaluate the sheared filter, the samples are weighted by the filter in uvt space and are reprojected to $(u,v,t)=(0,0,0.5)$ for each partition. Both a color and opacity value is accumulated per pixel. A reprojected sample will

reduce the opacity of all partitions in front of it at the reprojected position, but will only contribute to the color of the partition it belongs to.

Screen-Space Filtering and Compositing Finally, a rotated Gaussian screen-space filter is applied to the color and opacity functions, which concludes the evaluation of the 5D filter. The filtered layers are then composited front-to-back using alpha blending based on the filtered colors and opacity values. More formally, for a set of N layers with filtered layer irradiance $e_i(x,y)$ and filtered layer opacity $\alpha_i(x,y)$, the final irradiance $e(x,y)$ is approximated as:

$$e(x,y) \approx e_0(x,y) + \sum_{j=1}^{N-1} e_j(x,y) \prod_{k=0}^{j-1} (1 - \alpha_k(x,y)).$$

The key to performance is the separation of the sheared 5D filter into a reprojection step followed by a 2D screen-space filter, instead of an expensive gather in 5D space. This separation is possible because LRDM uses a common filter for each partition, rather than a filter that potentially varies for every pixel.

3. Algorithm Load Balancing

The original LRDM implementation focused on reducing memory bandwidth by keeping data in shared local memory. However, this limits the opportunities for optimizations, as most of the algorithm was implemented in a single kernel. For this workload, the memory bandwidth is far from saturated, and we have noted that it is far more important to efficiently load-balance the computational steps. By storing intermediary results in GPU memory we can divide the implementation into smaller kernels, and exploit hardware texture filtering.

In practice, we found that it is most efficient to use one kernel for each step (filter setup, reprojection and screen-space filter). Additionally, the screen-space filter is divided into two separable kernels. Each kernel can be parallelized independently, which improves load-balancing. We note that optimal thread and group sizes for the kernels vary somewhat by graphics vendor. Therefore, we run a setup script that evaluates performance for a set of work group sizes for each pass of the algorithm.

4. Filter Setup Optimizations

LDRM computes filter parameters for each partition (a depth layer in a tile), based on the mean and variance of the depth and motion vector of *all* samples in a partition, typically about 32×32 samples. This filter setup step can be significantly reduced through a few optimizations.

Sparse statistics An efficient way to optimize filter setup, is to approximate the statistics (sample mean and variance) of each partition based on a subset of the sam-

ples. LRDM requires that a reasonable number of samples fall into each partition to achieve good quality, and this is also commonly the case. Thus, when computing the mean and standard deviations, we process every n:th sample, where n can be tuned to trade quality for performance. We use a stratified random sampling strategy to select a representative subset of samples, where we address the samples of a tile in pixel Morton order and randomly pick an address from every bucket of n samples. The address can be generated with a simple random number generator such as xor-shift [Marsaglia 2003]. In practice, we use $n = 3 \times \text{spp}$, e.g., $n = 12$ for scenes with four samples per pixel, without noticeable image quality impact.

Layer merging LRDM uses a static layering strategy where camera space is divided into depth buckets. Each bucket spans a predetermined range in depth, which is based on the circle of confusion near the focus plane, and is uniform for the remainder of the depth range. This is motivated by equal spacing relative to circle of confusion being a good approximation of the partition's filter size with respect to depth of field. The uniform depth layers outside this region ensure partitions cover a small enough depth range not to introduce artifacts in the presence of motion blur. In both regions, a sample can be assigned to a layer in constant time.

However, since the layers are static in camera space, many unnecessary partitions can be created. For example, if a tile contains samples that fall in two depth buckets, we will create two partitions even if the samples have a narrow depth range. We improve on the original layering algorithm by sweeping through all populated buckets (in sorted depth order) and merge them, if possible. As merge heuristic, we compute screen-space filter extents for both candidates, as well as the merged result. We merge two partitions if the screen-space filter size do not shrink by more than a given threshold of 10%, which did not introduce any noticeable artifacts in any of our test scenes.

