

A Reconstruction Filter for Plausible Motion Blur

Morgan McGuire
NVIDIA and Williams College

Padraic Hennessy
Vicarious Visions

Michael Bukowski
Vicarious Visions

Brian Osman
Vicarious Visions

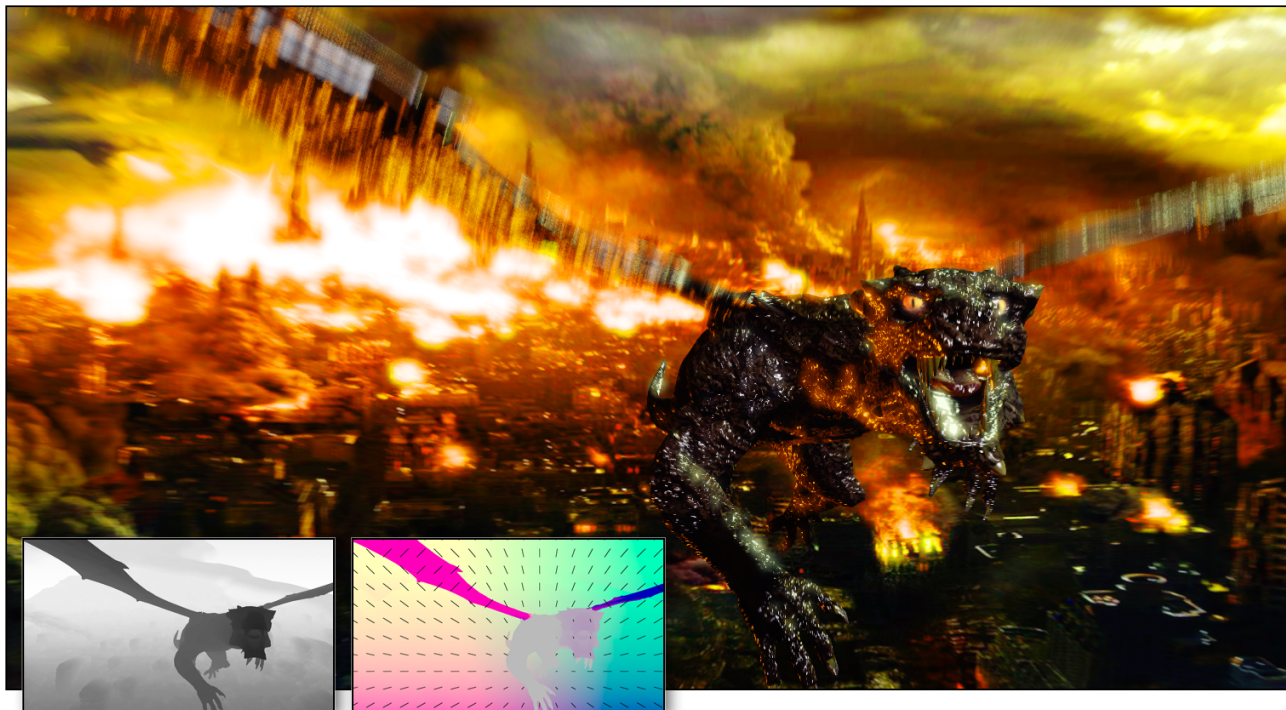


Figure 1: A dragon tracked by a dolly camera over a burning city, with motion blur result reconstructed in 3 ms at 1280×720 on GeForce 480 from a conventionally-rasterized framebuffer. This image contains both character and camera motion comprising zoom, pan, translation, deformation, and rotation. The full-resolution auxiliary depth and velocity input buffers are inset at reduced size. We visualize velocity with a coarse vector field over the fine velocity buffer observed when implementing the algorithm, in which $(r, g, b) = (dx/dt, dy/dt, 0)/(k + 0.5)$.

Abstract

This paper describes a novel filter for simulating motion blur phenomena in real time by applying ideas from offline stochastic reconstruction. The filter operates as a 2D post-process on a conventional framebuffer augmented with a screen-space velocity buffer. We demonstrate results on video game scenes rendered and reconstructed in real-time on NVIDIA GeForce 480 and Xbox 360 platforms, and show that the same filter can be applied to cinematic post-processing of offline-rendered images and real photographs. The technique is fast and robust enough that we deployed it in a production game engine used at Vicarious Visions.

1 Introduction and Related Work

A real camera integrates the incoming light over the exposure time at each pixel. This blurs images of moving objects along their image-space (screen-space) velocity vectors. We classify two dom-

inant approaches to capturing this effect for computer-generated images. **Plausible** post-processing methods seek to enhance images taken at a single instant in time with *phenomena* resembling motion blur. These are suitable for interactive applications because they run in real time, albeit at less-than-ideal quality (see section 1.1) because the input lacks information spanning the exposure. In contrast, **reconstruction** methods filter and decimate samples that span the exposure interval, attempting to invert the sampling operator and recover a physically-correct image of a moving scene. Classic reconstruction filters simply average all samples within a pixel or convolve against a fixed, narrow kernel [Cook et al. 1984]. Such filters are themselves fast, but they can only be applied in offline rendering because the input must be densely sampled, often by a stochastic process like ray tracing or micropolygon rasterization.

Recent physically-based **smart reconstruction** methods reason about the anisotropy of the underlying space-time imaging process [Egan et al. 2009; Lehtinen et al. 2011] or samples [Overbeck et al. 2009; Shirley et al. 2011; Sen and Darabi 2011]. These produce high-quality images from much lower input sample density than classic reconstruction, but the smart reconstruction itself may take seconds or minutes per frame. We note that these share characteristics of plausible methods—e.g., allow samples to contribute well beyond the borders of their own pixels and take the underlying sampling scheme and motion vectors into account.

