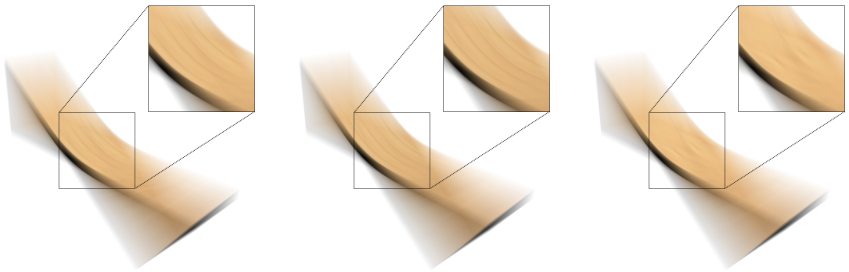
Improved Accuracy for Prism-Based Motion Blur

Improved Accuracy for Prism-Based Motion Blur

Mads J. L. Rønnow Ulf Assarsson Erik Sintorn Marco Fratarcangeli
Chalmers University of Technology



(a) **(b)** **(c)**

Figure 1. A wooden-textured plank undergoing rotation and slight translation. (a) Our method (1.36 ms). (b) Ground truth. (c) Linear interpolation (1.22 ms) [Rønnow et al. 2021]. As can be seen, linear interpolation of depth, normal, and uv coordinates does not provide faithful visual appearance for blur of certain motions, whereas our adaptive sampling method is virtually indistinguishable from the ground truth.

Abstract

For motion blur of dynamic triangulated objects, it is common to construct a prism-like shape for each triangle, from the linear trajectories of its three edges and the triangle's start and end position during the delta time step. Such a prism can be intersected with a primary ray to find the time points where the triangle starts and stops covering the pixel center. These intersections are paired into time intervals for the triangle and pixel. Then, all time intervals, potentially from many prisms, are used to aggregate a motion-blurred color contribution to the pixel.

For real-time rendering purposes, it is common to *linearly* interpolate the ray-triangle intersection and uv coordinates over the time interval. This approximation often works well, but the true path in 3D and uv space for the ray-triangle intersection, as a function of time, is in general nonlinear.

In this article, we start by noting that the path of the intersection point can even partially reside outside of the prism volume itself: i.e., the prism volume is not always identical to the volume swept by the triangle. Hence, we must first show that the prisms still work as bounding volumes when finding the time intervals with primary rays, as that may be less obvious when the volumes differ. Second, we show a simple and potentially common class of cases where

this happens, such as when a triangle undergoes a wobbling- or swinging-like motion during a time step. Third, when the volumes differ, linear interpolation between two points on the prism surfaces for triangle properties works particularly poorly, which leads to visual artifacts. Therefore, we finally modify a prism-based real-time motion-blur algorithm to use adaptive sampling along the correct paths regarding the triangle location and uv coordinates over which we want to compute a filtered color. Due to being adaptive, the algorithm has a negligible performance penalty on pixels where linear interpolation is sufficient, while being able to significantly improve the visual quality where needed, for a very small additional cost.

1. Introduction

Motion-blur renderers take into consideration the time-dependent pixel coverage of dynamic objects during the delta time steps between the discrete frames. Many different solutions exist, both for real-time [Loviscach 2005; Rosado 2007; Ritchie et al. 2010; Bowles et al. 2012; McGuire et al. 2012; Guertin et al. 2014; Guertin and Nowrouzezahrai 2015] and offline purposes [Akenine-Möller et al. 2007; Fatahalian et al. 2009; Gribel et al. 2010; Boulos et al. 2010; Vaidyanathan et al. 2012; Gribel et al. 2013]. This paper targets the prism-based approaches [Shkurko et al. 2017; Hong and Oh 2018; Rønnow et al. 2021].

For efficiency reasons, rasterization- and ray-tracing–based renderers need to compute a bound of the influence region on the screen of each moving triangle and time step. This can be done by computing some bound related to the triangle's delta motion, either in world space [Shkurko et al. 2017; Hong and Oh 2018; Rønnow et al. 2021] or in screen space [Gribel et al. 2010; Gribel et al. 2013]. The trajectory of a dynamic triangle is defined by the trajectory in space and time of each of its vertices. It is popular to construct a prism-like 3D shape from the sweep of the triangle edges along their trajectories, capped by the triangle's start and end position of the delta time step to form a closed 3D volume (see Figure 2(a)) [Brochu et al. 2012; Hong and Oh 2018; Shkurko et al. 2017; Rønnow et al. 2021]. The time values are embedded on the prism surface by interpolation from the prism vertices, and the same typically applies for the triangle's material attributes such as uv values (see Figure 2(b)). Not only can these prisms then be used to project a screen-space bound of the motion, but they can also be used to directly find the time intervals of the pixel coverage by computing the intersection points of each prism and primary ray (see Figure 2(c)).