We found that several partitions could be merged, without significantly impacting image quality. For our test scenes, the average number of partitions per tile is reduced from 2.7 with the original implementation to 1.9 after merging. Note that the layers are derived from visible samples only, and are not related to the scene's depth complexity. Rather, the figures should be interpreted as depth layers in a tile that has significantly different filtering characteristics (circle of confusion or motion vector). Put together, these two approximations more than double the performance of the filter setup step compared to the LRDM implementation. Furthermore, due to the reduced number of partitions, the subsequent steps are significantly faster.

5. Reprojection

In this kernel, all samples are reprojected to $(u,v,t) = (0,0,0.5)$, along partition-specific slopes derived in the filter setup step. The reprojected screen-space position

of a sample (x, y, u, v, t) is given by

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + A \begin{bmatrix} u \\ v \end{bmatrix} + \mathbf{b}\, t, \tag{1}$$

where A is a 2×2 diagonal matrix based upon the lens configuration, and the vector $\mathbf{b}$ is derived from the motion vector. Both A and $\mathbf{b}$ are unique per partition. The details of how these parameters are computed are outside the scope of this paper, and we refer to the work of Munkberg et al. [2014] for the full derivation. All samples are reprojected by the filter parameters of all (non-empty) partitions. If the sample lies in the depth range of the partition, both color and opacity is updated at the reprojected position. For partitions closer to the observer than the sample, the opacity is decreased, as the sample can be seen through that layer.

Apart from the separation into a separate kernel, and the reduced number of layers, our implementation for this step follows LRDM very closely.

6. Accelerated Anisotropic Gaussian Filtering

After sample reprojection, each partition is filtered using an anisotropic Gaussian filter kernel, $w(x, y, \sigma_u, \sigma_v, \theta)$, where σ_u and σ_v are the standard deviations in the filter's local coordinate frame, and θ is the angle of rotation. The angle of rotation is chosen such that the u-axis of the filter is aligned with the motion direction, and therefore $\sigma_u \geq \sigma_v$. For a detailed filter derivation, we refer to the LRDM paper [Munkberg et al. 2014].

In LRDM, the filtering step is implemented in two rotated separable passes along the u- and v-axes. Since the filter axes do not align with the pixel grid, bilinear interpolation is used for each filtering operation to improve quality. The interpolation could not leverage hardware texture filtering, as temporary results were stored in shared local memory. Furthermore, the rotated coordinate frame makes it impractical to use efficient approximations to Gaussian filters, e.g., recursive filters [Deriche 1992].

We outline two approximate anisotropic Gaussian filters below. The first method exploits anisotropic hardware texture filters, which have good performance but reduced quality, since the filtering hardware does not closely approximate a Gaussian filter, as seen in Figure 3.

The second alternative transforms the rotated xy-filter into a sheared filter and use a recursive filter along one axis, and a 1D Gaussian along the other sheared axis. This approach, shown to the right in Figure 3, has similar quality to LRDM, and performance almost on par with hardware texture filtering.

6.1. Hardware Texture Filtering

The exact implementation details of hardware anisotropic filtering may differ between GPU vendors, and the specifications are not publicly available. However, it is a

Reference (1024 spp) Aniso one tap Aniso four taps Sheared filter

Figure 3. Quality of our two proposed filtering approaches. Anisotropic texture filtering works well in regions with motion (upper row), but suffers from artifacts in regions with only depth of field (lower row). We improve aniso quality by performing multiple texture lookups.

reasonable assumption that most approaches try to mimic EWA filtering [Heckbert 1989], which is based on Gaussian filter kernels.