We introduce a novel plausible-reconstruction algorithm for motion

blur in real-time applications like games. It borrows ideas from offline smart reconstruction of stochastic samples to increase image quality, but operates on images produced by typical video game renderers, i.e., an instantaneous exposures with one sample in the center of each pixel. It succeeds in cases where some previous plausible methods are limited or fail. Compared to full smart reconstruction, the quality is imperfect but the performance is exceptional. For example, when observed in a *still image* like figure 1, the new blur is limited by the discrete sampling scheme, lack of information behind occlusions, and a heuristic that assumes that a single motion vector dominates in every pixel neighborhood. Yet, it renders in about 3 ms on current hardware and when viewed in the intended context—*under animation*—noise and misblurring artifacts have relatively modest visual impact.

In addition to better performance than smart reconstruction and better quality than previous plausible methods, the new plausible reconstruction allows certain non-physical manipulations of velocity. We developed the algorithm in the context of a production game, a case where artistic control of visual impact and clarity of on-screen elements is essential.

Camera Parameters. When the camera tracks a fast-moving object, that object remains sharp and the rest of the scene blurs (figure 8). This is because the motion with which we are concerned is measured on the image plane, and is thus relative to the viewer’s own motion and orientation. Today’s games and film are rendered independent the viewer’s attention within the frame. The experience of viewing the rendered image differs from viewing the original scene with a naked eye if the camera and eye track different objects. This is perhaps one reason that film exposure times are much shorter than the frame display time—another is that moving objects simply look too blurry otherwise. Valve recommends using only 15% of the full-frame shutter and a maximum velocity of 4% of the screen width [Vlachos 2008]. All results in this paper were rendered with a 1/2 frame exposure to exaggerate blur for analysis.

We speculate that with sufficiently robust and low-latency eye tracking, it might be desirable as future work to perform motion blur based on the viewer’s actual attention. Alternatively, with an extremely high refresh rate (e.g., 1000 fps), one might forgo rendered motion blur altogether. However, motion blur, like occlusion and defocus, is a powerful tool for abstracting detail to direct the viewer’s attention elsewhere. For example, a director might want the audience to attend to the action hero on a motorcycle and not allow them to focus on buildings whipping past in the background.

1.1 Previous Real-Time Plausible Methods

We now describe the main ideas of plausible blur approaches designed for real-time rendering. We emphasize recent games and game engines for brevity; this is not an exhaustive survey.

Accumulation buffering averages multiple sub-exposure frames. This generates ideal results if given enough samples, but the cost of rendering each frame many times is prohibitive.

MotoGP 2 and *Split Second* [Ritchie et al. 2010] are racing games that **blur textures** on moving objects with multiple or anisotropic samples [Hargreaves 2004; Loviscach 2005]. This incorrectly leaves sharp silhouettes and texture seams of moving objects.

To blur object borders, one can **extrude or augment** moving object geometry, blur the shading and textures across them, and then alpha-blend [Wloka and Zeleznik 1995; Tatarchuk et al. 2003] during forward rendering. This requires near-perfect depth sorting and approximately convex geometry.

Max and Lerner [1985] were the first to render sharp objects to sep-

arate buffers and then **blur independently by object velocity and compose** the results in depth order. This was used in *Need for Speed 2* and presumably in *MotoGP '07* and *Pure* with Pepper’s [2008] specialization for wheels. It works best if the objects exhibit little self-occlusion and can be properly sorted in depth, conditions that often hold in racing games, but not for general scenes. Shimizu et al. [2003] describe a per-pixel variant for a single object.

The current state of the art is **per-pixel blur** computed by rendering a velocity buffer, and then at each pixel accumulating along its velocity neighboring samples at or behind it [Rosado 2007; Ritchie et al. 2010]. This leaves four problems with blurry objects: sharp silhouettes, background behind them is always blurry, they shrink because blur is limited to the original silhouette, and binary depth comparison underblurs slanted surfaces.

Velocity Dilation Methods. We now describe three previous methods for dilating the velocity buffer, which address some of the issues with direct per-pixel blur. Our blur also dilates velocity by computing the dominant neighborhood velocity. We are able to overcome some of the limitations of the techniques described below because we use dilated velocity only as a heuristic and accumulate blur along original velocity.

Green [2003] extrudes objects when rendering velocity. This correctly softens the borders of moving objects but also overblurs the background near them. The extrusion process resembles shadow volume generation methods and shares their limitations, e.g., it requires water-tight geometry with extra stitches along geometric creases and doesn’t work well with particles or alpha-cutouts.

The details of *Lost Planet*’s blur are unpublished, but it probably dilates the velocity buffer by rendering opaque line segments with depth in a low-resolution buffer, based on Sawada’s presentation [Sawada 2007]. It is possible that it uses a variation of Potmesil’s [1983] method, however artifacts observable in game are consistent with the former method (e.g., in <http://4gamer.net/news/image/2007.08/20070809235901.13big.jpg>).

Unreal Engine 3 convolves the velocity buffer [Epic Games 2010] with a gaussian. This avoids the extrusion limitations not the overblur problem. We note that the overblur looks much better in motion than in still images, but even in motion, creates a sense of an “invisible” buffer around each object. Component-wise blurring of vectors also bends their directions and may cause opposing velocity vectors to cancel, when in fact they describe the same point spread.

CryENGINE 3 separately computes object and camera motion blur, and then combines them under a depth mask. It simulates camera motion on a half-resolution image (like Valve’s technique [Vlachos 2008]). For object blur, it dilates velocity in four 1D diffusion passes [Sousa 2008; Sousa 2011; Kasyan et al. 2011] and then accumulates $S = 9$ sample taps on consoles and $S = 24$ taps on PC. As with other methods, this corrupts the direction of blur when multiple objects move and overblurs the background.

Other Approaches. We refer the interested reader to related techniques for analytic [Grant 1985; Sung et al. 2002; Gribel et al. 2010; Gribel et al. 2011], and non-photorealistic [Jones and Keyser 2005; Obayashi et al. 2005; Bouvier-Zappa et al. 2007] motion blur that are beyond this paper’s scope of game-like interactive applications with plausibly realistic appearance.