A major reason for this kind of prism is that they are simple and fast to create geometrically, in contrast to a triangle's swept volume, which can be complicated to both compute and represent. However, they have two implications that have not been fully scrutinized in previous work. First, the prism forms an accurate 2D bounding volume of the triangle sweep when projected onto screen space. This is despite the fact that the prism is not a 3D bounding volume of the sweep, as will be discussed in Section 2.2. Second, the time intervals are guaranteed to be bound by the time values

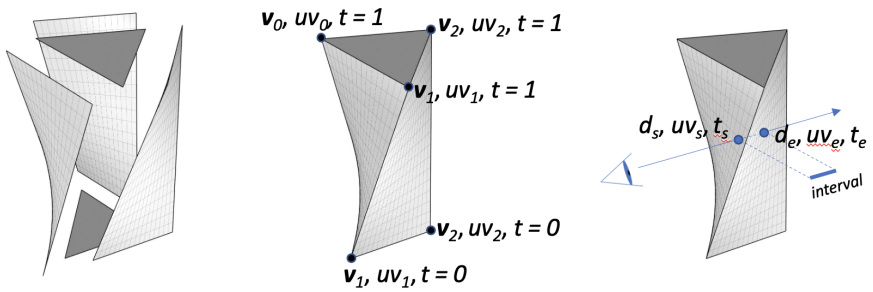


Figure 2. (a) The prism consists of the triangle's start and end positions with sides from the swept triangle edges. (b) Time and uv values are embedded on the prism surface by interpolation from the vertex attributes. (c) The intersection with a primary ray provides time intervals with t, uv, d at its extreme points.

at the ray intersection points on the prism surface, but this does not hold for other properties, e.g., the uv values and triangle location that covers the pixel center during the sweep.

It is quite common to assume that the triangle vertices move linearly between their start and end positions of a time step—even for cases when the triangles represent a moving rigid object and the vertex paths should be nonlinear to maintain the rigid shape [Gribel et al. 2013]. For linear vertex motions, the prism sides become bilinear patches (see Figure 2). It is relatively easy to compute some conservative screen-space bound for such prisms and also to accurately interpolate properties over the bilinear patches from the prism vertices [Reshetov 2019]. Sometimes, the bilinear patches are also further approximated—by two or more triangles each [Hong and Oh 2018; Shkurko et al. 2017; Brochu et al. 2012].

The dynamic triangle will cover individual pixel centers during continuous time intervals. For linear vertex motions, there can in theory be up to four such intervals per pixel and prism [Gribel et al. 2010], although there is often just one interval for simple motions. These time intervals are constructed by finding all intersection points between the prism and a primary ray through the pixel center. These intersection points provide the time value when the ray starts and stops intersecting the triangle. The time value is paired with the depth location along the ray and possibly other parameters such as uv values. This can be done by explicit ray casting [Shkurko et al. 2017; Rønnow et al. 2021] or by rasterizing the prism [Hong and Oh 2018].

Because a dynamic triangle by definition is planar at each instant during its motion, the time intervals of one and the same triangle cannot overlap. They are disjoint: i.e., a ray cannot intersect the dynamic triangle twice at any single moment, t. Hence, it is possible to sort the ray-prism intersection points on time and pairwise connect them into intervals [Shkurko et al. 2017]. The property of disjoint time intervals is

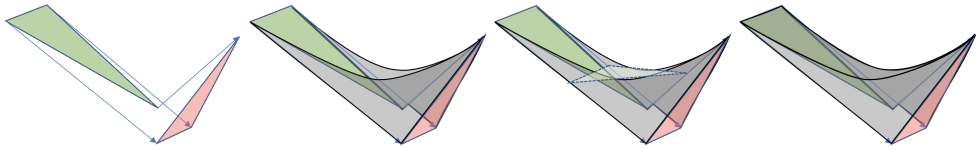
very important, as it allows interval construction by only considering the intersection points. In contrast, it would be harder to construct intervals from the intersection points of the surface of the triangle's swept volume (as opposed to the prism), in the general case, because those intersection points represent entry and exit points in space with spatial intervals that may overlap.

