High Performance Graphics (2014)
Jonathan Ragan-Kelley and Ingo Wald (Editors)

# A Fast and Stable Feature-Aware Motion Blur Filter

Jean-Philippe Guertin[1] Morgan McGuire[2,3] Derek Nowrouzezahrai[1]

[1]Université de Montréal [2]Williams College [3]NVIDIA Research

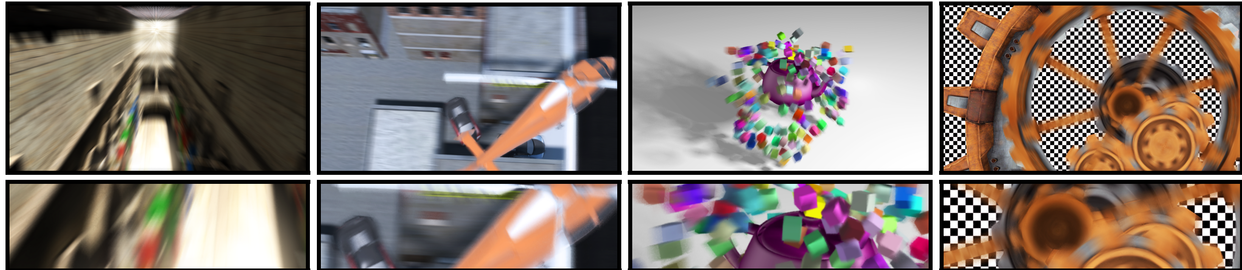

**Figure 1:** *We render temporally coherent motion blur without any motion artifacts, even on animation sequences with complex depth and motion relationships that are challenging for previous post-process techniques. All results are computed in under 2ms at* 1280×720 *on a GeForce GTX780, and our filter integrates seamlessly with post-process anti-aliasing and depth of field.*

**Abstract**

*High-quality motion blur is an increasingly important effect in interactive graphics however, even in the context of offline rendering, it is often approximated as a post process. Recent motion blur post-processes (e.g., [MHBO12, Sou13]) generate plausible results with interactive performance, however distracting artifacts still remain in the presence of e.g. overlapping motion or large- and fine-scale motion features. We address these artifacts with a more robust sampling and filtering scheme with only a small additional runtime cost. We render plausible, temporally-coherent motion blur on several complex animation sequences, all in under 2ms at a resolution 1280 × 720. Moreover, our filter is designed to integrate seamlessly with post-process anti-aliasing and depth of field.*

## 1 Introduction

Motion blur is an essential effect in realistic image synthesis, providing important motion cues and directing viewer attention. It is one of the few effects that distinguishes high-quality film production renderings from interactive graphics. We phenomenologically model the perceptual cues of motion blur sequences to approximate this effect with high quality using a simple, high-performance post-process.

We are motivated by work in offline motion blur post-processing [Coo86, ETH*09, LAC*11], where the foundational work on combining reconstruction filters with sampling was set forth. These approaches are too heavyweight for modern game engines, however many of their ideas remain useful. Conversely, adhoc approaches (e.g., blurring moving objects) have existed in various forms for several years, however the question of how to approach motion blur post-processing using a phenomenological methodology has only recently gained traction in the interactive rendering community. This methodology had first found use (and success) for post-process anti-aliasing [Lot09] and depth of field [Eng06]. We target temporally-coherent, plausible high-quality motion blur that rivals super-sampling, but with a small performance budget and simple implementation. As such, we build upon recent post-processes [MHBO12, Sou13], resolving several of their limitations while producing stable, feature-preserving and plausible motion blur.

**Contributions.** Previous works rely on conservative assumptions about local velocity distributions at each pixel in order to apply plausible, yet efficient, blurs. Unfortunately, distracting artifacts arise when these assumptions are broken; this occurs when objects nearby in image-space move with different velocities, which is unavoidable even in simple scenes, or when objects under (relative) motion have geometric features of different scales. We eliminate these artifacts and compute stable motion blur for both simple and complex animation configurations. Our contributions are:

- an improved, variance-driven directional sampling scheme that handles anisotropic velocity distributions,
- a sample-weighting scheme that preserves unblurred object details and captures fine- and large-scale blurring, and
- a more robust treatment of tile boundaries, and interaction with post-process anti-aliasing and depth of field.

© The Eurographics Association 2014.

DOI: 10.2312/hpg.20141093

delivered by EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org

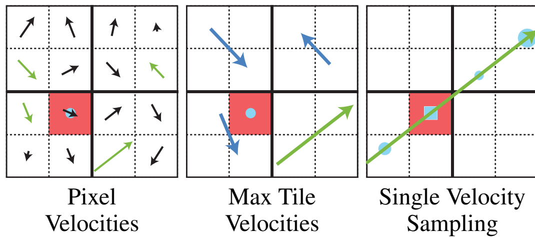


**Figure 2:** *Motion blur with "single-velocity" techniques.*

We operate on g-buffers, captured at a single time instance, with standard rasterization. Our results are stable under animation, robustly handle complex motion scenarios, and all in about 3ms per frame (Figure 1). Since ours is an image post-process filter, it can readily be used in an art-driven context to generate non-physical and exaggerated motion blur effects. We provide our full pseudocode in Appendix A.

## 2 Previous Work and Preliminaries

Given the extensive work on motion blur, we discuss recent work most related to our approach and refer interested readers to a recent survey on the topic [NSG11].

**Sampling Analysis and Reconstruction.** Cook's seminal work on distribution effects [Coo86] was the first to apply different (image) filters to reduce artifacts in the context of stochastic ray-tracing and micropolygon rasterization. More recent approaches filter noise while retaining important visual details, operating in the multi-dimensional primal [HJW*08], wavelet [ODR09] or data-driven [SD12] domains. Egan et al. [ETH*09] specifically analyze the effects of sample placement and filtering, in the frequency domain of object motion, and propose sheared reconstruction filters that operate on stochastically distributed spatio-temporal samples. Lehtinen et al. [LAC*11] use sparse samples in ray-space to reconstruct the anisotropy of the spatio-temporal light-field at a pixel and reconstruct a filtered pixel value for distribution effects. These latter two techniques aim to minimize integration error given fixed sampling budgets in stochastic rendering engines. We also reduce visible artifacts due to low-sampling rates, however we limit ourselves to interactive graphics pipelines where distributed temporal sampling is not an option and where compute budgets are on the order of milliseconds, not minutes.

**Stochastic Rasterization.** Recent work on extending triangle rasterization to support the temporal- and lens-domains [AMMH07], balances the advantages and disadvantages of GPU rasterization, stochastic ray-tracing, and modern GPU micropolygon renderers [FLB*09]. Here, camera-visibility and shading are evaluated at many samples in space-lens-time. Even with efficient implementations on conventional GPU pipelines [MESL10], these approaches remain too costly for modern interactive graphics applications. Still, Shirley et al. [SAC*11] discuss image-space filters for plausible motion blur given spatio-temporal output from such stochastic multi-sample renderers. We are motivated by their phenomenological analysis of motion blur behavior and extend their analysis to more robustly handle complex motion scenarios (see Section 4).