1.2 From Offline-Smart to Real-Time-Plausible Reconstruction

The structure of our blur is modeled on Shirley et al.’s [2011] smart reconstruction. We readily note that while Shirley et al.’s method

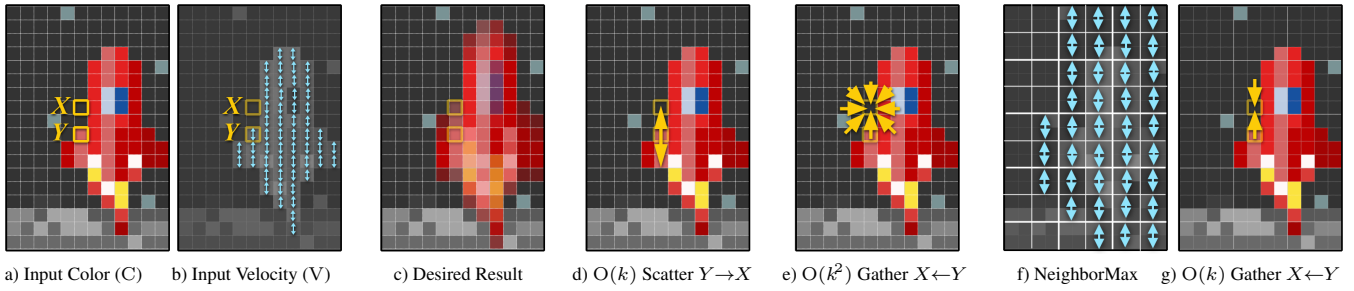


Figure 2: Blurring a red rocketship with vertical velocity by three different strategies: (d) line-scatter, (e) conservative disk-gather, and the heuristic (g) line-gather used by our algorithm. Cyan arrows indicate velocity. Yellow arrows denote memory operations affecting the final shade computed at X , which depends on the initial shade of Y . $k = 2$ for this example.

should converge to the physically-correct result as sample density grows, it is not sophisticated about the light field and sampling compared to other smart reconstructions methods. This simplicity is what makes it amenable to acceleration by our techniques.

Shirley et al.’s original method requires many stochastic samples as input at each pixel and must sort them in depth, so it would be very slow and difficult to apply on current rasterization hardware—it was framed as a proposal for new stochastic hardware. However, we observe that the two core ideas underlying their technique do not actually require dense or stochastic input: 1) Consider samples well-outside a pixel. 2) Categorize and combine samples along two discrete axes: static (i.e., sharp) vs. moving (i.e. blurry) and foreground vs. background. The Shirley et al. method must combine many (e.g., $S = 5 \times 5 \times 16 = 400$) filter taps at each pixel to reduce both sample noise and discrete classification. It implements the z -order categorization via a heavyweight sorting. Our new method instead implements these steps with continuous, order-independent sample classification for both blurriness and z -order. That enables motion blur reconstruction without any sorting and using relatively few ($S = 5$ to 15) filter taps.

2 Algorithm

2.1 Overview

Figure 3 shows the structure of the blur algorithm for input and output buffers of size $n = w \times h$. Input C is an instantaneous color image, e.g., as rendered by traditional rasterization. The screen-space velocity buffer V encodes the pixel offset of each point from the location it would have projected to in the previous frame, and Z

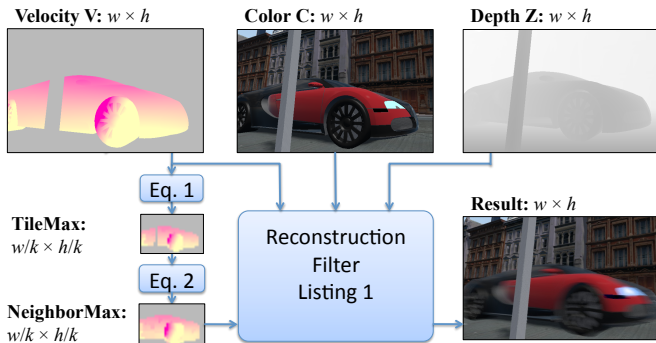


Figure 3: System diagram of the three input buffers and three filter passes for the blur algorithm.

stores camera-space depth.

The blur algorithm produces two small intermediate buffers from V and then reconstructs the output from these and all input buffers. The reconstruction filter (listing 1) has application-specific parameters for maximum motion blur radius k pixels and S reconstruction filter sample taps. We choose k based on the expected distance from the viewer to objects, velocities of objects, and exposure time, with the caveat that large values (e.g., 40 pixels) may adversely affect texture cache performance. We choose S as an odd integer based on the acceptable noise due to sample variance in the final image—which itself depends on artistic vision, display type, frame rate, scene texture, post-process filters, and scene local contrast. Unless otherwise noted, images in this paper were rendered with $k = 20$, $S = 15$ at 1280×720 .

Motivating Analysis Holding other aspects fixed, we hypothesize that the ratio k/S should be constant as resolution or velocity grows to maintain approximately constant image quality. With that simplifying assumption, we now sketch an asymptotic analysis. This explains why the blur algorithm is both well-suited to GPUs today and will scale to higher resolutions and more parallelism in the future. Experimental results in section 3 show that the constants abstracted by this analysis yield practical rendering times on current architectures.

The key idea of the blur algorithm is converting an $O(kn)$ -time synchronized *scatter* algorithm into an $O(kn/m)$ -time parallel *gather* algorithm for an m -way processor. Figure 2a shows a rocketship blasting off the moon, at 12×16 resolution for expository purposes. This represents an instantaneous-exposure image from classic deterministic rasterization. Subfigure (b) represents the velocity buffer. Consider pixel Y on the tail fin, which during the exposure time should contribute to five pixels, including X , as shown in (c).

Estimating motion blur by scattering as shown in (d) is trivial but inefficient: iterate over the input, replacing each point with a line segment, and then composite the results *taking depth-ordering into account* [Potmesil and Chakravarty 1983]. This is an $O(kn)$ -time solution. One can do something similar at a per-polygon level [Max 1990]. Scatter blur is inefficient on parallel rasterization architectures with wide memory busses, e.g., GPUs. Ordering requires synchronization, and the irregular output iteration limits the coherence and coalescing of memory traffic.

Any bounded-offset scatter operation can be converted to a parallel gather algorithm by simply gathering over a disk of the worst-case radius at each pixel. A conservative gathering implementation of motion blur (e) iterates over the output, gathering $O(k^2)$ input samples at each pixel but ignoring most of them because their velocities

are not directed towards the current pixel. This is an $O(k^2 n/m)$ -time solution.