Motion blur is then reconstructed per pixel using each interval's time span and visibility with respect to other triangle intervals. The visibility is resolved by sorting, per pixel, all the ray-prism intervals based on the their start times. Where these intervals overlap in time, they are clipped into subintervals, such that overlapping subintervals can be sorted on depth to resolve visibility between all the triangles covering the pixel during the time step [Gribel et al. 2010; Rønnow et al. 2021]. If subintervals also overlap in depth, which means that two triangles collide in space and time, they are further clipped into non-overlapping parts, for proper depth sorting.

2. Problem

There is, however, an important problem with the aforementioned approach. For efficiency reasons, linear interpolation is typically used for surface properties between an interval's start and end points [Hong and Oh 2018; Shkurko et al. 2017; Rønnow et al. 2021]. Such surface properties include depth and uv used for visibility and shading. Though linear interpolation often works well enough, it is inaccurate in the general case [Gribel et al. 2010]. This is particularly pronounced when the prism does not truly match the volume swept by the triangle (see Figures 3 and 4).

First, we demonstrate a simple such case that can be common in practice. We also note that the prism and triangle's swept volume have identical 2D projections, which may not be totally clear for situations where the volumes are different. This is important in order to be able to find all the time intervals from just ray-prism intersections. Then we will discuss the depth- and uv-interpolation problems due to nonlinearity. Finally, in the next section, we present a real-time solution that adaptively subdivides



(a) *Triangle motion,* $t = 0 \rightarrow 1$ **(b)** *Prism* **(c)** *Triangle location at* $t = 0.5$ **(d)** *Triangle's swept volume*

Figure 3. The prism does not always represent the volume swept by the triangle. (a) Such a triangle motion. (b) The corresponding prism is hollow above the green triangle. (c) The triangle's position halfway through its complete motion, where the triangle is mostly located outside of the prism volume. (d) The triangle's swept volume.

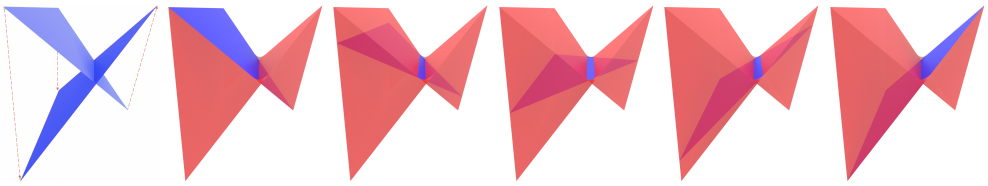


Figure 4. (a) A triangle undergoes a rotational and translational motion from one position to another with linear vertex motion, and passes outside its bounding prism. In (b) and (f), the triangle's start and end positions constitute parts of the prism boundary. However, as the triangle moves between the two end positions, the triangle appears outside the prism at the rotation pivot area, as shown in blue in (c)–(e). Note that the triangle silhouette stays within the silhouette of the prism.

the time intervals, with linear interpolation within each subdivision.

2.1. Prism Volume Can Differ from Swept Volume

Figure 3 shows an example of a swinging-like motion where the swept volume of the triangle, from its start to end positions, differs from the prism volume. This happens when the interior of the triangle surface sweeps outside of the prism. The triangle edges will by definition always lie on the prism sides. However, there are no guarantees that the interior triangle points are located within the prism volume. An example of a wobbling-like motion around a pivot region is shown in Figure 4.

2.2. Prism and Swept-Triangle Volume Have Equal 2D Projections

A prism and the actual volume swept by a triangle cover the same pixels under a perspective projection onto a 2D plane. This means that their silhouettes as seen from any camera position are identical. An illustrated example can be seen in Figure 3(b) versus 3(d) and in Figure 4. If this would not be the case, then intersections between the prism and primary rays from the camera position would not be guaranteed to find all pixels covered by the swept triangle. The following presents an argumentation for why and under which conditions the equal projections hold.

Prisms describe all the 3D locations where a ray will time-wise start and stop intersecting the swept triangle. From a rasterization perspective, these times correspond to when the pixel starts and stops being covered by the triangle (see Figure 5(a)). This is true for the following reasons: First, the start and end triangles of the prism mark the dynamic triangle's 3D locations for $t = 0$ and $t = 1$, if t is the normalized time within the delta time step. Hence, if the triangle covers the pixel center at $t = 0$ or $t = 1$, it will be found by a ray-prism intersection. Second, for $0 < t < 1$, the triangle can only start or stop covering the pixel center, i.e., hitting the primary ray, via its