Accelerating our Gaussian filtering step using hardware texture filtering is straightforward, with the main caveat being that we need to construct MIP map hierarchies on the fly for each partition. The filtering pass is implemented with `SampleGrad` texture lookups. The derivatives, ddx and ddy, indicate the full size of the filter footprint, as opposed to our σ_u and σ_v parameters, which represent the standard deviation of the Gaussian distribution. In our implementation, we determine ddx and ddy such that the filter footprint captures 95% of the Gaussian energy.

The integral of a Gaussian distribution with standard deviations σ_u, σ_v is:

$$\frac{1}{2\pi\sigma_u\sigma_v}\int_{-b}^{b}\int_{-a}^{a} e^{-\frac{1}{2}\left(\frac{u^2}{\sigma_u^2}+\frac{v^2}{\sigma_v^2}\right)} du dv = \operatorname{erf}\left(\frac{a}{\sqrt{2}\sigma_u}\right)\operatorname{erf}\left(\frac{b}{\sqrt{2}\sigma_v}\right). \tag{2}$$

We determine integration bounds to retain 95% of the energy. Furthermore, we want to discard energy equally along both axes, which is achieved when $a/\sigma_u = b/\sigma_v$. We let $a = \alpha\sigma_u$ and $b = \alpha\sigma_v$, and solve for α, given our 95% energy constraint:

$$\operatorname{erf}\left(\frac{\alpha}{\sqrt{2}}\right)^2 = 0.95 \Leftrightarrow \alpha = \sqrt{2}\operatorname{erf}^{-1}\left(\sqrt{0.95}\right) \approx 2.24, \tag{3}$$

This is the scaling coefficient used in our implementation. Taking the rotation of the filter into account, we get:

$$ddx = \alpha(\sigma_u\cos\theta, -\sigma_u\sin\theta), \quad ddy = \alpha(\sigma_v\sin\theta, \sigma_v\cos\theta). \tag{4}$$

This filtering method is very fast, but it has poor quality when the texture footprint is nearly isotropic. Typically, anisotropic texture filtering is implemented using multiple isotropic lookups along the major axis of the ellipsoid. If the footprint is isotropic we get a single trilinear lookup. Therefore, static or slowly moving regions with depth

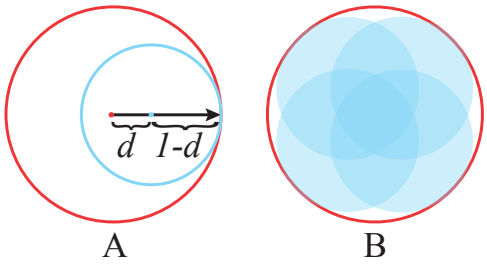


Figure 4. We improve quality with hardware texture filtering and near-isotropic kernels by sampling additional taps. A: We constrain each tap so that it is offset a distance d from the filter center, with a footprint such that it touches the original edge. B: We place four samples in the filter kernel, which gives us a coverage as illustrated by the transparent circles.

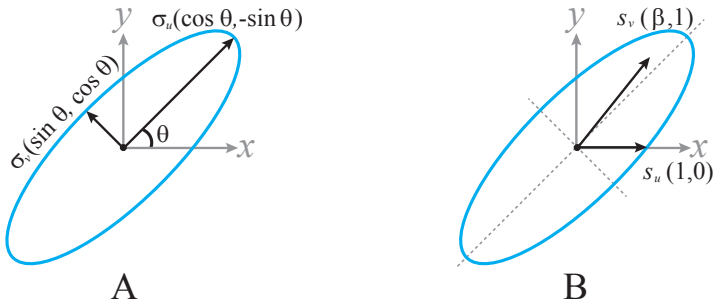


Figure 5. Any rotated 2D Gaussian filter can be decomposed into a sheared Gaussian filter, where one of the axes of the sheared filter is aligned with the x-axis. A: A rotated and scaled Gaussian filter. B: An identical Gaussian filter expressed by a sheared coordinate frame.

of field tend to look blocky, as seen in Figure 3. We alleviate this problem by using multiple texture lookups for larger filter kernels, as shown in Figure 4. In practice, four filter taps are sufficient to eliminate visual artifacts in our test scenes. Finding the optimal sample placement is non-trivial, as we wish to approximate a Gaussian, and the nature of the anisotropic filter may be hardware dependent. We chose an empirical approach, and put constraints on tap distance d, as shown in Figure 4A. Given this constraint, we found that $d \approx 0.33$ (applied to a unit Gaussian before scaling) minimized error as compared to a reference image, while avoiding most of the MIP map related artifacts.