The problem with the strawman disk-gather operation is that at each pixel X it must exhaustively search to find pixels like Y that might blur over it. To retain the parallelism and cache efficiency of gathering without the quadratic search cost, we compute a 1D gather footprint for each pixel based on the maximum velocity within a neighborhood, NeighborMax (f). A line-gathering filter (g) then needs only to search along NeighborMax $[X]$. We compute NeighborMax with two m -way parallel-gather operations on n/k^2 tiles, each of size k^2 pixels. The entire algorithm is $O(kn/m)$ -time and exhibits both high memory locality and parallel memory access.

NeighborMax provides a heuristic kernel, so in some cases the gather operation will fail to find all neighbors that should blur over a pixel. This results in an under-blurring artifact. The results section reports on several scenes constructed to stress this case and demonstrate the visual impact of the error, which we found negligible for our application.

2.2 Buffers

The (sharp) input color buffer C is a high-dynamic range framebuffer, before tone-mapping or gamma encoding; we use GL_RGB16F format. Our result images include typical game post-processing of bloom, tone-mapping, and FXAA [Lottes 2009] antialiasing *after* motion blur. If MSAA is used, C should be the post-resolve buffer with one sample per pixel. One can envision a future variation of the blur algorithm that directly reads MSAA samples, potentially increasing image quality at edges and reducing noise.

We use the SS_GBuffer shader from the G3D library [http://g3d.sf.net] to generate the velocity and linear Z buffers. The half-spread velocity in our equations is limited to the exposure time and clamped to maximum k , i.e.,

$$\begin{aligned}\vec{q}_X &= \frac{1}{2}(X - X') \cdot (\text{exposure time}) \cdot (\text{frame rate}) \\ V[X] &= \frac{\vec{q}_X \max(0.5\text{px}, \min(\|\vec{q}_X\|, k))}{\|\vec{q}_X\| + \varepsilon}\end{aligned}$$

where X' and X the image-space positions of a point in the previous and current frame. A point thus contributes to pixels on the line covering $X \pm V[X]$. We bias and scale V to store it in an GL_RG8 buffer. On consoles, we render V at low resolution to conserve fill-rate and memory. On more recent GPUs we render at full resolution to an additional draw buffer bound during the color rendering pass.

We implement Z as a GL_R16F linear camera-space z -buffer with small, negative near values and large, negative far values. Any depth encoding can be substituted with appropriate comparison signs and constants. TileMax and NeighborMax each have size $w/k \times h/k$ and follow the bit format of V . The output has size $w \times h$ and follows the bit format of C . The combined size of all data buffers is 15 MB at 720p.

2.3 Filter Passes

The algorithm makes three 2D passes. The first gathers the dominant velocity per tile into TileMax by eq. 1. The second computes the maximum velocity in any adjacent tile by eq. 2. Since $\|V[X]\| \leq k \forall X$, this guarantees that each pixel is aware of the most distant neighbor that can affect it when the dominant velocity assumption holds. Our vmax operator returns the vector with the largest magnitude rather than a per-component maximum.

$$\text{TileMax}[x, y] = \text{vmax}_{u \in [0, k]} \text{vmax}_{v \in [0, k]} (V[kx + u, ky + v]) \quad (1)$$

$$\text{NeighborMax}[x, y] = \text{vmax}_{u \in [-1, 1]} \text{vmax}_{v \in [-1, 1]} (\text{TileMax}[x + u, y + v]) \quad (2)$$

The third pass applies the reconstruction filter from listing 1 at full resolution. At each pixel X , it computes the contribution α_Y for each neighbor Y_i along the dominant velocity vector \vec{v}_N for the neighborhood. There are three terms in α_{Y_i} . The obvious one (case 1) is that Y contributes when it is blurry and passes in front of X . Note that in this case we implicitly assume that $V[Y]$ is along \vec{v}_N . This is a source of error, however we found that it was better to sometimes blur in the wrong direction than to not blur at all in that situation, e.g., by incorporating a factor like $|V[Y] \cdot \vec{v}_N|$.

In case 2, X itself is blurry so we should be able to see through it. Unfortunately, the background was occluded by X in C and is thus unknown. Even with stochastic input, this case could occur; here, Shirley et al.'s algorithm estimates the background by averaging every neighboring background sample, regardless of whether $V[Y]$ justifies that. We follow their rationale that *any* distant point is a better estimator of background than none at all. Case 3 is where X and Y blur together because both are blurry and lie within each other's velocity spread. Note that all categorizations are continuous instead of discrete (figure 4) for robustness to limited precision in both xy for V and z for Z and are order-independent to avoid sorting.

```
function reconstruct( $X, C, V, Z, \text{NeighborMax}$ ):
  // Largest velocity in the neighborhood
  Let  $\vec{v}_N = \text{NeighborMax}[\lfloor X/k \rfloor]$ 
  if  $\|\vec{v}_N\| \leq \varepsilon + 0.5\text{px}$ : return  $C[X]$  // No blur

  // Sample the current point
  weight =  $1.0/\|V[X]\|$ ; sum =  $C[X] \cdot \text{weight}$ 

  // Take  $S - 1$  additional neighbor samples
  let  $j = \text{random}(-0.5, 0.5)$ 
  for  $i \in [0, S), i \neq (S - 1)/2$ :
    // Choose evenly placed filter taps along  $\pm \vec{v}_N$ ,
    // but jitter the whole filter to prevent ghosting
     $t = \text{mix}(-1.0, 1.0, (i + j + 1.0)/(S + 1.0))$ 
    let  $Y = \lfloor X + \vec{v}_N \cdot t + 0.5\text{px} \rfloor$  // round to nearest

    // Fore- vs. background classification of  $Y$  relative to  $X$ 
    let  $f = \text{softDepthCompare}(Z[X], Z[Y])$ 
    let  $b = \text{softDepthCompare}(Z[Y], Z[X])$ 

    // Case 1: Blurry  $Y$  in front of any  $X$ 
     $\alpha_Y = f \cdot \text{cone}(Y, X, V[Y]) +$ 

    // Case 2: Any  $Y$  behind blurry  $X$ ; estimate background
     $b \cdot \text{cone}(X, Y, V[X]) +$ 

    // Case 3: Simultaneously blurry  $X$  and  $Y$ 
     $\text{cylinder}(Y, X, V[Y]) \cdot \text{cylinder}(X, Y, V[X]) \cdot 2$ 

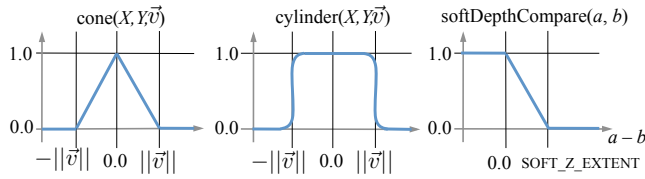
  // Accumulate
  weight +=  $\alpha_Y$ ; sum +=  $\alpha_Y \cdot C[Y]$ 

  return sum/weight
```