**Interactive Heuristics.** Some approaches blur albedo textures prior to texture mapping (static) geometry [Lov05] or extrude object geometry using shaders [TBI03], however neither approach properly handles silhouette blurring, resulting in unrealistic motion blur. Max and Lerner [ML85], and Pepper [Pep03], sort objects by depth, blur along their velocities, and compose the result in the final image, but this strategy fails when a scene invalidates the painter's visibility assumption. Per-pixel variants of this approach can reduce artifacts [Ros08, RMM10], especially when image velocities are dilated prior to sampling [Sou11, KS11], however this can corrupt background details and important motion features when multiple objects are present.

Motivated by recent tile-based, *single blur velocity* approaches [Len10, MHBO12, ZG12, Sou13] (see Section 3), we also dilate velocities to reason about large-scale motion behavior while sampling from the original velocity field to reason about the spatially-varying blur we apply. However, we additionally incorporate higher-order motion information and feature-aware sample weighting to produce results that are stable and robust to complex motion, effectively eliminating the artifacts present in the aforementioned "*single-velocity*" approaches. Specifically, we robustly handle:

- interactions between many moving and static objects,
- complex temporally-varying depth relationships,
- correct blurring regardless of object size or tile-alignment,
- feature-preservation for static objects/backgrounds.

We build atop existing state-of-the-art tile-based plausible motion blur approaches, detailed in Section 3, and present our more robust *multi-velocity* extensions in Section 4.

## 3 Tile-Based Dominant Velocity Filtering Overview

We base our approach on recent **single-velocity** techniques that combine phenomenological observations with sampling-aware reconstruction filters [Len10, MHBO12, ZG12, Sou13] (Figure 2) where: first, the image is split into $r \times r$ tiles for a maximum image-space blur radius $r$, ensuring that each pixel is influenced by at-most its (1-ring) tile neighborhood; second, each tile is assigned a *single* dominant neighborhood velocity; lastly, the pixel is blended with weighted taps along this dominant direction.

We review some notable details of this approach below:

- a tile's dominant velocity $\mathbf{v}_{\max}$ (Figure 2b; green) is computed in two steps: *maximum* per-pixel velocities are retained per-tile (Figure 2a,b; green, blue), and a 1-ring *maximum* of the per-tile maxima yields $\mathbf{v}_{\max}$,
- pixels are sampled *exclusively* along $\mathbf{v}_{\max}$, yielding high cache coherence but ignoring complex motions, and

© The Eurographics Association 2014.

- sample weighting uses a depth-aware metric that considers only velocity *magnitudes* at sample points.

The "'TileMax"' pass can be computed in two passes, one per image dimension, to greatly improve performance at the cost of a slight increase in memory usage. Samples are jittered to reduce (but not eliminate; see Section 4.5) banding, and samples are weighted (Figure 2c) to reproduce the following phenomenological motion effects:

1. a distant pixel/object blurring over the shading pixel,
2. transparency at the shading pixel from its motion, and
3. the proper depth-aware combination of effects 1 and 2.

These techniques generate plausible motion blur with a simple, high-performance post-process and have thus already been adopted in production game engines; however, the single dominant velocity assumption, coupled with the sample weighting scheme, results in noticeable visual artifacts that limit their ability to properly handle: overlapping objects that move in different directions, tile boundaries, and thin objects (see Figures 3, 7, 10, 12 and 16).

We identify, explain, and evaluate new solutions for these limitations. We follow the well-founded phenomenological methodology established by *single-velocity* approaches and other prior work [Len10, SAC*11, MHBO12, ZG12, Sou13], and our technique maintains high cache coherence and parallelizable divergence-free computation for an equally simple and high-performance implementation.

## 4 Stable and Robust Feature-Aware Motion Blur

We motivate our improvements by presenting artifacts in existing (single-velocity) approaches. We demonstrate clear improvements in visual quality with negligible additional cost. Furthermore, our supplemental video illustrates the stability of our solution under animation and on scenes with complex geometry, depth, velocity and texture.

### 4.1 Several Influential Motion Vectors

The *dominant velocity* assumption breaks down when a tile contains pixels with many different velocities, resulting in both an incorrect blur *inside* a tile and blur mismatches *between* tiles (see Figures 3 and 7). This can occur, for example, with rotating objects (see Figure 10), objects with features smaller than a tile (see Figure 12), or when the view and motion directions are (nearly) parallel (see Figure 7).

Specifically, by exploiting this assumption to reduce the sampling domain to 1D, *single-velocity* approaches weight samples along $\mathbf{v}_{max}$ according only to *the magnitudes* of their velocities, and not the *directions*. This can result in overblurred shading when samples (that lie along the dominant direction) should otherwise not contribute to the pixel but are still factored into the sum, especially if they are moving quickly (i.e., have large velocity magnitudes).

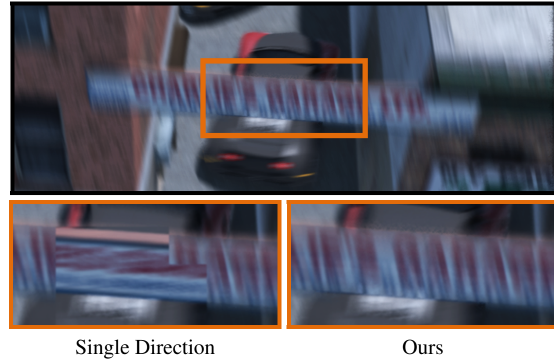


**Figure 3:** *Top: an animation with complex depth and motion relationships. The camera is moving upwards and turning while the car is making a sharp turn and moving up. Bottom: previous approaches (left) cannot handle these cases, resulting in distracting artifacts between tiles; our filter (right) correctly blurs and is temporally stable (see video).*

To reduce these artifacts we sample along a second, carefully chosen direction. Moreover, we split samples between these two directions according to the variance in the neighborhood's velocities, while also weighting samples using a locally-adaptive metric based on the deviation of sample velocities from the blur direction. This scheme better resolves complex motion details and still retains cache coherence by sampling along (fixed) 1D domains. Section 5 details our algorithm and we provide pseudocode in Appendix A.

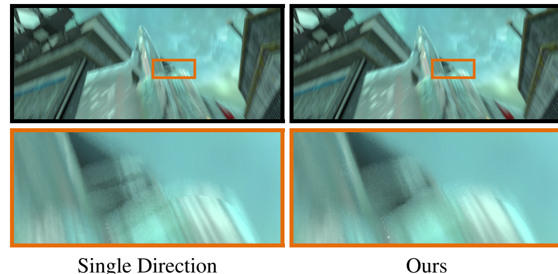


**Figure 4:** *Complex scene with high velocities in various directions. Even at just 15 samples per pixel, our algorithm produces more pleasing results than previous algorithms.*

**Local Velocity Sampling.** If a pixel's velocity differs significantly from the dominant direction, then it should also be considered during sampling. As such, we sample along both the pixel's velocity and the dominant velocity directions. This immediately improves the blur for scenes with complex pixel- and neighborhood-velocity relationships (e.g., Figure 3), however at the cost of increased noise since we are effectively halving the sampling rate in each 1D sub-domain.