6.2. Sheared Screen-Space Filtering

Our sheared screen space filter is based on recursive filters, which are most efficient when the filter axis is aligned with the pixel grid. We follow Geusebroek et al. [2003], who show that any rotated anisotropic Gaussian kernel can be decomposed into a sheared Gaussian kernel that is separable in x, and a sheared axis. For completeness, we derive the filter transformation below.

A rotated anisotropic Gaussian in matrix form is given by [Heckbert 1989]:

$$f(\mathbf{x}) = \frac{1}{2\pi|A|} e^{-\frac{1}{2}\mathbf{x}^\top(\mathbf{A}\mathbf{A}^\top)^{-1}\mathbf{x}}, \tag{5}$$

where $\mathbf{x} = [x, y]^\top$ is the position to be evaluated, and

$$\mathbf{A} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} \sigma_u & 0 \\ 0 & \sigma_v \end{bmatrix} \tag{6}$$

is the matrix of the rotated filter, which first performs scaling by the filter's standard deviations, followed by rotation. Similarly, referring to Figure 5, we can express a sheared Gaussian filter, with one axis aligned with the x-axis, $e_0 = (1,0)$ and the sheared axis given by $e_1 = (\beta, 1)$, as:

$$f(\mathbf{x}) = \frac{1}{2\pi|B|} e^{-\frac{1}{2}\mathbf{x}^\top(\mathbf{B}\mathbf{B}^\top)^{-1}\mathbf{x}}, \tag{7}$$

where we have used different scale factors, s_u and s_v, and

$$\mathbf{B} = \begin{bmatrix} 1 & \beta \\ 0 & 1 \end{bmatrix} \begin{bmatrix} s_u & 0 \\ 0 & s_v \end{bmatrix}. \tag{8}$$

Thus, we need to solve $\mathbf{B}\mathbf{B}^\top = \mathbf{A}\mathbf{A}^\top$. The matrices, representing coefficients for quadratic forms, are both diagonally symmetric, and have three degrees of freedom. After simplification, we arrive at the following coefficients for the sheared Gaussian:

$$s_u = \frac{\sigma_u\sigma_v}{\rho}, \quad s_v = \rho, \quad \beta = \frac{\cos\theta\sin\theta(\sigma_u^2 - \sigma_v^2)}{\rho^2}, \quad \rho = \sqrt{\sigma_v^2\cos^2\theta + \sigma_u^2\sin^2\theta} \tag{9}$$

The sheared Gaussian can be expressed as a separable filter where one filter lies along the x axis and another along the sheared axis $e_1 = (\beta, 1)$.

We evaluated both a standard convolution filter and a recursive filter approximation, and observed that the latter provides a worthwhile performance improvement in most cases. Our recursive Gaussian approximation comes from Gastal and Oliveira [2011]:

$$\text{out}[i] = (1-\gamma)\ \text{in}[i] + \gamma\,\text{out}[i-1], \tag{10}$$

where $\gamma = e^{(-\sqrt{2}/\sigma)}$. We perform recursive filtering on each tile independently. Although recursive filters have infinite support, we are still approximating a Gaussian filter, and it is reasonable to assume that we can use the same guard band as would have been used in a convolution filter.

For the filter along the sheared axis, we use a standard separable Gaussian kernel. Geusebroek et al. [2003] outline an implementation that uses recursive filters for the sheared axis as well. However, their approach requires using an additional compute-grid aligned with the sheared axis, and would unfortunately be very complicated to efficiently parallelize with our data layout.