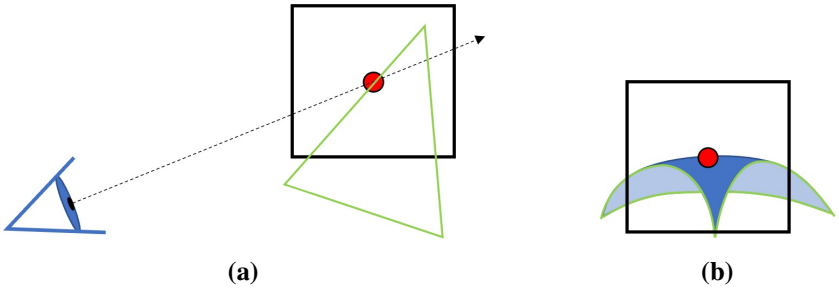


Figure 5. (a) A triangle sweeping by a pixel center (red) must start and stop intersecting the primary ray at the triangle edges. (b) This is not true for curved patches. The example shows a dynamic curved patch starting or stopping intersecting the pixel at an interior point instead of along its edges (green): i.e., a curved patch's silhouette is not guaranteed to consist of the patch edges only.

edges, or more accurately, at the time when at least one of its edges intersects the pixel center (see Figure 5(a)). This also covers the degenerate case when the primary ray is in the triangle plane, such that the intersection is a line across the triangle. This is true because triangles are planar. Figure 5(b) shows an example of a curved triangular patch intersecting the pixel center at a patch-interior point, i.e., not along its edges.

Hence, a ray intersection with the prism safely finds all the start and stop times for $0 \leq t \leq 1$. Thus, a rasterization of the prism will also cover the same pixels as the rasterization of the swept volume, no matter the viewpoint or sampling density. This means that the 2D projection of the prism is identical to the 2D projection of the swept volume, despite the fact that their 3D volumes may differ. This in turns means that their silhouettes are identical from any viewing position.

Another way to understand the latter is that because a moving triangle at any time instance is planar, interiors of the triangle can never become the silhouette during its motion (unless the triangle is parallel to the view ray, in which case the edge is also the silhouette). Hence, the swept triangle and the prism containing the swept edges have the same silhouette.

As we have not been forced to make assumption about the vertex motions, other than being continuous, this means that the prism's and the swept volume's projections are also identical even for nonlinear vertex motions. In this paper, however, we assume linear vertex motions because that simplifies constructing the prisms.

2.3. Nonlinear Depth and *uv* Functions

When the prism volume and the triangle's swept volume differ, the nonlinearity of the depth and uv functions are particularly pronounced. The depth interval that the triangle covers at the pixel center during its motion can extend outside the values found at the time intervals' extreme points, and the same is true for the uv values.

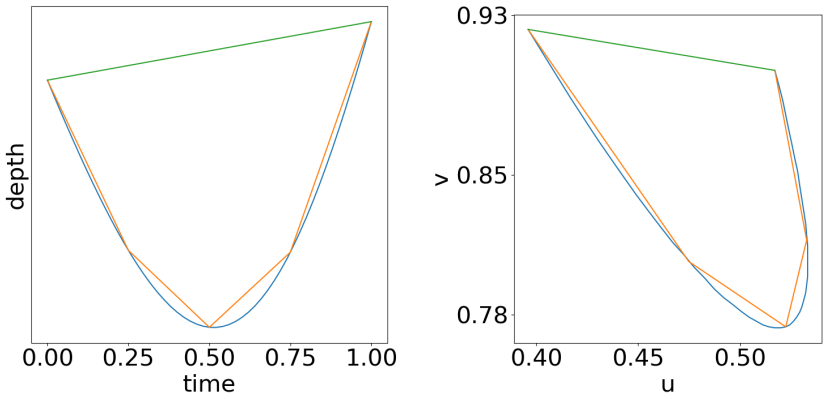


Figure 6. Plots of the nonlinear depth and uv values in a pixel at the rotation pivot area of the triangle in Figure 4. The blue curve is the ground truth, the green segment represents linear interpolation, and the orange segment is the result achieved by our adaptive subdivision.

This causes concerns in the depth-sorting step and also for shading.

For dynamic scenes, there will be overlapping depth intervals between separate triangles, for which triangle visibility needs to be resolved. The depths at the time interval's end points cannot safely be used, because they will not include the full nonlinear depth information of the interval. Hence, the depth functions need to be more properly considered [Gribel et al. 2010].

We use adaptive subdivision that splits an interval in half if the difference between the interpolated depth and the true depth computed by the analytic depth function at the midpoint between start depth and end depth is greater than a certain threshold. This also improves on the inaccuracies in the lighting computations regarding linearly interpolated surface locations.

