Graphics Hardware (2007)
Timo Aila and Mark Segal (Editors)

Stochastic Rasterization using Time-Continuous Triangles

Tomas Akenine-Möller Jacob Munkberg Jon Hasselgren

Lund University

Abstract

We present a novel algorithm for stochastic rasterization which can rasterize triangles with attributes depending on a parameter, t, varying continuously from $t = 0$ to $t = 1$ inside a single frame. These primitives are called time-continuous triangles, and can be used to render motion blur. We develop efficient techniques for rasterizing time-continuous triangles, and specialized sampling and filtering algorithms for improved image quality. Our algorithm needs some new hardware mechanisms implemented on top of today's graphics hardware pipelines. However, our algorithm can leverage much of the already existing hardware units in contemporary GPUs, which makes the implementation fairly inexpensive. We introduce time-dependent textures, and show that motion blurred shadows and motion blurred reflections can be handled in our framework. In addition, we also present new techniques for efficient rendering of depth of field and glossy planar reflections using our stochastic rasterizer.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware ArchitectureGraphics processors; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and RealismAnimation, Color, shading, shadowing, and texture, Hidden line/surface removal;

1. Introduction

If objects in the field of view of the camera, or the camera itself move, and the shutter of the camera is open for a finite amount of time, an image with motion blur is obtained. Real photographs and video often contain motion blur, and therefore, this effect is commonly and heavily used in the movie industry using offline rendering tools. In contrast, most real-time graphics applications assume the shutter is open only for an infinitesimal amount of time, which means that motion blur is absent. However, it is our impression that motion blur is a highly desirable feature even for real-time games.

Rendering motion blur is a hard problem to attack since it involves solving visibility in the spatio-temporal domain, i.e., both in screen space and in time. Currently there exists only a few algorithms capable of rendering this effect in real time. However, they usually only solve the problem for a limited domain, e.g., only the textures of the objects are blurred and not the geometrical objects themselves, and consequently, visibility is solved incorrectly.

Cook et al. [CPC84] concluded the following on rendering correct motion blur, and this appear to hold true even today:

> "*Point sampling seems to be the only approach that offers any promise of solving the motion blur problem.*"

Therefore, we introduce an algorithm for rasterization-based point sampling in time using a time-continuous triangle representation. This makes it possible to render motion blurred images with sufficient quality for real-time graphics at only four samples per pixel. Since current GPUs already support spatial supersampling with that amount of samples, we can integrate our algorithm into an existing GPU without increasing the number of samples. In addition, some parts of our algorithm can be executed using geometry and pixel shaders. Only a small portion of our algorithm needs new hardware mechanisms on top of the existing units already available in contemporary GPUs.

This introduction and the entire description of our our algorithms (Section 3) focus on rendering the motion blur effect only. The reason for this is that it greatly simplifies the presentation. However, in our results (Section 4), we show that the exact same framework can be used to render depth of field and glossy reflections as well.

2. Previous Work

An excellent overview of previous work in motion blur research is presented by Sung et al. [SPW02]. In the following, we will review related work that is of particular interest to our research. This means, for example, that we avoid discussing algorithms that produce motion blur only as a post-process, as these cannot solve the problem properly.

Several analytical models for motion blur have been developed [KB83, Cat84, Gra85] for scanline renderers. Due to the evolution of the GPUs into stream processors, these algorithms are not directly well suited for hardware implementation in their current state, since they require a sorting pass to resolve time-space visibility per pixel.

Rendering motion blur using graphics hardware can be done by rendering n images at different points in time, and then averaging these using an accumulation buffer [DWS*88, HA90]. It should be noted that strobing ar-

GH 2007, San Diego, California, August 04 - 05, 2007
© 2007 ACM 978-1-59593-625-7/07/0008 $ 5.00

tifacts appear unless many images are used. However, the final image converges to the correct result when more images are added. These algorithms are expensive in terms of geometry processing, since the entire scene needs to be sent to the graphics pipeline n times. A variant of these accumulation buffer techniques is *practically frameless rendering* (PFR) [WZM95], which is a rasterization-based version of the original frameless rendering algorithm for ray tracing [BFMZ94]. In PFR, less than one sample per pixel is generated per frame. For example, one can choose to render to only one fourth of the pixels every frame. After four frames, a complete image has been rendered. A variant of this, called *temporal anti-aliasing*, is supported by some ATI graphics cards [ATI04].

In the REYES rendering architecture [CCC87], primitives are diced until they reach subpixel size, then shading is computed, and finally the primitives are sampled. This is basically a high-quality rasterization engine. However, motion blurred shading cannot be handled correctly since shading is done before sampling. Furthermore, in this original approach, shadows appear to lack motion blur.