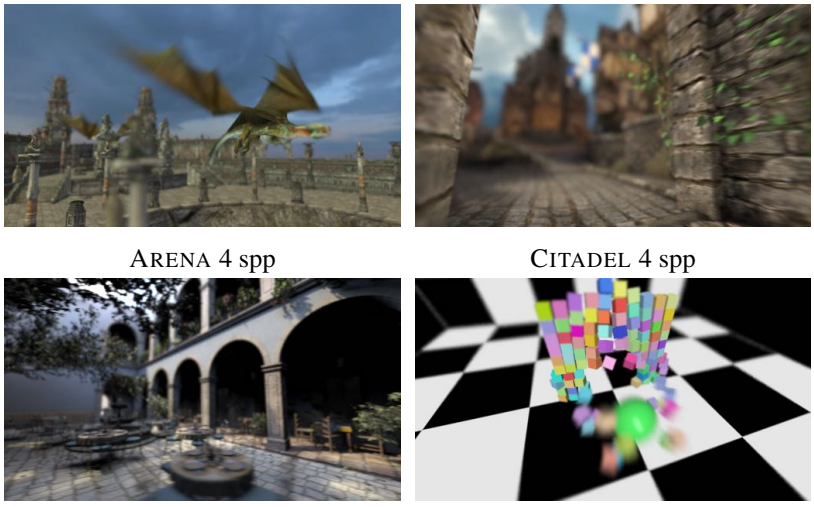


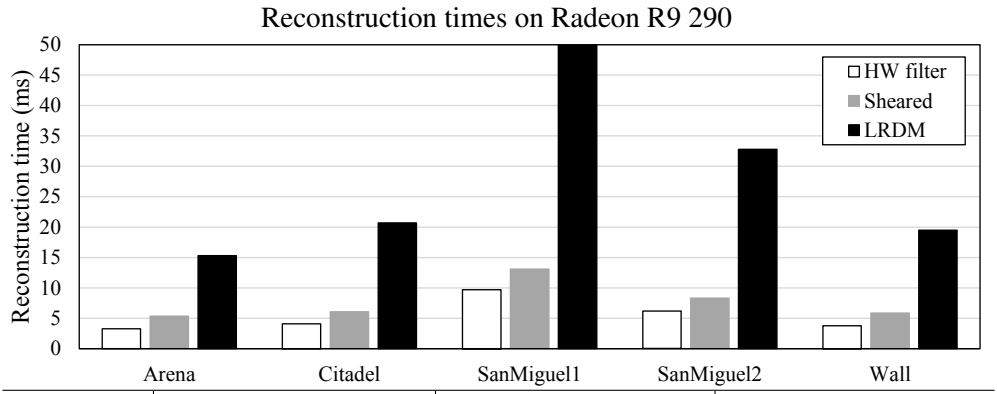
Figure 6. We use the same five test scenes as in the original paper by Munkberg et al. [2014] in order to make direct comparison easier. Refer to Figure 1 for an image of the SANMIGUEL1 scene. The scenes are rendered at 1280×720 and use between 4 and 8 samples per pixel.

7. Results

We have extended the original implementation of Munkberg et al. [2014], denoted LRDM, with the optimizations presented in this paper, and made the source code readily available online [1]. The test scenes, shown in Figure 1 and Figure 6 are using the same input data as in the original LRDM paper. All scenes are rendered at 1280×720 pixels. In Table 1, we show execution times of our algorithm on three different GPUs from major vendors. As can be seen from the results, we get up to $5\times$ performance improvement for the implementation using hardware anisotropic filters, with worthwhile performance improvements for all measured configurations. We have optimized the original LRDM implementation and therefore our scores differ slightly from the results presented by Munkberg et al. [2014]. In addition, we present a timing breakdown for the different algorithm passes in Table 2.