The uv values are also a nonlinear function of time (see Figure 6). Though we could also adaptively subdivide the time intervals based on how much the uv values deviate from linear interpolation, we choose as a heuristic to only base the subdivision on the depth values. The results in the next section indicate that this works well for our test cases.

3. Our Real-Time Method

Our method is based on the GPU implementation by Rønnow et al. [2021], which uses linearly approximated depth and uv values for the intervals. Their system is devised of four stages: (1) creating the prisms based on time step start and end vertex attributes, (2) generating the intervals based on ray-prism intersections, (3) sorting the intervals by start time, and (4) resolving final pixel colors based on interval order and overlaps.

Though the stages of our method remain the same as in that work, we have ex-

tended the interval generation (Step 2) in order to facilitate the adaptive subdivision of intervals.

In our adaptive subdivision method, the interval generation is a hybrid approach. First, the intervals are generated from the prism intersections, and then they are checked for nonlinearity using the edge equation–based functions described by Gribel et al. [2010]. The barycentric coordinates of the ray–moving triangle intersection are computed by the function

$$(u, v) = \frac{1}{e_0 + e_1 + e_2}(e_1, e_2), \tag{1}$$

where e_i are the time-dependent edge equations. The general vertex attribute interpolation function is defined by

$$s(u, v) = (1 - u - v)\mathbf{p}^0 + u\mathbf{p}^1 + v\mathbf{p}^2, \tag{2}$$

where $\mathbf{p}^k$ are the attributes at the three vertices of the triangle, and u, v are the barycentric coordinates computed by Equation (1). The depth function is as follows:

$$d(t) = \frac{s_z}{s_w} = \frac{(1 - u - v)\mathbf{p}_z^0 + u\mathbf{p}_z^1 + v\mathbf{p}_z^2}{(1 - u - v)\mathbf{p}_w^0 + u\mathbf{p}_w^1 + v\mathbf{p}_w^2}, \tag{3}$$

where $d(t)$ is the depth along the ray at its intersection with the triangle at time t, $\frac{s_z}{s_w}$ is the perspective-correct interpolation of the depth attributes, and $\mathbf{p}^k = (p_x^k, p_y^k, p_z^k, p_w^k)$, $k \in \{0, 1, 2\}$, are the three triangle vertices in clip space, i.e., before the division by w.

This hybrid approach removes the need for the edge equation root finding, which is replaced by the ray-prism intersection operation. The intersection time value is found with the ray-prism intersection test, and the time value is then used as input to the depth function (Equation (3)). The method is outlined in Algorithm 1.

To determine if an interval needs to be subdivided, we find the approximate extrema by sampling the depth values between the interval's start and end points using the depth function. If the depth value at one sampling point is not between the values of its two neighbors, then we regard it as an extrema point. This gives us the (approximate) extrema points of the nonlinear depth function, which can be used to linearize it. To further increase the accuracy of the linearization, the subintervals are conditionally subdivided again if the interpolated depth value at the subinterval midpoint differs (above a chosen threshold) from the analytically computed depth at the midpoint. This results in up to two subintervals between an original interval end point and an extrema point. The orange plots in Figure 6 are an example of such a linearization.

The rest of the subinterval data is then finally computed based on the subinterval time values. We again use the functions described by Gribel et al. [2010]. For mesh uv and normal values, we use the general attribute interpolation defined by Equation (2).

Algorithm 1: Adaptive subdivision algorithm.

Input : A prism interval: I_{in}
Output: List of subdivided intervals I_{out}
Initialize:
    d_s = depthFunction($I_{in}.t_s$); d_e = depthFunction($I_{in}.t_e$);
    numIntervals = 20.0f · ($I_{in}.t_e$ – $I_{in}.t_s$)
    dt = ($I_{in}.t_e$ – $I_{in}.t_s$) · (1.0f / nIntervals);
    t = $I_{in}.t_s$ + dt;
    // Array of extreme points from t_s to and including t_e:
    depths[6], times[6]; // Depth and corresponding time values
    depths[0] = d_s; times[0] = $I_{in}.t_s$;
    d_{prev} = d_s;
    depth = (nIntervals < 2.0f) ? d_e : depthFunction(t); currIndex = 0;
// Find approximate extrema points:
for *interval = 0; interval < nIntervals; interval++* **do**
    t = t + dt;
    d_{next} = (interval == int(nIntervals – 1.0f)) ? d_e : depthFunction(t + dt);
    // If d_{prev} and d_{next} are the same then skip:
    if $abs(d_{prev} - d_{next}) > \epsilon$ **then**
        // If depth is not between d_{prev} and d_{next}, then t is a local extreme:
        if $(depth - d_{prev}) \cdot (d_{next} - depth) < 0$ **then**
            currIndex = currIndex + 1;
            depths[currIndex] = depth;
            times[currIndex] = t;
    d_{prev} = depth;
    depth = d_{next};
// Add the point at t_e:
currIndex = currIndex + 1;
depths[currIndex] = d_e; times[currIndex] = $I_{in}.t_e$;