Listing 1: Reconstruction applied at (integer) pixel location X

3 Results

Figure 5 compares the result of the new blur algorithm to our implementations of previous plausible methods for per-pixel blur. The



```

function cone( $X, Y, \vec{v}$ ):
    return clamp( $1 - ||X - Y||/||\vec{v}||$ , 0, 1)

function cylinder( $X, Y, \vec{v}$ ):
    return  $1.0 - \text{smoothstep}(0.95||\vec{v}||, 1.05||\vec{v}||, ||X - Y||)$ 

// We use SOFT_Z_EXTENT = 1mm to 10cm for our results
function softDepthCompare( $z_a, z_b$ ):
    return clamp( $1 - (z_a - z_b)/\text{SOFT\_Z\_EXTENT}$ , 0, 1)

```

Figure 4: Continuous classification filters. *cone*: Is X within Y 's point-spread function? *softDepthCompare*: is z_b closer than z_a ?

scene layout and textures are designed to stress the blurry-over-sharp and sharp-over-blurry cases. The top row shows the unfiltered input image C. The second row shows the result of gathering along the velocity vector without dilation [Rosado 2007; Vlachos 2008; Ritchie et al. 2010]; artifacts from that method are the sharp silhouette, apparent shrinking of the moving object, and incorrect blurring of the background behind the cube. The third row demonstrates improvement from a dilated velocity buffer ([Sawada 2007]; comparable to [Epic Games 2010; Sousa 2008]). The moving object now correctly affects pixels beyond its original silhouette, but the background is also blurred where seen through the cube and some silhouettes that should be blurry remain sharp. The final row shows our new method. The moving cube correctly blurs beyond its original silhouette, the background is sharp where seen through a moving object, and all moving edges are blurry.

Figure 6 shows our motion blur resulting from large camera motion. There is no distinction between camera motion and object motion in our blur; both occur from scene points projecting to different locations in screen space. In the top subimage the camera spins inside a cargo bay, creating a predominantly horizontal blur. In the bottom the camera is flying forward through the Crytek Sponza atrium, creating a radial blur that varies with scene depth.

Figure 7 shows interlocking gears rotating at different speeds and directions. This is a hard case for our algorithm because the assumption that each tile is dominated by a single motion vector does not hold. The resulting underblur artifact can be observed in the upper-right gear with the internal spokes. The lower-left spokes are blurrier than the upper-right ones. That is because the large gear in the background dominates NeighborMax in the upper right and suppresses the spoke velocity.

Figure 7 also demonstrates the impact of S on noise in the result image. We find that $S = 5$ is sufficient for consoles, which display on televisions that often suppress and filter this noise. For desktop graphics with more precise monitors, $S = 15$ yields a perceptible increase in image quality and has nearly the same cost, since the overhead of executing the filter on every pixel dominates the small number of texture fetches.

Figure 8 shows a source image of a sports car and two results rendered with different velocity buffers. The center result was rendered with the camera tracking next to the car, so the background and road are blurry. Note that the car color does not pollute the background.

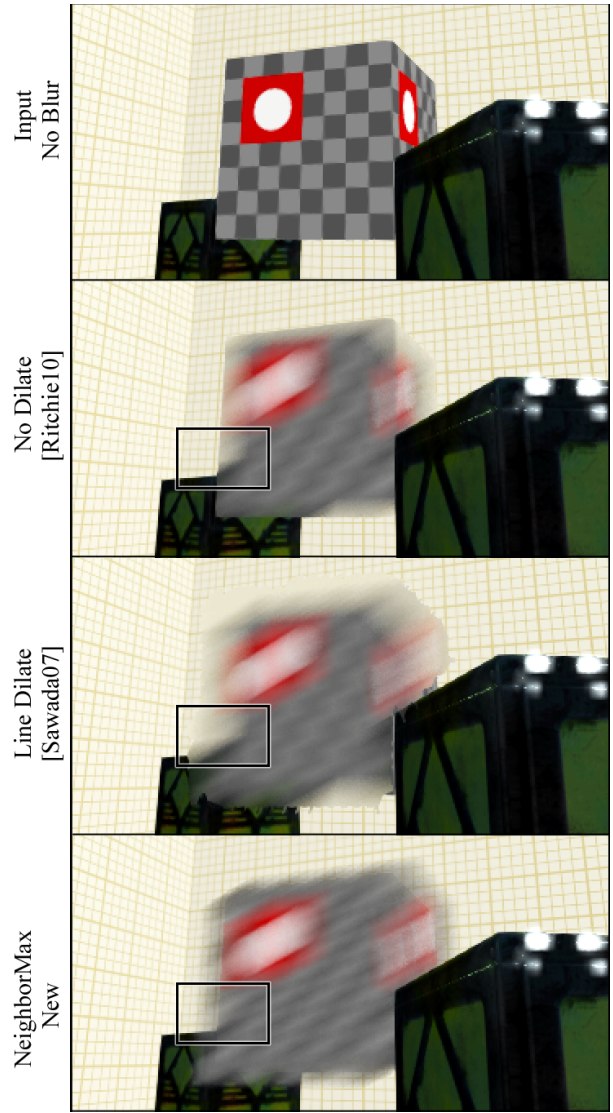


Figure 5: Comparison of plausible, real-time blur methods. A cube moves diagonally in image space between two crates in a room with grid walls. The inset box shows the critical transition region near the border of a moving object.