In Table 3, we present peak signal to noise ratio (PSNR) scores for all variants of the algorithm, compared to reference images generated using 1024 samples per pixel. As expected, further approximations introduced in this paper reduce quality when compared to the original algorithm. However, the quality reduction is typically small, with the largest reduction being 2.6 dB for the SANMIGUEL1 scene. We believe this is a reasonable tradeoff for real time applications. All images are included in the supplemental material.

[1] https://software.intel.com/en-us/articles/layered-reconstruction-for-defocus-and-motion-blur



	Radeon R9 290, 275 W			GTX980, 165 W			Iris Pro 5200, 47 W		
	New		LRDM	New		LRDM	New		LRDM
ARENA	3.3	5.2	15.3	4.4	5.6	13.2	34.9	50.1	155
CITADEL	4.1	6.1	20.7	5.1	6.7	19.1	36.8	50.8	204
SM1 8 spp	9.7	12.9	50.2	10.8	13.0	54.1	97.2	123	449
SM2 8 spp	6.1	8.3	32.8	7.4	9.3	31.1	62.9	80.9	298
WALL	3.8	5.9	19.5	4.8	6.3	17.2	34.2	49.0	195

Table 1. Performance compared to LRDM. The two numbers given for our new implementation represent hardware texture filter (left) and sheared filter (right) performance in milliseconds. Our implementation using sheared filters is 2-4× faster than previous work, and the version using hardware texture filters is 3-5× faster. Note that unlike the discrete cards, the 47W TDP for the Iris Pro includes both CPU and GPU.

LRDM		HW filter		Sheared filter	
Stats	7.08 ms	Stats	1.96 ms	Stats	1.97 ms
Reproj & filter	12.00 ms	Reproj	2.17 ms	Reproj	2.30 ms
		MIP map	0.59 ms	X filter	1.75 ms
		HWFilter	0.34 ms	Y filter	0.70 ms

Table 2. Performance breakdown, in milliseconds, for the different passes of all implementations on the CITADEL scene and Geforce GTX 980 hardware. Note that unlike LRDM we have decoupled the reprojection and filter passes.

We also integrated our improved reconstruction algorithm into a software stochastic rasterization framework [McGuire et al. 2010; Clarberg and Munkberg 2014]. In Figure 7, we present total render time for interleaved rasterization [Fatahalian et al. 2009] and reconstruction, for an animation with the camera flying through the ARENA scene. Performance of both rendering and reconstruction is consistent for this animation with frame rate varies between 30 and 40 frames per second at a resolution of 1920×1080 pixels. The reconstructed video is included as supplemental material.

Scene	HW filter	Sheared filter	LRDM
ARENA	43.0 dB	43.2 dB	44.3 dB
CITADEL	41.1 dB	41.6 dB	42.2 dB
SANMIGUEL1	34.2 dB	35.0 dB	36.1 dB
SANMIGUEL2	35.0 dB	35.0 dB	36.6 dB
WALL	35.4 dB	36.5 dB	36.1 dB

Table 3. Peak-Signal to Noise scores for our algorithms and LRDM.

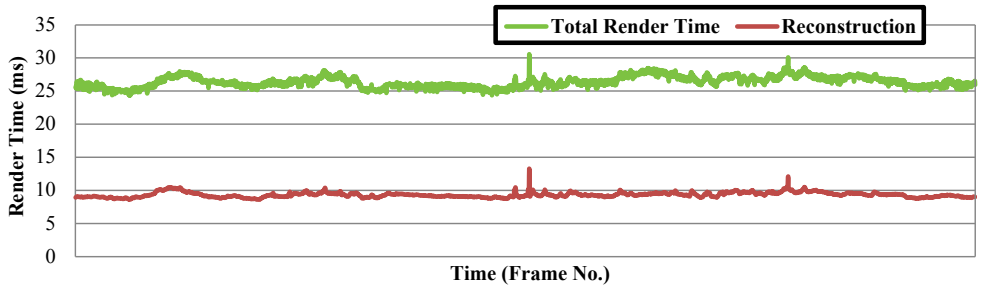


Figure 7. A camera animation through the ARENA scene, rendered at real-time frame rates using a software stochastic rasterizer with reconstruction, at 1920×1080 resolution on Radeon R9 290 hardware.