// Further subdivide conditionally:
nIndices = currIndex;
for *i = 0; i < nIndices; i++* **do**
    midDepth = (depths[i] + depths[i+1]) · 0.5f;
    midTime = (times[i] + times[i+1]) · 0.5f;
    if $abs(depthFunction(midTime) - midDepth) > \epsilon$ **then**
        I_{out}.insert(createSubInterval(times[i], midTime));
        I_{out}.insert(createSubInterval(midTime, times[i+1]));
    else
        I_{out}.insert(createSubInterval(times[i], times[i+1]));

	Plank 12 tris 60 prism faces		Character Running 49k tris 245k prism faces		Chip Rotating 508 tris 2.54k prism faces	
	Linear	Ours	Linear	Ours	Linear	Ours
Time per frame (ms)	1.22	1.36	22.6	23.4	2.09	2.13
Max. intervals per pixel	7	13	525.1	525.2	73.8	74.8
Interval count $\left(\times 10^6\right)$	0.240	0.360	2.278	2.282	0.785	0.850
GPU memory (MB)	23	28.5	188.6	188.9	96.7	100.2

Table 1. Performance comparison between linear interpolation [Rønnow et al. 2021] and our method. For the *wooden-textured plank* (Plank) scene the values are based on a single frame. For the *character running* and the *chip rotating* scenes, the values displayed are averages over all frames in the sequence, whereas GPU memory is the maximum allocated GPU memory over the entire sequence. The resolutions used are 1024×1024 for the *wooden-textured plank* scene, 1920×1080 for the *character running* scene, and 512×1024 for the *chip rotating* scene.

4. Results

Our adaptive subdivision motion blur system has been implemented and tested in OpenGL 4.6 and C++/CUDA 10.2 on an NVIDIA RTX 3080 system running Windows 10. We have tested the scenes *wooden-textured plank* and *rotating chip*, as well as the *character running* scene [Rønnow et al. 2021] for performance comparison.

Performance results can be seen in Table 1. As the results in the table show, the *character running* test scene exhibits that our adaptive subdivision method deals well with scenes with little or no nonlinear motion by only marginally negatively affecting performance compared to the linear method [Rønnow et al. 2021]. Visually, the two methods produce, as expected, indistinguishable results (see Figure 7). Similarly, our results for the *wooden-textured plank* and *rotating chip* scenes also perform only marginally worse than the linear method.

The improved image quality of the adaptive subdivision approach compared with linear depth and uv values is made clear in the *wooden-textured plank* (Figure 1) and *rotating chip* (Figure 8) scenes. In the plank test, we see that the linear method loses the details of the lines in the texture at the rotation pivot area, whereas our method retains these details to a quality close to the ground truth. The *rotating chip* scene suffers from striping artifacts with the linear method, which are not present in the results produced by our method. Ground truth results were produced by the same brute-force method as was used for the ground truth by Rønnow et al. [2021], i.e., by averaging from a few hundred up to a few thousand images with small time increments.

[Rønnow et al. 2021] *Ours* *[Rønnow et al. 2021]* *Ours*

Figure 7. Comparison render of two different frames of the *character running* scene. Our method is able to render the same scene with adaptive subdivision at only marginally higher frame times (see Table 1). The two methods produce visually indistinguishable results, as there is no notable nonlinear motion in this animation sequence.