If the pixel's velocity is small, this scheme effectively "wastes" all the samples placed on the second direction. In these cases, we replace the pixel's velocity with the velocity *perpendicular* to the dominant direction, sampling along this

© The Eurographics Association 2014.

new vector for half of the total samples (and in the dominant direction for the other half). This helps in scenarios where the dominant velocity entirely masks smaller velocities with different directions in the neighborhood. Here, the perpendicular direction serves as a "best-guess" to maximize the probability of sampling along an otherwise ignored, important direction. We also compute this perpendicular direction such that the angle between it and the velocity at the pixel is always less than $\pi$, inverting its direction if necessary. If no such important (albeit secondary) direction exists, then samples placed along this perpendicular direction are (at worst) also "wasted". This choice was motivated by the appearance of more visible artifacts due to "missed" samples (which these perpendicular samples greatly reduce) compared to the added noise from the unused samples. In more complex scenes, such as Figure 4, the number of wasted samples and their effect is vastly smaller, even at low sampling rates.

We ultimately combine the ideas of pixel ($\mathbf{v}(\mathbf{p})$) and perpendicular-dominant ($\mathbf{v}_{\max}(\mathbf{t})$) velocities at the shading pixel $\mathbf{p}$'s tile $\mathbf{t}$: we place a number (see below) of samples along the *center direction*, interpolating between the normalized directions of $\mathbf{v}(\mathbf{p})$ and $\mathbf{v}^{\perp}_{\max}(\mathbf{t})$ as the pixel's velocity diminishes past a minimum user threshold $\gamma$ (see Figure 5),

$$\mathbf{v}_c(\mathbf{p}) = \operatorname{lerp}\left(\mathbf{v}(\mathbf{p}), \mathbf{v}^{\perp}_{\max}(\mathbf{t}), (\|\mathbf{v}(\mathbf{p})\| - 0.5)/\gamma\right) . \quad (1)$$

Sampling along both $\mathbf{v}_{\max}(\mathbf{t})$ and $\mathbf{v}_c(\mathbf{p})$ ensures that each sample contributes usefully to the blur, better capturing complex motions. This approach remains robust to low (or zero, for static objects; see Figures 3 and 7) pixel velocities.

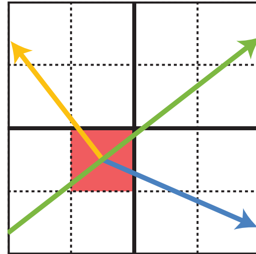

**Figure 5:** *Sampling directions $\mathbf{v}_{max}$ (green), $\mathbf{v}^{\perp}_{max}$ (yellow) and $\mathbf{v}_c(\mathbf{p})$ (blue).*

The number of samples we place along $\mathbf{v}_c$, and their weights (both along $\mathbf{v}_c$ and $\mathbf{v}_{\max}$), can significantly impact the final appearance and quality. We address the problems of assigning samples to each direction, and weighting them, separately: first, we detail the sample distribution over the directions; later, we present a more accurate scheme for weighting their contributions. Our final sampling scheme is robust to complex scenes and stable under animation.

**Tile Variance for Sample Assignment.** We first address the segmentation and assignment of samples to $\mathbf{v}_c$ and $\mathbf{v}_{\max}$. In cases where the dominant velocity assumption holds, we should sample *exclusively* from $\mathbf{v}_{\max}$, as the standard *single-velocity* approaches [Len10, MHBO12, ZG12, Sou13] do; however, it is rare that this assumption holds *completely* and we would like to find the ideal assignment of samples to the two directions in order to simultaneously capture more complex motions while reducing noise. This amounts to minimizing the number of "wasted" samples. To do so, we propose a simple variance estimation metric.

At each tile, we first compute the angular variation between a tile's and its neighbor's maximum velocities as

$$\upsilon(\mathbf{t}) = 1 - \frac{1}{|\mathcal{N}|} \sum_{\bar{\mathbf{t}} \in \mathcal{N}} \operatorname{abs}\left[\mathbf{v}_{\max}(\mathbf{t}) \cdot \mathbf{v}_{\max}(\bar{\mathbf{t}})\right] , \quad (2)$$

where $\mathcal{N}$ is the (1-ring) neighborhood tile set around (and including) $\mathbf{t}$. The absolute value prevents opposed-sign products from canceling themselves. The variance $0 \leq \upsilon \leq 1$ is larger when neighboring tile velocities differ from $\mathbf{v}_{\max}(\mathbf{t})$. As such, we can use $\upsilon(\mathbf{t})$ to determine the number of samples assigned to $\mathbf{v}_c$ and $\mathbf{v}_{\max}$ with: $\upsilon \times N$ assigned to $\mathbf{v}_c$ and $(1-\upsilon) \times N$ assigned to $\mathbf{v}_{\max}$ (Figure 6).

This metric works well in practice, reducing visual artifacts near complex motion and reverting to distributing fewer samples along $\mathbf{v}_c$ when the motion is simple; however, we require an extra (reduced-resolution) render pass (and texture) to compute (and store) $\upsilon$, which is less than ideal for pipeline integration (albeit with negligible performance impact).

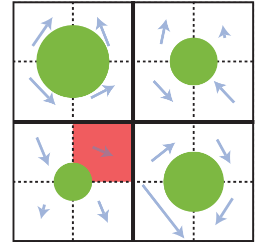

**Figure 6:** *Variance (green) for tiles neighboring the pixel $\mathbf{p}$ (red).*

While the variance-based sample assignment does improve the quality of the result, conservatively splitting our samples evenly between $\mathbf{v}_c$ and $\mathbf{v}_{\max}$ (i.e., $N/2$ samples for each) generates roughly equal quality results (see Figure 7).

We note, however, that the *weighting* of each sample plays a sizable role in both the quality of the final result and the ability to properly and consistently reconstruct complex motion blurs. We detail our new weighting scheme and contrast it to the scheme used in prior *single-velocity* approaches.

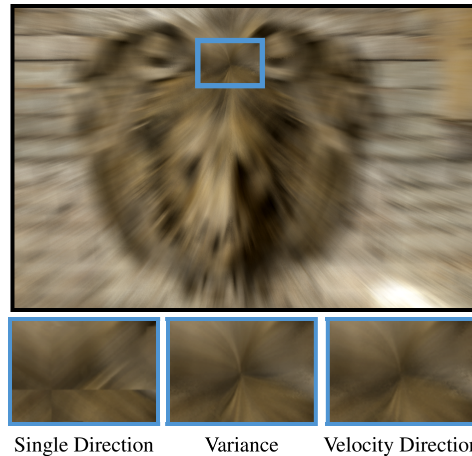


**Figure 7:** *The lion in Sponza moving directly towards the viewer. Bottom: single-velocity results (left), our variance-based sample distribution (middle) conservative sample distribution (right) both with $\mathbf{v}_{max}$ and $\mathbf{v}_c$ direction sampling.*

© The Eurographics Association 2014.