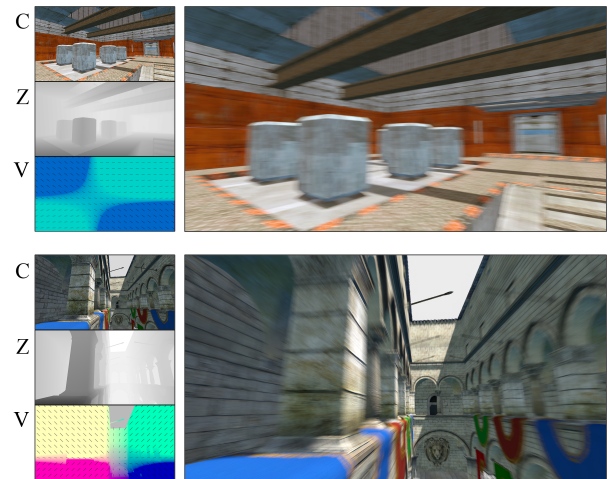


Figure 6: Camera motion results, with input buffers shown at reduced size on the left. Top: Rotation about the vertical axis. Bottom: Translation along the view vector.

The wheels are slightly blurry because they still rotate relative to the camera. The lower result was produced with a static camera, so the car is blurry. Note that background detail is visible through the blurred-out leading edge of the car.

Figure 9 shows a sequence of frames captured while interactively controlling a game character. Per-pixel velocity correctly varies the blur across the character’s body, and the order-independent filter contributions ensure correct blending where the character overlaps himself as well as the scene.

With camera motion creating blur at every pixel at 1280×720 as in figure 1, GeForce 480 renders blur in 6.2ms with $k = 31$, 3.0ms with $k = 15$, 2.8ms with $k = 7$ and 2.7ms with $k = 5$. Xbox 360 renders in 1.5ms with $k = 5$.

4 Discussion

4.1 Artistic Controls

Artistic controls are essential for production special effects. Working in image space makes it relatively easy for us to deviate from a physically-correct velocity buffer for aesthetic purposes. One can render geometry and particles directly into the velocity buffer, bypassing color and depth (figure 10). *Lost Planet 2* leveraged the resulting desirable image smear to render strong explosions. It could also be applied to mimic heat distortion, loud speakers, and semi-visible “cloaked” characters.

It is often desirable to over-blur particle effects and character limbs to enhance fast motion, like attacks, and to under-blur specific objects like signage that the eye would naturally track. Objects can alter the magnitude of the velocity they render by a per-mesh constant or using a velocity-magnitude texture map.

As previously mentioned, a human viewer’s attention shifts within the frame and is hard to predict. For example, a person spinning in real life does not stare radially outward, but instead rapidly shifts focus between fixed scene points. Unless the camera specifically tracks an object with large, slowly-varying velocity (like a plane or car), we recommend diminishing the significance of camera motion. To do so, compute each X' using the object position of the previous frame but a camera transform interpolated between the previous and current frames. Use slerp for the rotation and lerp the translation. We recommend using only 0-15% of the previous camera for a third-person view, and then ramping it up for falling, jump pads and other special cases of high velocity.

Although we developed this method for real-time rendering, it can post-process any image. Figure 11 shows a ray-traced image processed with our blur (data from Lehtinen et al. [2011]). Figure 12 shows blur applied to a photograph augmented with hand-painted coarse depth and velocity maps as a real-world recreation of Cook et al.’s [1984] famous result. Because the blur runs in real-time, the artist can interactively paint the V and Z buffers, as if with a motion-blur paintbrush.

4.2 Limitations

Because the input is a conventionally-rendered image, the blur algorithm fixes the lighting at a single instant within the exposure time. This is a common limitation of many motion blur techniques, including all of the prior plausible, real-time ones. Compared to offline methods that take stochastic input, however, ours also blurs shading and shadows based on the motion of the object instead of the light or shadow caster. We find the result distracting on reflections and hard shadows, and briefly investigated methods for

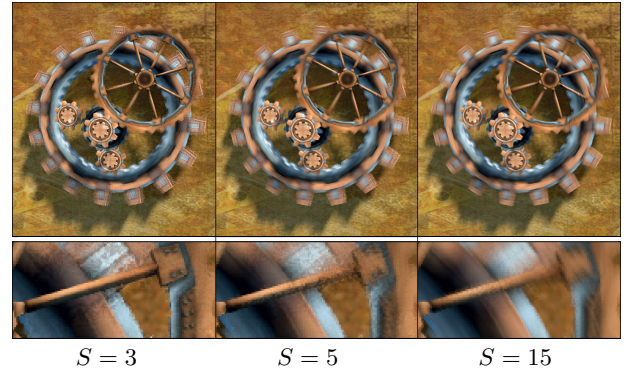


Figure 7: A scene with overlapping elements rotating in the image plane reconstructed with varying sample counts S in the reconstruction filter. The bottom row shows a zoomed detail from the upper-left of the large images.



Figure 8: Top: Input image; no blur. Middle: Camera tracking the car; the background and road appear blurry. Bottom: Camera with no motion in world space; the car appears blurry.

reconstructing shadows, highlights, and reflections based on a separate velocity field for the shadow casters and reflected objects. We abandoned this approach because we observed that for blurry shadows due to PCF or true area-light penumbrae, the visual impact of the artifact was small. It seems that it is more perceptually important to correctly distinguish “sharp” from “blurry” than “moderately blurry” from “very blurry”.