For offline high-quality rendering, Wexler et al. [WGER05] conclude that accumulation buffering works well when many images are used, and so they use that approach in their Gelato renderer. However, they also investigate whether a specialized shader can be used to sample stochastically in time. This approach degrades more gracefully than uniform sampling when decreasing the number of samples. They abandon this technique due to inefficient rasterization and because early Z-culling cannot be used, since they write to the depth buffer in the shader. Our work was inspired by Wexler et al's stochastic sampling, but instead of focusing on using only existing hardware, we also develop new hardware mechanisms suitable for implementation on top of today's pipelines for potentially much higher performance.

The remaining motion blur algorithms which we will describe are targeted for real-time graphics. A common disadvantage for these is that the rendered images do not converge to the correct result even if more computations or more samples are used. Some algorithms compute the silhouette of motion, extend the silhouette geometry in the direction of motion and then render semi-transparent primitives [WZ96, JK01]. These algorithms cannot correctly handle shaded and textured objects, and so in practice, they are not very useful. In contrast, Loviscach [Lov05] has presented an algorithm that deals with motion blurred textures. However, blurring takes place only in texture space, and hence spatio-temporal visibility is not solved at all. Another approach is to render an object once into a texture, and at the same time create a vector field of the per-pixel motion [SSC03]. In a final pass, the texture is blurred according to the vector field. Again, spatio-temporal visibility is not handled correctly.

Depth of field (DOF) is the effect in which objects outside some distance range appear out of focus. A good survey of techniques to simulate DOF is presented by Demers [Dem04]. Correct DOF can be rendered by distributing rays stochastically over the camera lens, rather than shooting a ray from a single point, or equivalently, render the scene from multiple cameras and accumulate the results. However, for acceptable quality, these approaches require many rays or render passes and are currently too costly for real-time graphics. Faster methods using depth layers, point splatting and variable blur kernels exist, but they cannot resolve visibility correctly.

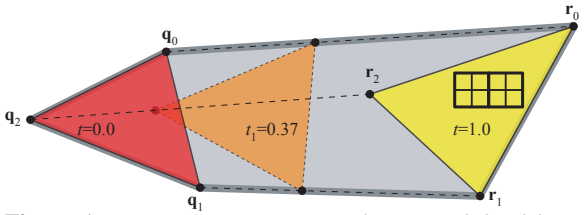


Figure 1: *A time-continuous triangle (TCT) defined by a starting triangle, $\Delta\mathbf{q}_0\mathbf{q}_1\mathbf{q}_2$, at $t = 0$, and an ending triangle, $\Delta\mathbf{r}_0\mathbf{r}_1\mathbf{r}_2$, at $t = 1$. The TCT is simply the continuous set of linearly interpolated triangles between $t = 0$ and $t = 1$.*

3. Stochastic Rasterization

In this section, we present our algorithm for stochastic rasterization. As a high-level overview, we rasterize one *time-continuous triangle* (TCT) at a time, and sample it both spatially and in time on a per-tile basis. The design choice of processing one TCT at a time was simple as we would otherwise break the feed-forward principle of contemporary GPUs. Note again that our presentation focuses on rendering motion blur, however in Section 4, we will show that the same algorithm can be used to render other effects, such as depth of field and glossy reflections.

We assume that a TCT is defined at two different instants, $t = 0$ and $t = 1$. See Figure 1. This basically adds another "dimension" to a triangle. If the instants are interpreted as *different times* at a beginning and end of a frame, we can render images with motion blur, for example. The vertices in homogeneous clip space, i.e., after application of the projection matrix (but before division by w), at $t = 0$ are denoted $\mathbf{q}_k$, and at $t = 1$ they are denoted $\mathbf{r}_k$, $k \in \{0,1,2\}$. Furthermore, we assume that the vertices are interpolated linearly in this space,[†] which is equivalent to linear interpolation in world space. For a certain instant, $t \in [0,1)$, the vertices are: $\mathbf{p}_k(t) = (1-t)\mathbf{q}_k + t\mathbf{r}_k$. This is illustrated in Figure 1. All vertex attributes are linearly interpolated as well for different values of t. A major advantage of the TCT is that we only need to perform geometry processing once, which enables sampling of a triangle at *arbitrary* t-values, $t \in [0,1)$.

[†] This is in contrast to the approach taken by Sung et al. [SPW02], where interpolation takes place in screen space. As a consequence, they cannot handle perspective foreshortening of moving primitives correctly.