In Figure 8, we compare LRDM to our optimized implementation on a simple, yet challenging scene with high contrast materials. The quality is very similar, albeit at $2.5\times$ faster reconstruction. The most notable artifact in this scene (for LRDM and our version) is the streaking pattern on the moving checkerboard, due to the sparse input samples. This effect can be reduced by increasing the spatial filter, σ_x, at the cost of lost sharpness. This is illustrated in Figure 9.

Defocus Blur Only LRDM is an extension of previous work on layered depth of field reconstruction [Vaidyanathan et al. 2015]. Therefore, our improvements apply to depth of field reconstruction (i.e., 4D reconstruction) as well. As a proof-of-concept, we stripped the code from the motion vector and shutter filter, which gave an additional $10-30\%$ performance boost compared to the full 5D filter. Compared to the performance reported in Vaidyanathan et al.'s paper, our optimized defocus blur implementation is $3-4\times$ faster with very similar image quality.

8. Conclusions and Future Work

With our method for reconstructing sparsely sampled motion blur and depth of field, we demonstrate up to a $5\times$ improvement in performance over previous work, potentially bringing high quality camera effects into the real-time graphics domain.

It should be noted however, that our implementation trades performance for a

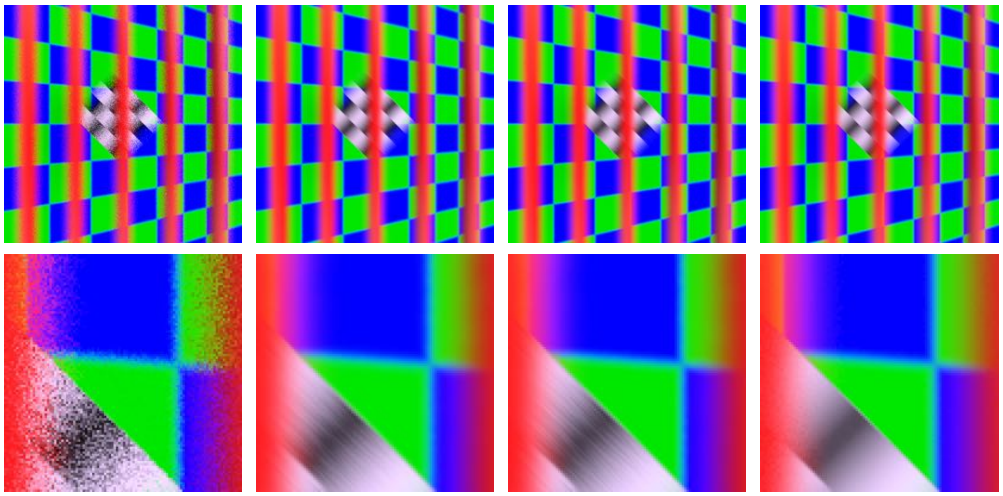


Figure 8. A simple scene with high contrast materials, an out-of-focus fence in front of a diagonally moving checkerboard and a static slanted checkerboard. The scenes is rendered at 512×512 and use eight samples per pixel.