**Feature-Aware Sample Weights.** *Single-velocity* approaches compute weights by combining two metrics:

- depth difference between the sampled position and $\mathbf{p}$, and
- the *magnitude* of the velocity at the sample point and at $\mathbf{p}$.

This does not consider that samples can still fall on objects that move in directions different than $\mathbf{v}_{\max}$ (and even $\mathbf{v}_c$). We instead additionally consider the velocity *direction* at samples when computing their weights. This yields a scheme that more appropriately adapts to fine-scale motions, without complicating the original scheme. Specifically, we will consider the dot product between blurring directions and sampled velocity directions. Recalling the three phenomenological motion blur effects in Section 3, we modify the weights corresponding to each component as follows:

1. the contribution of distant objects blurring onto $\mathbf{p}$, along the sampling direction, are (additionally) weighted by the dot product between the sample's velocity and the sampling direction (either $\mathbf{v}_{\max}$ or $\mathbf{v}_c$, as discussed earlier); here, the weight accounts for the color that blurs from the distant sample onto $\mathbf{p}$ and, as such, can be adjusted as the sample velocity $\mathbf{v_s}$ differs from the blurring direction,
2. the transparency caused by $\mathbf{p}$'s blur onto its surrounding is (additionally) weighted by the dot product of its velocity (actually, $\mathbf{v}_c$) and the dominant blur direction $\mathbf{v}_{\max}$; here, the total weight models the visibility of the background behind the pixel/object at $\mathbf{p}$ and, as such, if $\mathbf{v}_c$ differs from $\mathbf{v}_{\max}$, the contribution should reduce appropriately, and
3. the *single-velocity* approaches include a correction term to model the combination of the two blurring effects above and account for object edge discontinuities; we (additionally) multiply this weight by the *maximum* of the two dot products above; here, the maximum is a conservative estimate that may slightly oversmooth such edges. such edges.

These simple changes (see pseudocode in Appendix A) significantly improve quality within and between tiles, particularly when many motion directions are present (Figure 7).

## 4.2 Tile Boundary Discontinuities

The tile-based nature of *single-velocity* approaches, combined with their dependence on a single dominant velocity, often leads to distracting tile-boundary discontinuities when blur directions vary significantly between tiles (see Figure 10). This occurs when $\mathbf{v}_{\max}$ differs significantly between adjacent tiles and, since the original approach samples *exclusively* along $\mathbf{v}_{\max}$, neighboring tiles are blurred in completely different directions. Our sampling and weighting approaches (Section 4.1) already help to reduce this artifact, but we still sample from $\mathbf{v}_{\max}$ (albeit not exclusively, and with different weights), and so we remain sensitive (to a lesser extent) to quick changes in $\mathbf{v}_{\max}$ (see Figures 9 and 10).

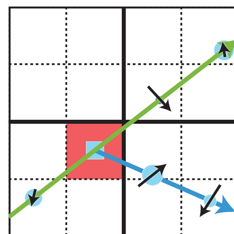

**Figure 8:** *Adapting per-sample weights according to the fine-scale velocity information.*

It is important to note that, while seemingly intuitive, linearly interpolating neighborhood velocities along tile edges produces incorrect results. Indeed, not only is the interpolation itself ill-defined (e.g., interpolating two antiparallel velocities requires special care), but more importantly, two objects blurring on one another with different motion directions is not equivalent to blurring along the average of their directions. The latter causes distracting "wavey" artifacts and often exaggerates, as opposed to masking, tile boundaries.

To further reduce this artifact, we stochastically offset our `NeighborMax` maximum neighborhood velocity texture lookup for pixels near a tile edge. This trades banding for noise in the image near tile boundaries (ie. edges in the $\mathbf{v}_{\max}$ buffer). Thus, the probability of sampling along the $\mathbf{v}_{\max}$ of a neighboring tile falls off as a pixel's distance to a tile border increases (Figure 9); we use a simple linear fall-off with a controllable (but fixed; see Section 5) slope $\tau$.

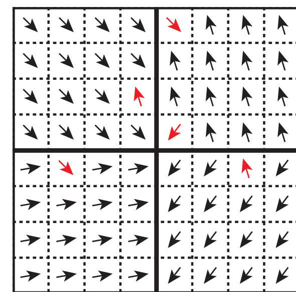

**Figure 9:** *We jitter the* $\mathbf{v}_{max}(\mathbf{t})$ *sample with higher probability closer to tile borders.*

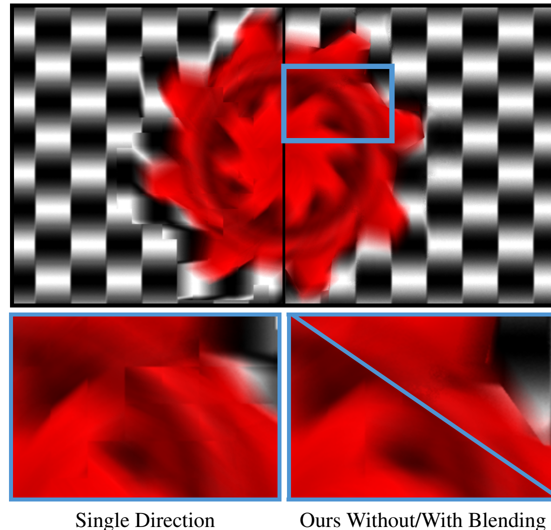


**Figure 10:** *Tile edge artifacts. Bottom: single-velocity blurring (left) results in disturbing tile-edge artifacts that are reduced, in part, using multi-direction sampling (right; bottom) and, in full, with stochastic* $\mathbf{v}_{max}$ *blending (right; top).*

© The Eurographics Association 2014.

### 4.3 Preserving Thin Features

*Single-velocity* filtering ignores the (jittered) mid-point sample closest to the pixel and instead explicitly weighs the color at **p** by the *magnitude* of the **p**'s velocity, without considering relative depth or velocity information (as with the remaining samples). This was designed to retain some of a pixel's original color regardless of the final blur, but it is not robust to scenes with thin features or large local depth/velocity variation. Underestimating this *center weight* causes thin objects to disappear or "ghost" (see Figure 12). Furthermore, the weight is not normalized and so its effect reduces as $N$ increases. These artifacts are distracting and unrealistic, and the weight's dependence on $N$ makes it difficult to control.