As do many blur techniques, the new method assumes linear velocity for all objects. Experimental results on moderate character and rotational motion (e.g., figures 1, 9, and 7) demonstrate that the visual impact of error due to this assumption is limited even for

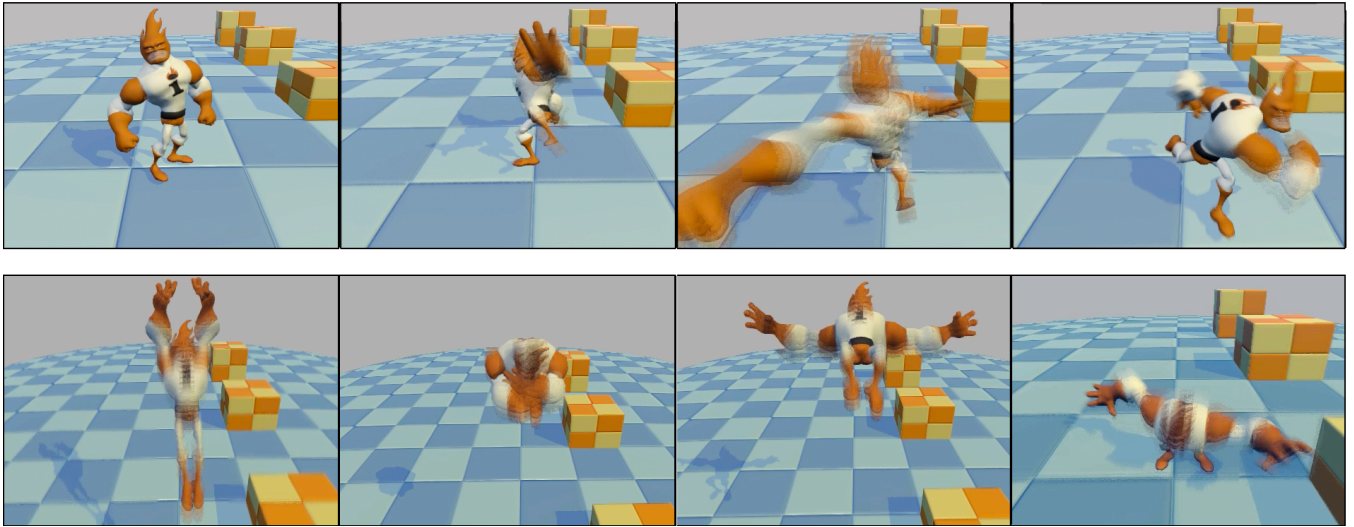


Figure 9: Frames from in-game rendering of interactive character and camera animation ($S = 5$). Velocity and blur correctly vary across the character within a single frame as a result of complex animation.



Figure 10: “Velocity particles” render only into the velocity buffer for special effects such as explosions.



Figure 11: Demonstration of post-processing a ray-traced image with accompanying buffers.

action scenes, but image quality will obviously degrade for large non-linear motions rendered at low framerates.

As with all screen-space filters, the blur technically requires a width- k guard band about the viewport so that off-screen objects may blur onto the screen. However, televisions crop console rendered images, so in practice we do not use a guard band.

4.3 Future Work

For game-like applications, the next steps are to address highly-varying velocities within a neighborhood and to combine motion blur with plausible defocus. The artifacts arising from violation of the dominant-velocity assumption are largest when camera rotation

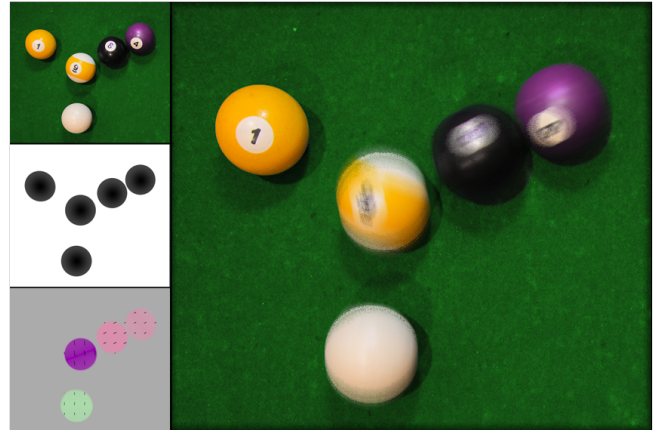


Figure 12: A real photograph of pool balls blurred using hand-painted depth and velocity buffers ($S = 9$).

creates 2D velocity for static elements that is orthogonal to that of dynamic elements.

We note several possibilities for supporting multiple velocities, such as sampling along $V[X]$ as well as \vec{v}_N , or tracking the second principle component in a neighborhood.

Plausible defocus methods are already very effective in isolation. The open questions are whether to apply one before, after, or simultaneously with motion blur, and which method to use.

If implemented in hardware, stochastic rasterization [Akenine-Möller et al. 2007; Fatahalian et al. 2009; Brunhaver et al. 2010; Boulos et al. 2010; McGuire et al. 2010; Munkberg et al. 2011] with MSAA could provide much better input to our reconstruction algorithm. Extending it to that context will be an interesting challenge shaped by the specific rasterization algorithm and sample pattern.

Acknowledgements We thank Jaakko Lehtinen and David Luebke (NVIDIA), and Naty Hoffman (Activision) for their suggestions and editing. Michael Mara (Williams) helped create figure 12.