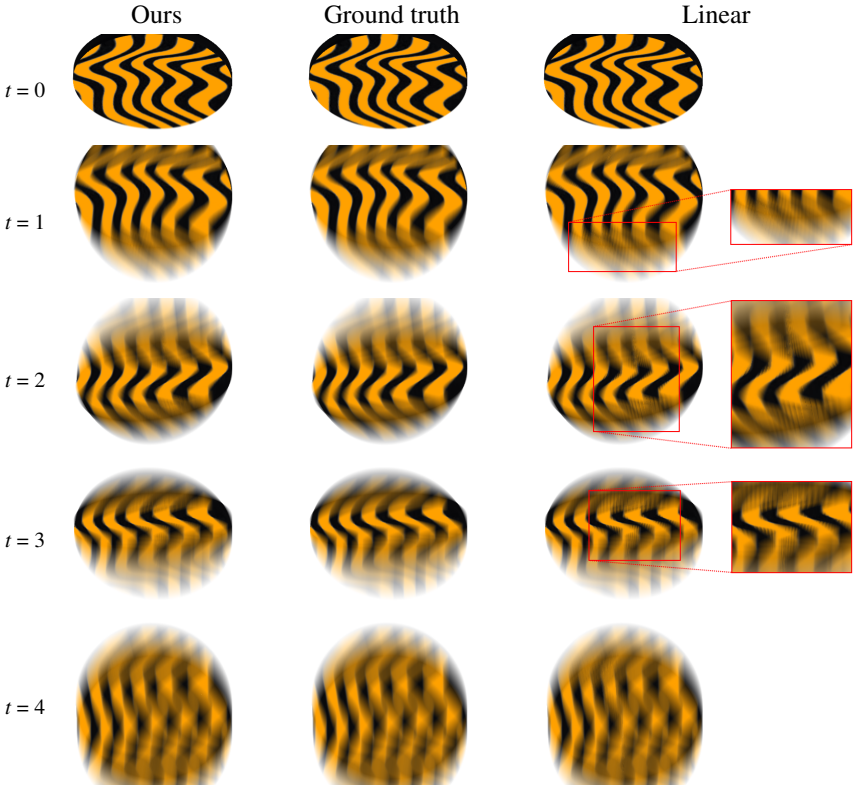


Figure 8. Chip rotating while falling quality comparison. Left: Our method. Middle: Ground truth. Right: Linear interpolation [Rønnow et al. 2021]. For this scene, rendering was done without shading to remove the effect of intervals only having start normals, and to highlight the negative effect of linearly interpolating uv values. In the linear case there is undesirable incorrect interpolation of texture attributes (highlighted by the red boxes).

5. Conclusion and Future Work

We have constructed a hybrid ray-prism intersection and edge equation–based adaptive subdivision solution to approximate the nonlinear depth, normal, and uv functions. This adaptive subdivision improves the visual quality, especially in areas where the prism does not bound the triangle motion conservatively, and with only a small performance penalty. As future work, our subdivision scheme could be used for other properties than uv or depth, such as a filtered shadow-map value or filtered reflection color from a cube map.

We have shown examples of when a prism differs from the swept volume and, hence, does not constitute the 3D bounding volume. As future work, it would be interesting to explore these implications for robust prism-based collision detection [Brochu et al. 2012] on the GPU.

Acknowledgements

This work was supported by the Swedish Research Council under Grants 2015-05345 and 2014-4559.

References

AKENINE-MÖLLER, T., MUNKBERG, J., AND HASSELGREN, J. 2007. Stochastic rasterization using time-continuous triangles. In *ACM SIGGRAPH/Eurographics Graphics Hardware*, Eurographics Association, 7–16. URL: https://dl.acm.org/doi/10.5555/1280094.1280096. 83

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, 11–18. URL: https://dl.acm.org/doi/10.5555/1921479.1921482. 83

BOWLES, H., MITCHELL, K., SUMNER, R. W., MOORE, J., AND GROSS, M. 2012. Iterative image warping. *Computer Graphics Forum 31*, 2pt1, 237–246. URL: https://doi.org/10.1111/j.1467-8659.2012.03002.x. 83

BROCHU, T., EDWARDS, E., AND BRIDSON, R. 2012. Efficient geometrically exact continuous collision detection. *ACM Transactions on Graphics 31*, 4, 96:1–96:7. URL: https://dl.acm.org/doi/10.1145/2185520.2185592. 83, 84, 93

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 *Proceedings of the Conference on High Performance Graphics 2009*, ACM, 59–68. URL: https://dl.acm.org/doi/10.1145/1572769.1572780. 83

GRIBEL, C. J., DOGGETT, M. C., AND AKENINE-MÖLLER, T. 2010. Analytical motion blur rasterization with compression. In *High Performance Graphics*, Eurographics Association, 163–172. URL: https://dl.acm.org/doi/10.5555/1921479.1921504. 83, 84, 85, 88, 89

GRIBEL, C. J., MUNKBERG, J., HASSELGREN, J., AND AKENINE-MÖLLER, T. 2013. Theory and analysis of higher-order motion blur rasterization. In *Proceedings of the 5th High-Performance Graphics Conference*, ACM, 7–15. URL: https://dl.acm.org/doi/10.1145/2492045.2492046. 83, 84