We instead set this *center weight* as $w_p = \|\mathbf{v}_c\|^{-1} \times N/\kappa$, where $\kappa$ is a user-parameter to bias its importance. We use $\kappa = 40$ in all of our results (see Section 5 for all parameter settings). The second term in $w_p$ serves as a pseudo-energy conservation normalization, making $w_p$ robust to varying sampling rates $N$ (unlike previous *single-velocity* approaches). Lastly, we do not omit the mid-point sample closest to **p**, treating it just as any other sample and applying our modified feature-aware sampling scheme (Section 4.1). As such, we also account for relative velocity variations at the midpoint, resulting in plausible motion blur robust to thin features and to different sampling rates (see Figure 12).

### 4.4 Neighbor Blurring

Both our approach and previous *single-velocity* approaches rely on an efficient approximation of the dominant neighborhood velocity $\mathbf{v}_{\max}$. We present a modification to McGuire et al.'s [MHBO12] scheme that increases robustness by reducing superfluous blur artifacts present when the $\mathbf{v}_{\max}$ estimate deviates from its true value. Specifically, the `NeighborMax` pass in [MHBO12] conservatively computes the maximum velocity using the eight neighboring tiles (see Section 3 and Figure 2b), which can potentially result in an overestimation of the actual maximum velocity affecting the central tile.

We instead only consider a diagonal tile in the $\mathbf{v}_{\max}$ computation if its maximum blur direction would in fact affect the current tile. For instance, if the tile to the top-left of the central tile has a maximum velocity that does not point towards the middle, it is not considered in the $\mathbf{v}_{\max}$ computation (Figure 11). The reasoning is that slight velocity deviations at on-axis tiles can result in blurs that overlap the middle tile, however larger deviations are required at the corner (off-axis) tiles. The impact of this modification is minor in practice due to the rare circumstances, but it still improves the worst case estimate.

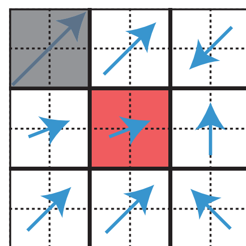

**Figure 11:** *Off-axis neighbors are only used for $\mathbf{v}_{max}$ computation if they may blur over the central tile.*

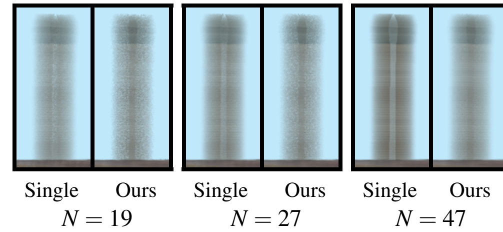


**Figure 12:** *Thin objects like the flagpole in Sponza ghost, to varying degrees depending on the sampling rate, with the single-velocity approach. Our approach resolves these details and is robust to changes in the sampling rate.*

### 4.5 Stochastic Noise and Post-Process Anti-Aliasing

We have discussed how to choose the direction along which to place each sample (either $\mathbf{v}_{\max}$ or $\mathbf{v}_c$) as well as how to weight them (Section 4.1), however we have not discussed *where* to place samples along the 1D domain. Numerical integration approaches are sensitive to sample distribution and, in the case of uniform samples distributed over our 1D domain, the quality of both ours and the *single-velocity* techniques can be significantly influenced by this choice. McGuire et al. [MHBO12] use equally-spaced uniform samples and jitter the pattern at each pixel using a hashed noise texture. This *jittered uniform* distribution has been recently analyzed in the context of 1D shadowing problems with linear lights [RAMN12] where it was proven to reduce variance better than low-discrepancy and stratified sampling.

We modify this strategy for motion blur integration and use a deterministic Halton sequence [WLH97] to jitter the per-pixel sample sets with a larger maximum jitter value $\eta$ (in pixel units), which improves the quality of the results (any other low discrepancy set could be used). Furthermore, as discussed earlier, we do not ignore the central sample closest to **p** and, as such, properly take its relative depth and local-velocity into account during weighting. Finally, we reduce the jitter value depending on the sampling rate, which reduces noise at higher $N$, without causing any banding, by multiplying $\eta$ by $\phi/N$, where $\phi$ is a user-determined constant which affects the "baseline" jitter level.

**Noise Patterns and Post-Process Anti-Aliasing.** The noise patterns produced by our stochastic integration are well-suited as input to post-processed screen-space anti-aliasing methods, such as FXAA. Specifically, per-pixel jitter ensures that wherever there is residual noise, it appears as a high-frequency pattern that triggers the antialiasing luminance edge detector (Figure 13, top right). The post-process antialiasing then blurs each of these pixels (Figure 13, bottom left), yielding a much smoother result than the unfiltered

© The Eurographics Association 2014.

image (Figure 13, top left). Stochastic $\mathbf{v}_{max}$ blending (Section 4.2) is compatible with this effect.

Moreover, it is possible to maximize the noise smoothing properties of post-processed antialiasing by feeding a maximum-intensity, pixel-sized checkerboard to the edge detector's luminance input. This causes the antialiasing filter to detect edges on all motion blurred pixels, thus suppressing residual motion blur sampling noise (Figure 13, bottom right). While this scheme could be improved, for instance by only blurring in the direction of motion, it has the significant advantage of being a trivial modification that leverages modern rendering pipelines (where post-processed antialiasing is common) with negligible performance cost.

## 5 Implementation and Results

All our results are captured live at $1280 \times 720$ (with the exception of the teapot scene at $1920 \times 1080$) on an Intel Core i7-3770K and an NVIDIA GTX780. We recommend that readers digitally zoom-in on our (high-resolution) results to note fine-scale details. Our performance results are detailed in Table 1. We use the same parameter settings for all our scenes: $\{N, r, \tau, \kappa, \eta, \gamma, \phi\} = \{35, 40, 1, 40, 0.95, 1.5, 27\}$. All intermediate textures, `TileMax`, `NeighborMax` [MHBO12] and `TileVariance` (Section 4.1), are stored in `UINT8` format and sampled using nearest neighbor interpolation, except for `TileVariance` that required a bilinear interpolant to eliminate residual tile boundary artifacts.

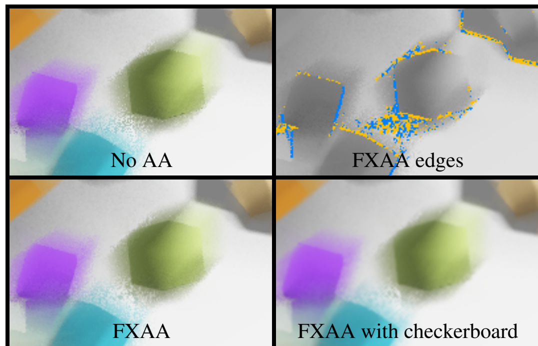


**Figure 13:** *Our noise is well-suited for standard post-process FXAA edge-detection, and we can further "hint" FXAA using a pixel-frequency luminance checkerboard.*

| $N$ | GTX780 720p | GTX780 1080p | GT650M (battery) 720p | GT650M (battery) 1080p |
|---|---|---|---|---|
| 19 | 1.11 (0.93) | 1.96 (2.06) | 6.32 (5.79) | 14.15 (13.00) |
| 27 | 1.41 (1.19) | 3.17 (2.71) | 9.19 (8.02) | 20.59 (17.99) |
| 35 | 1.68 (1.52) | 3.77 (3.44) | 11.59 (10.25) | 25.97 (23.06) |
| 47 | 2.18 (2.02) | 4.91 (4.54) | 15.26 (13.61) | 34.18 (30.52) |