References

- AKENINE-MÖLLER, T., MUNKBERG, J., AND HASSELGREN, J. 2007. Stochastic rasterization using time-continuous triangles. In *Graphics Hardware*, Eurographics Association, Aire-la-Ville, Switzerland, 7–16.
- BOULOS, S., LUONG, E., FATAHALIAN, K., MORETON, H., AND HANRAHAN, P. 2010. Space-time hierarchical occlusion culling for micropolygon rendering with motion blur. In *High Performance Graphics*, Eurographics Association, Aire-la-Ville, Switzerland, 11–18.
- BOUVIER-ZAPPA, S., OSTROMOUKHOV, V., AND POULIN, P. 2007. Motion cues for illustration of skeletal motion capture data. In *NPAR*, ACM, New York, NY, USA, 133–140.
- BRUNHAVER, J. S., FATAHALIAN, K., AND HANRAHAN, P. 2010. Hardware implementation of micropolygon rasterization with motion and defocus blur. In *High Performance Graphics*, Eurographics Association, Aire-la-Ville, Switzerland, 1–9.
- COOK, R. L., PORTER, T., AND CARPENTER, L. 1984. Distributed ray tracing. In *SIGGRAPH*, ACM, New York, NY, USA, vol. 18, 137–145.
- EGAN, K., TSENG, Y.-T., HOLZSCHUCH, N., DURAND, F., AND RAMAMOORTHY, R. 2009. Frequency analysis and sheared reconstruction for rendering motion blur. In *SIGGRAPH*, 93:1–13.
- EPIC GAMES, 2010. MotionBlur post process effect. Unreal Developer Network, Unreal Engine 3 Documentation, <http://udn.epicgames.com/Three/MotionBlur.html>.
- 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*, ACM, New York, NY, USA, 59–68.
- GRANT, C. W. 1985. Integrated analytic spatial and temporal anti-aliasing for polyhedra in 4-space. *SIGGRAPH* (July), 79–84.
- GREEN, S., 2003. Stupid OpenGL shader tricks. Talk at Game Developers Conference.
- GRIBEL, C. J., DOGGETT, M., AND AKENINE-MÖLLER, T. 2010. Analytical motion blur rasterization with compression. In *High Performance Graphics*, Eurographics Association, Aire-la-Ville, Switzerland, 163–172.
- GRIBEL, C. J., BARRINGER, R., AND AKENINE-MÖLLER, T. 2011. High-quality spatio-temporal rendering using semi-analytical visibility. In *SIGGRAPH*, ACM, New York, NY, USA, 54:1–54:12.
- HARGREAVES, S. 2004. In *Detail Texture Motion Blur*, W. Engel, Ed. Charles River Media, ch. 2.11, 205–214.
- HERZOG, R., EISEMANN, E., MYSZKOWSKI, K., AND SEIDEL, H.-P. 2010. Spatio-temporal upsampling on the GPU. In *I3D 2010*, ACM.
- JONES, N., AND KEYSER, J. 2005. Real-time geometric motion blur for a deforming polygonal mesh. In *Computer Graphics International 2005*, IEEE Computer Society, Washington, DC, USA, 26–31.
- KASYAN, N., SCHULZ, N., AND SOUSA, T., 2011. Secrets of CryENGINE 3 graphics technology. *SIGGRAPH 2011 Talks*.
- LEHTINEN, J., AILA, T., CHEN, J., LAINE, S., AND DURAND, F. 2011. Temporal light field reconstruction for rendering distribution effects. *ACM Trans. Graph.* 30, 4.
- LOTTE, T., 2009. FXAA, February. http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_Whitepaper.pdf.
- LOVISCACH, J. 2005. Motion blur for textures by means of anisotropic filtering. In *EGSR*.
- MAX, N. L., AND LERNER, D. M. 1985. A two-and-a-half-D motion-blur algorithm. In *SIGGRAPH*, ACM, New York, NY, USA, 85–93.
- MAX, N. 1990. Polygon-based post-process motion blur. *Visual Computer* 6 (November), 308–314.
- MCGUIRE, M., ENDERTON, E., SHIRLEY, P., AND LUEBKE, D. 2010. Hardware-accelerated stochastic rasterization on conventional GPU architectures. In *High Performance Graphics*.
- MUNKBERG, J., CLARBERG, P., HASSELGREN, J., TOTH, R., SUGIHARA, M., AND AKENINE-MÖLLER, T. 2011. Hierarchical stochastic motion blur rasterization. In *High Performance Graphics*, ACM, New York, NY, USA, 107–118.
- OBAYASHI, S., KONDO, K., KONMA, T., AND IWAMOTO, K.-I. 2005. Non-photorealistic motion blur for 3d animation. In *SIGGRAPH Sketches*, ACM, New York, NY, USA.
- OVERBECK, R. S., DONNER, C., AND RAMAMOORTHY, R. 2009. Adaptive Wavelet Rendering. *SIGGRAPH Asia* 28, 5, 1–12.
- PEPPER, D. 2008. Per-pixel motion blur for wheels. In *ShaderX⁶*, W. Engel, Ed. Charles River Media, ch. 3.4, 175–188.
- POTMESIL, M., AND CHAKRAVARTY, I. 1983. Modeling motion blur in computer-generated images. In *SIGGRAPH*, ACM, New York, NY, USA, 389–399.
- RITCHIE, M., MODERN, G., AND MITCHELL, K. 2010. Split second motion blur. In *SIGGRAPH Talks*, ACM, New York, NY, USA, 17:1–17:1.
- ROSADO, G. 2007. Motion blur as a post-processing effect. In *GPU Gems 3*. Addison Wesley, ch. 27, 575–581.
- SAWADA, Y., 2007. Talk at Game Developers Conference CEDEC.
- SEN, P., AND DARABI, S. 2011. On Filtering the Noise from the Random Parameters in Monte Carlo Rendering. *ACM Transactions on Graphics (TOG) to appear*.
- SHIMIZU, C., SHESH, A., AND CHEN, B. 2003. Hardware accelerated motion blur generation. *Eurographics*.
- SHIRLEY, P., AILA, T., COHEN, J., ENDERTON, E., LAINE, S., LUEBKE, D., AND MCGUIRE, M. 2011. A local image reconstruction algorithm for stochastic rendering. In *I3D*, ACM, New York, NY, USA, 9–14.
- SOUSA, T., 2008. Crysis next gen effects. Talk at Game Developers Conference.
- SOUSA, T., 2011. CryENGINE 3 rendering techniques, August. Talk at Microsoft Game Technology conference Gamefest 2011.
- SUNG, K., PEARCE, A., AND WANG, C. 2002. Spatial-temporal antialiasing. *IEEE TVCG* 8 (April), 144–153.
- TATARCHUK, N., BRENNAN, C., ISIDORO, J., AND VLACHOS, A. 2003. Motion blur using geometry and shading distortion. In *ShaderX²*, W. Engel, Ed. Charles River Media.
- VLACHOS, A., 2008. Post processing in the Orange Box, February. Talk at Game Developers Conference 2008.
- WLOKA, M., AND ZELEZNIK, R. 1995. Interactive real-time motion blur. In *CGI '94*.