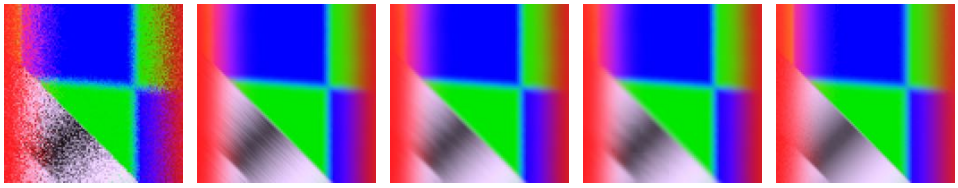


Figure 9. By varying the spatial filter, σ_x, we can reduce streaking artifacts at the cost of lost sharpness (look at the edge along the motion direction). We use a spatial filter of $\sigma_x = 0.33$ in all other images.

larger memory footprint than LRDM. We store intermediate results in textures, and we therefore need enough storage to hold tiles, including guard bands, for all layers in the scene. For 1920×1080 pixel resolution with an average of three layers per tile, we estimate that hardware texture filtering approach requires about 240 MB of video memory while sheared filtering requires 760 MB due to additional ping-pong buffers.

As future work, to limit memory consumption, we propose to split the frame buffer into a set of large tiles and running the entire algorithm on a per-tile basis, thereby limiting the memory footprint with a potential trade off in parallelism.

Acknowledgements

We thank Intel's Advanced Rendering Technology (ART) team. We also thank David Blythe and Chuck Lingle for supporting this research. The CITADEL test scene is courtesy of Epic Games, Inc., SANMIGUEL was modeled by Guillermo M. Leal Llaguno, www.evvisual.com.

References

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. 46

CLARBERG, P., AND MUNKBERG, J. 2014. Deep Shading Buffers on Commodity GPUs. *ACM Transactions on Graphics, 33*, 6, 227:1–227:12. 54

DERICHE, R. 1992. Recursively Implementing the Gaussian and its Derivatives. In *Proceedings of the 2nd Conference on Image Processing*, 263–267. 49

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. 46

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. 46

FATAHALIAN, K., LUONG, E., BOULOS, S., AKELEY, K., MARK, W. R., AND HANRAHAN, P. 2009. Data-Parallel Rasterization of Micropolygons with Defocus and Motion Blur. In *High Performance Graphics*, 59–68. 54

GASTAL, E. S. L., AND OLIVEIRA, M. M. 2011. Domain Transform for Edge-Aware Image and Video Processing. *ACM Transactions on Graphics 30*, 4, 69:1–69:12. 52

GEUSEBROEK, J.-M., SMEULDERS, A., AND VAN DE WEIJER, J. 2003. Fast Anisotropic Gauss Filtering. *IEEE Transactions on Image Processing 12*, 8, 938–943. 51, 52

HECKBERT, P. S. 1989. *Fundamentals of Texture Mapping and Image Warping*. Master's thesis, University of California, Berkeley. 50, 52

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. 45, 46

LEHTINEN, J., AILA, T., LAINE, S., AND DURAND, F. 2012. Reconstructing the Indirect Light Field for Global Illumination. *ACM Transactions on Graphics, 31*, 4, 51:1–51:10. 46

MARSAGLIA, G. 2003. Xorshift RNGs. *Journal of Statistical Software 8*, 1–6. 48

MCGUIRE, M., ENDERTON, E., SHIRLEY, P., AND LUEBKE, D. 2010. Real-Time Stochastic Rasterization on Conventional GPU Architectures. In *High Performance Graphics*, 173–182. 54

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. 45, 46, 49, 53

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. 45, 55

Author Contact Information

Jon Hasselgren
Intel Corporation
jon.n.hasselgren@intel.com

Jacob Munkberg
Intel Corporation
jacob.munkberg@intel.com

Karthik Vaidyanathan
Intel Corporation
karthik.vaidyanathan@intel.com

https://software.intel.com/en-us/intel-rendering-technologies

Hasselgren, Munkberg and Vaidyanathan, Practical Layered Reconstruction for Defocus and Motion Blur, *Journal of Computer Graphics Techniques (JCGT)*, vol. 4, no. 2, 45–58, 2015
http://jcgt.org/published/0004/02/04/

Received:	2014-12-04		
Recommended:	2015-04-10	Corresponding Editor:	Elmar Eisemann
Published:	2015-06-10	Editor-in-Chief:	Morgan McGuire