**Table 1:** *Our performance results (in miliseconds) across resolutions, sample counts, and platforms in the Sponza scene. The camera is undergoing translation and all pixels are covered by the scene in motion. Parenthetical values are from our single-direction reference implementation.*

We note the importance of properly quantizing and encoding the per-pixel velocity when using integer buffers for storage (as is typically done in e.g. game engines). Specifically, a limitation in the encoding used in McGuire et al.'s implementation, $\texttt{V[x,y]} = \mathbf{v}(\mathbf{p}_{x,y})/2r + 0.5$, is that the $x$ and $y$ velocity components are clamped *separately* to $\pm r$, causing large velocities to only take on one of four possible values: $(\pm r, \pm r)$. We instead used a similar computation to that of the article, which normalizes according to the range $[-r, r]$, with $\varepsilon = 10^{-5}$ and $E$ the exposure time in seconds:

$$\texttt{V[x,y]} = \frac{\mathbf{v}(\mathbf{p}_{x,y})}{2r} \times \frac{\max(\min(|\mathbf{v}(\mathbf{p}_{x,y})| \times E, r), 0.5)}{|\mathbf{v}(\mathbf{p}_{x,y})| + \varepsilon} + 0.5\ .$$

We modify the *continuous depth comparison* function (`zCompare`) used by McGuire et al. to better support depth-aware fore- and background blurring as follows: instead of using a hard-coded, *scene-dependent* depth-transition interval, we use the relative depth interval

$$\texttt{zCompare}[z_a, z_b] = \min\left(\max\left(0, 1 - \frac{(z_a - z_b)}{\min(z_a, z_b)}\right), 1\right)\ ,$$

where $z_a$ and $z_b$ are both depth values. This relative test has important advantages: it works on values in a scene-independent manner, e.g. comparing objects at 10 and 20 z-units will give similar results to objects at 1000 and 2000 z-units. This allows smooth blending between distant objects, compensating for their reduced on-screen velocity coverage, all while remaining robust to arbitrary scene scaling.

Motivated by Sousa's [Sou13] endorsement of McGuire et al.'s *single-velocity* implementation, which has already been used in several game engines, all our comparisons were conducted against an optimized version of the open source implementation provided by McGuire et al., with both our Halton jittering scheme and our velocity encoding. While the variance-based $\upsilon$ metric for distributing samples between $\mathbf{v}_{max}$ and $\mathbf{v}_c$ yields slightly improved results over e.g. a 50/50 sample distribution, we observe no perceptual benefit in using $\upsilon$ during interactive animation; as such, we disabled this feature in all our results (except Figure 7).

**Post-Process Depth of Field.** Earlier, we discussed our approach's integration with FXAA, a commonly used post-process in modern game engines. Another commonly used post-process effect is depth of field (DoF); however, the combination of DoF and motion blur post-processes is a subject of little investigation. We implemented the post-process DoF approach of Gilham in ShaderX5 [Eng06] and briefly discuss its interaction with our motion blur filter. Specifically, the correct order in which to apply the two filters is not immediately apparent. We experimented with both options and illustrate our results in an especially difficult scene, as far as motion and DoF complexity are concerned: three

© The Eurographics Association 2014.

distinctly colored wheels, each undergoing different translational motion, at three different depths, and with the focus on the green middle wheel (see Figure 14).

The differences are subtle and our experiments far from comprehensive, however we note (somewhat counter-intuitively) that applying DoF *after* motion blur yields fewer visible edges in blurred regions and sharper features on in-focus regions. Applying DoF after motion blur has the added benefit of further blurring our noise artifacts.

## 6 Limitations and Future Work

While our approach addresses many of limitations, there remain several interesting avenues for future work.

### 6.1 Noise

The most prevalent artifact in our technique is noise. In most circumstances, we explicitly chose to trade banding for noise. We outline some possible solutions worth investigating that are outside the scope of our work.

**Single-velocity Fallback.** In images dominated by a single coherent velocity, e.g. Figure 13, our approach uses roughly twice as many samples for equal image quality compared to previous work. Correcting this behavior by, for example, dynamically adjusting the sampling directions according to the velocity spread in a scene is one potential avenue of future work. Our tile variance experiments (Section 4.1) could be repurposed to inform the algorithm on the circumstances where the variance is low enough to only sample in one direction, as opposed to two. However, such adaptive sampling can have unexpected repercussions, particularly on the center sample weight: when half the samples are wasted, the total weight of the sum is effectively halved, but if those samples are dynamically adjusted such that none are wasted, the relative weight of the center sample is halved. Any adaptive scheme must therefore also adjust the central weight to compensate for variations in the total weight.

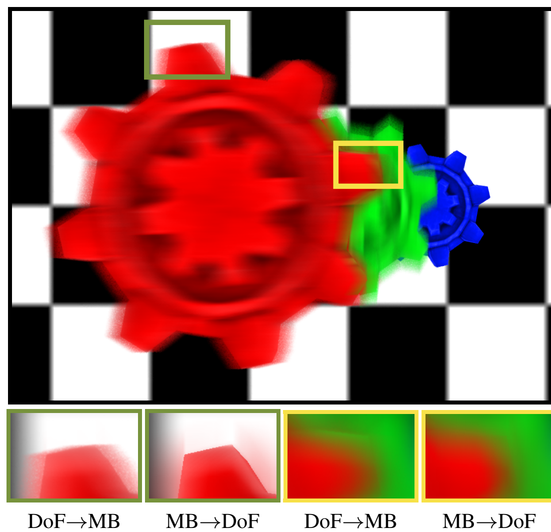


**Figure 14:** *DoF and MB stress test. Top: MB only. Bottom: Comparison of motion blur and depth of field ordering. We note slightly better results when DoF is applied after MB.*

**Adaptive Sampling Rate.** Similarly speaking, adjusting the accumulation sample count, perhaps according to the magnitude of the velocity at the tile or pixel, would reduce the cost of the summation when the motion is relatively small, thus allowing larger motion to be oversampled. This could result in a uniform amount of noise throughout the image, whereas our current implementation is noisier in regions of larger velocity. Here, for performance purposes, it would be critical to prevent branching within the same GPU warp.

**Filtering.** We briefly investigate post-process FXAA filtering in Section 4.5, however a post-process AA filter designed from scratch for motion blur, e.g. one that re-blurs in the direction of motion (as in [Sou08]), could potentially help reduce the sampling rate in each pass.

### 6.2 Other Limitations

**Acceleration to/from Rest.** Albeit less common a scenario, a distracting visual artifact can appear under very high camera exposure time: an object under acceleration will blur beyond its initial location because our approach assumes constant object velocity.