GUERTIN, J.-P., AND NOWROUZEZAHRAI, D. 2015. High performance non-linear motion blur. In *Eurographics Symposium on Rendering—Experimental Ideas & Implementations*, Eurographics Association, 99–105. URL: https://diglib.eg.org/handle/10.2312/sre20151171. 83

GUERTIN, J.-P., MCGUIRE, M., AND NOWROUZEZAHRAI, D. 2014. A fast and stable feature-aware motion blur filter. In *High Performance Graphics*, Eurographics Association, 51–60. URL: https://diglib.eg.org/handle/10.2312/hpg.20141093.051-060. 83

HONG, M.-P., AND OH, K. 2018. Real-time motion blur using extruded triangles. *Multimedia Tools and Applications 77*, 11, 13323–13341. URL: https://link.springer.com/article/10.1007/s11042-017-4949-6. 83, 84, 85

LOVISCACH, J. 2005. Motion blur for textures by means of anisotropic filtering. In *Eurographics Symposium on Rendering (2005)*, K. Bala and P. Dutre, Eds., Eurographics Association, 105–110. URL: http://diglib.eg.org/handle/10.2312/EGWR.EGSR05.105-110, doi:10.2312/EGWR/EGSR05/105-110. 83

MCGUIRE, M., HENNESSY, P., BUKOWSKI, M., AND OSMAN, B. 2012. A reconstruction filter for plausible motion blur. In *Interactive 3D Graphics and Games*, ACM, 135–142. URL: https://dl.acm.org/doi/10.1145/2159616.2159639. 83

RESHETOV, A. 2019. Cool patches: A geometric approach to ray/bilinear patch intersections. In *Ray Tracing Gems*, E. Haines and T. Akenine-Möller, Eds. Apress, 95–109. URL: https://link.springer.com/chapter/10.1007/978-1-4842-4427-2_8. 84

RITCHIE, M., MODERN, G., AND MITCHELL, K. 2010. Split second motion blur. In *ACM SIGGRAPH 2010 Talks*, ACM, 17:1. URL: https://doi.org/10.1145/1837026.1837048, doi:10.1145/1837026.1837048. 83

RØNNOW, M. J., ASSARSSON, U., AND FRATARCANGELI, M. 2021. Fast analytical motion blur with transparency. *Computers & Graphics 95*, 36–46. URL: https://doi.org/10.1016/j.cag.2021.01.006. 82, 83, 84, 85, 88, 91, 92

ROSADO, G. 2007. Motion blur as a post-processing effect. In *GPU Gems 3*, H. Nguyen, Ed. Addison-Wesley, 575–581. URL: https://developer.nvidia.com/gpugems/gpugems3/part-iv-image-effects/chapter-27-motion-blur-post-processing-effect. 83

SHKURKO, K., YUKSEL, C., KOPTA, D., MALLETT, I., AND BRUNVAND, E. 2017. Time interval ray tracing for motion blur. *IEEE Transactions on Visualization and Computer Graphics 24*, 12, 3225–3238. URL: https://ieeexplore.ieee.org/document/8115176. 83, 84, 85

VAIDYANATHAN, K., TOTH, R., SALVI, M., BOULOS, S., AND LEFOHN, A. 2012. Adaptive image space shading for motion and defocus blur. In *High-Performance Graphics*, Eurographics Association, 13–21. URL: https://dl.acm.org/doi/10.5555/2383795.2383798. 83

Author Contact Information

Mads J. L. Rønnow
Chalmers University of Technology
roennowmads@gmail.com

Ulf Assarsson
Chalmers University of Technology
uffe@chalmers.se
www.cse.chalmers.se/~uffe

Erik Sintorn
Chalmers University of Technology
erik.sintorn@chalmers.se

Marco Fratarcangeli
Chalmers University of Technology
marco.fratarcangeli@gmail.com
www.cse.chalmers.se/~marcof

Mads J. L. Rønnow, Ulf Assarsson, Erik Sintorn, and Marco Fratarcangeli, Improved Accuracy for Prism-Based Motion Blur, *Journal of Computer Graphics Techniques (JCGT)*, vol. 11, no. 1, 82–95, 2022
http://jcgt.org/published/0011/01/05/

Received:	2021-07-10		
Recommended:	2021-12-27	Corresponding Editor:	Angelo Pesce
Published:	2022-03-04	Editor-in-Chief:	Marc Olano