**Velocity Interpolation.** We interpolate velocities in order to smooth transitions during motion. For slow transitions, the effect can be visually distracting as the blur begins in a direction that is completely unrelated to the motion.

**Non-linear Motion.** As with previous work, we assume linear motion. While this assumption often holds, extreme rotational motion or complex deformation (e.g., Figure 15), especially with high exposure times, may result in artifacts.

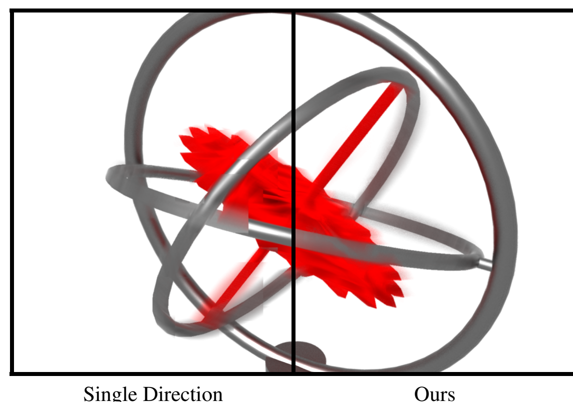


**Figure 15:** *Rotation stress test: multiple overlapping rotational velocities are a challenging scenario.*

© The Eurographics Association 2014.

### 6.3 Future Work

**Time Component.** As mentioned earlier, our algorithm, while stable across frames, does not explicitly treat the temporal dimension. Doing so could have multiple advantages, e.g. using previous frame data to reduce noise in the current frame, but would also introduce several complex issues when involving advanced animation that do not follow predictable, linear motion paths. An explicit time component would also allow for acceleration-aware filtering, which could correct overblurring in such situations.

**Shadows.** As a screen-space postprocess, our algorithm is oblivious to any shadowing techniques and their interactions with objects in the scene. An interesting open problem is the proper incorporation of blurred shadows according to the motion of blockers and light sources.

**Simultaneous DoF.** Combining motion blur and depth of field into a single algorithm is not a new idea and has been explored in e.g. [MESL10]. While we briefly investigate the interaction between depth of field and motion blur, we do not attempt to combine both effects into a single algorithm.

## 7 Conclusion

We presented a high-performance post-process motion blur filter that is robust to scenes with complex inter-object motions and fine-scale details. We demonstrated our approach on several such scenes with clear quality benefits, both in images and in animation (see the supplemental video), and at a negligible cost compared to the state-of-the-art. Our approach is temporally coherent, easy to integrate, and readily compatible with other commonly used post-process effects.

### Acknowledgments

We thank the reviewers for their helpful comments. This work was supported by NSERC Discovery and FQRNT New Researcher Grants (Nowrouzezahrai), as well as an FQRNT MSc Scholarship (Guertin). We also thank NVIDIA for equipment donations. The NeighborMax tile culling algorithm was contributed by Dan Evangelakos at Williams College. Observations about FXAA, the use of the luminance checkerboard, and considering the central velocity in the gather kernel are due to our colleagues at Vicarious Visions: Padraic Hennessy, Brian Osman, and Michael Bukowski.

### References

[AMMH07] Akenine-Möller T., Munkberg J., Hasselgren J.: Stochastic rasterization using time-continuous triangles. In *Graphics Hardware* (2007), Eurographics, pp. 7–16. 2

[Coo86] Cook R. L.: Stochastic sampling in computer graphics. *ACM Trans. Graph. 5*, 1 (1986), 51–72. 1, 2

[Eng06] Engel W.: *Shader X5: Advanced Rendering Techniques*. Charles River Media, MA, USA, 2006. 1, 7

[ETH*09] Egan K., Tseng Y.-T., Holzschuch N., Durand F., Ramamoorthi R.: Frequency analysis and sheared reconstruction for rendering motion blur. *ACM Trans. Graph. 28*, 3 (2009). 1, 2

[FLB*09] Fatahalian K., Luong E., Boulos S., Akeley K., Mark W. R., Hanrahan P.: Data-parallel rasterization of micropolygons with defocus and motion blur. In *High Performance Graphics* (2009), ACM, pp. 59–68. 2

[HJW*08] Hachisuka T., Jarosz W., Weistroffer R. P., Dale K., Humphreys G., Zwicker M., Jensen H. W.: Multidimensional adaptive sampling and reconstruction for ray tracing. *ACM Trans. Graph. 27*, 3 (2008). 2

[KS11] Kasyan N., Schulz N.: Secrets of cryengine 3 graphics technology. In *SIGGRAPH Talks*. ACM, 2011. 2

[LAC*11] Lehtinen J., Aila T., Chen J., Laine S., Durand F.: Temporal light field reconstruction for rendering distribution effects. *ACM Trans. Graph. 30*, 4 (2011), 55. 1, 2

[Len10] Lengyel E.: Motion blur and the velocity-depth-gradient buffer. In *Game Engine Gems*, Lengyel E., (Ed.). Jones & Bartlett Publishers, March 2010. 2, 3, 4

[Lot09] Lottes T.: Fast approximate anti-aliasing (fxaa). 2009. URL: http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf. 1

[Lov05] Loviscach J.: Motion blur for textures by means of anisotropic filtering. EGSR, pp. 105–110. 2

[MESL10] McGuire M., Enderton E., Shirley P., Luebke D. P.: Real-time stochastic rasterization on conventional GPU architectures. In *High Performance Graphics* (2010). 2, 9

[MHBO12] McGuire M., Hennessy P., Bukowski M., Osman B.: A reconstruction filter for plausible motion blur. In *I3D* (2012), pp. 135–142. 1, 2, 3, 4, 6, 7, 10

[ML85] Max N. L., Lerner D. M.: A two-and-a-half-d motion-blur algorithm. In *Proc. of SIGGRAPH* (NY, 1985), ACM, pp. 85–93. 2

[NSG11] Navarro F., Serón F. J., Gutierrez D.: Motion blur rendering: State of the art. *Computer Graphics Forum 30*, 1 (2011), 3–26. 2

[ODR09] Overbeck R. S., Donner C., Ramamoorthi R.: Adaptive wavelet rendering. *ACM Trans. Graph. 28*, 5 (2009). 2

[Pep03] Pepper D.: Per-pixel motion blur for wheels. In *ShaderX6*, Engel W., (Ed.). Charles River Media, 2003. 2

[RAMN12] Ramamoorthi R., Anderson J., Meyer M., Nowrouzezahrai D.: A theory of monte carlo visibility sampling. *ACM Transactions on Graphics* (2012). 6

[RMM10] Ritchie M., Modern G., Mitchell K.: Split second motion blur. In *SIGGRAPH Talks* (NY, 2010), ACM. 2

[Ros08] Rosado G.: Motion blur as a post-processing effect. In *GPU Gems 3*. Addison-Wesley, 2008, pp. 575–581. 2

[SAC*11] Shirley P., Aila T., Cohen J. D., Enderton E., Laine S., Luebke D. P., McGuire M.: A local image reconstruction algorithm for stochastic rendering. In *I3D* (2011), pp. 9–14. 2, 3

[SD12] Sen P., Darabi S.: On filtering the noise from the random parameters in monte carlo rendering. *ACM Trans. Graph. 31*, 3 (2012), 18. 2

[Sou08] Sousa T.: Crysis Next Gen Effects, February 2008. Presentation at GDC 2008. 8

[Sou11] Sousa T.: Cryengine 3 rendering techniques. In *Microsoft Game Technology Conference*. August 2011. 2

[Sou13] Sousa T.: Graphics gems from cryengine 3. In *ACM SIGGRAPH Course Notes* (2013). 1, 2, 3, 4, 7

© The Eurographics Association 2014.

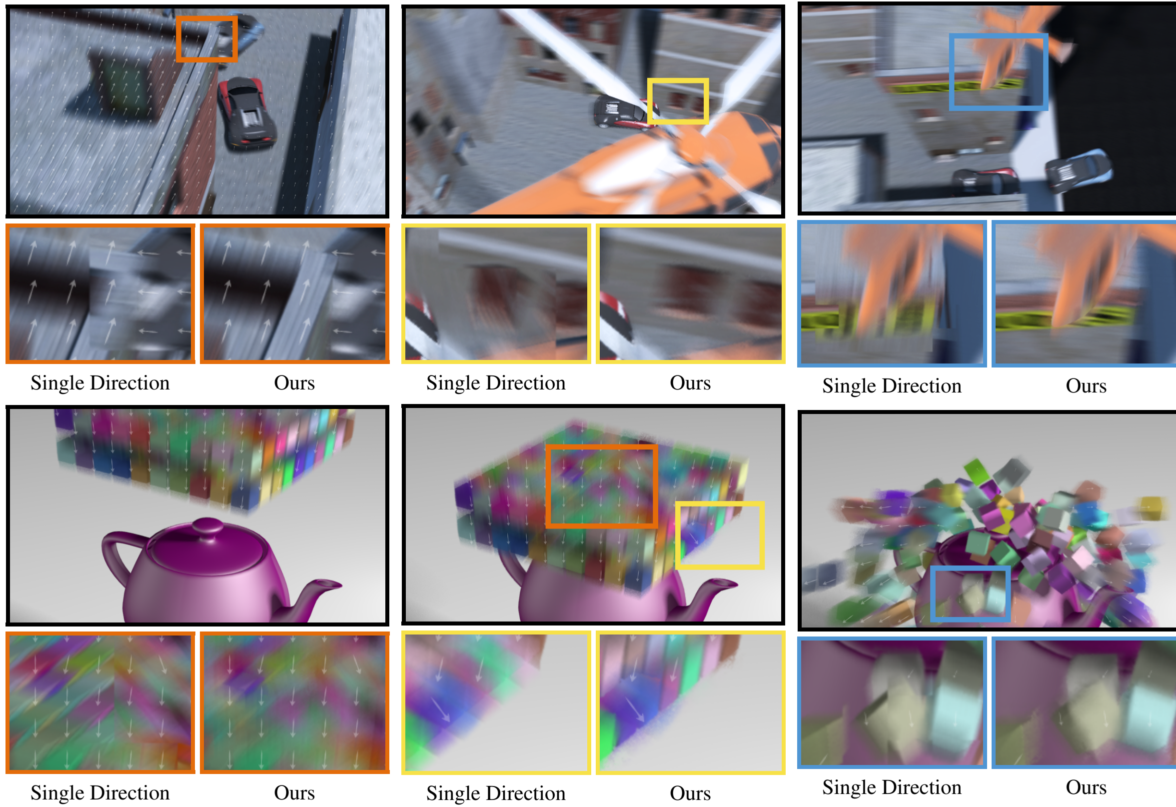


**Figure 16:** *Our results are temporally stable on complex scenes where previous approaches suffer from distracting artifacts.*

[TBI03] TATARCHUK N., BRENNAN C., ISIDORO J. R.: Motion blur using geometry and shading distortion. In *ShaderX2: Shader Prog. Tips & Tricks with DirectX 9.0*, Engel W., (Ed.). 2003. 2

[WLH97] WONG T.-T., LUK W.-S., HENG P.-A.: Sampling with hammersley and halton points. *J. Graph. Tools* (1997). 6

[ZG12] ZIOMA R., GREEN S.: Mastering DirectX 11 with Unity, March 2012. Presentation at GDC 2012. 2, 3, 4

## A Pseudocode

Our pseudocode relies on the following operators: `sOffset` jitters a tile lookup (but never into a diagonal tile), `rnmix` is a vector lerp followed by normalization, `norm` returns a normalized vector, $\lfloor \cdot \rfloor$ returns the integer component, and & denotes bitwise **and**. Unless otherwise specified, we use the notation/functions of McGuire et al. [MHBO12]. Our function returns **four** values: the filtered color, and a luminance for an (optional) FXAA post-process.

```
function filter(p):
  let j = halton(−1, 1)
  let vmax = NeighborMax[p/r + sOffset(p, j)]
  if (‖vmax‖ ≤ 0.5)
    return (color[X], luma(color[X]))

  let w_n = norm(vmax), v_c = V[p], w_p = (−w_ny, w_nx)
  if (w_p · v_c < 0) w_p = −w_p
  let w_c = rnmix(w_p, norm(v_c), (‖v_c‖ −0.5)/γ)

  // First integration sample: p with center weight
  let totalWeight = N/(κ × ‖v_c‖)
  let result = color[p] × totalWeight
  let j′ = j η φ / N

  for i ∈ [0, N)
    let t = mix(−1, 1, (i + j′ + 1)/(N + 1)) // jitter sample

    // Compute point S; split samples between {v_max, v_c}
    let d = v_c if i odd or v_max if i even
    let T = t × ‖vmax‖, S = ⌊t × d⌋ + p

    // Compute S's velocity and color
    let v_s = V[S], colorSample = color[S]

    // Fore- vs. background classification of Y w.r.t. p
    let f = zCompare(Z[p], Z[S])
    let b = zCompare(Z[S], Z[p])

    // Sample weight and velocity-aware factors (Sec. 4.1)
    // The length of v_s is clamped to 0.5 minimum during normalization
    let weight = 0, wA = w_c · d, wB = norm(v_s) · d

    // 3 phenomenological cases (Secs. 3, 4.1): Objects
    // moving over p, p's blurred motion, & their blending
    weight += f · cone(T, 1/‖v_s‖) × wB
    weight += b · cone(T, 1/‖v_c‖) × wA
    weight += cylinder(T, min(‖v_s‖, ‖v_c‖)) × max(wA, wB) × 2

    totalWeight += weight // For normalization
    result += colorSample × weight

  return (result/totalWeight, (p_x + p_y)&1)
```

© The Eurographics Association 2014.