3.1. Overview

The basic algorithm works as follows for each time-continuous triangle (with respect to Figure 1), where each pixel is sampled at n different times, t_i, $i \in \{0,\ldots,n-1\}$:

1. Find tight bounding volume (BV) of time-continuous triangle (Section 3.3.1).
2. Compute time-dependent edge functions (Section 3.3.2).
3. For each *quad* (2×2 pixels) that overlaps the BV, fetch (or compute) the times, t_i, for the samples in that quad.
4. For each time, t_i, compute edge functions for the triangle $\Delta \mathbf{p}_0(t_i)\mathbf{p}_1(t_i)\mathbf{p}_2(t_i)$ using the time-dependent edge functions. Check whether the quad overlaps this triangle.
5. If overlap from previous step, linearly interpolate vertex attributes using t_i, and execute the pixel shader for the current quad.

Next, we present the details of our algorithm. We start by describing an inexpensive sampling strategy, and continue by developing robust and efficient rasterization of a TCT with Zmin/Zmax-culling. Finally, we introduce time-dependent textures, which can be used for shadow mapping, for example.

3.2. Sampling Strategy

In this section, we will describe our sampling strategy that makes it possible to use as few as four samples per pixel to get usable motion blur. However, our algorithm is not limited by this, and can be generalized to using more samples per pixel. Today, most GPUs have spatial antialiasing schemes with 4–8 samples per pixel or more, and each sample can even execute the pixel shader separately for higher quality. To keep the cost low, we simply want to add the time dimension to each of the samples for such hardware.

Our approach is to use n spatio-temporal samples, $\mathbf{s}_i = (x_i, y_i, t_i), i \in \{0, n-1\}$ per pixel, where (x_i, y_i) is the spatial position and t_i is the sample time. Contemporary GPUs always rasterize one *quad*, i.e., 2×2 pixels, at a time, since the GPU then can compute derivatives based on differences in x and y. Our algorithm clearly needs to comply with that requirement. Therefore, a certain time sample, t_i, must occur in *each* of the 2×2 pixels in a quad. Adjacent quads may preferably have a different set of times. Note that each sample has its own depth value, just as in super/multi-sampling.

All our spatio-temporal sampling patterns are completely deterministic, and do not change from frame to frame. In general, if a pixel uses n samples, we let each sample use a predetermined random time, such that $t_i \in T_i$, where T_i is the interval $\left[\frac{i}{n}, \frac{i+1}{n}\right)$ and $i \in \{0,\ldots,n-1\}$. This set will be used in one quad and gives us jittered sampling in time. For an adjacent quad, a new set of time samples $t_i' \in T_i$ is used.

Virtually all contemporary GPUs have some form of rotated grid supersampling (RGSS) implemented. This scheme fulfils the N-rooks requirement [Shi90], and it is illustrated in Figure 2. It is generally accepted that it gives good quality at a cost of only four samples per pixel. In the following,

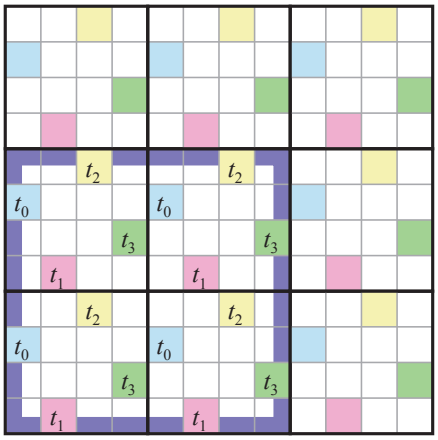


Figure 2: *3×3 pixels with RGSS sampling. One spatio-temporal sample lies in each colored subpixel. For the lower left quad (outlined in purple), the time samples, t_0, t_1, t_2, and t_3, all appear once in each pixel. This gives rise to a undesired quad-sized "pixelation" effect. Notice that all samples that belong to the same time interval, T_i, have the same color. For example, all samples in T_0 are light blue.*

we describe an example of our sampling scheme that uses RGSS. Note that our algorithm is not at all limited to this particular pattern, nor the number of samples. We focus instead on temporal sampling, while allowing different spatial sampling schemes. When adding time to each of these samples, the quad requirement makes the samples share four different times in each quad, and this basically means that the "pixels in time" will appear to have a size of 2×2 instead of the ideal case of 1×1 pixel.

To avoid this problem, we offset the quads depending on which time interval, T_i, the sample belong to. See Figure 3. For all samples within the time interval T_0, we use the standard quads, but for T_1 we offset the quad by one pixel to the right. For samples in T_2, we offset the quad one pixel upwards, and finally, and for T_3, the quad is offset one pixel to the right, and one pixel upwards. As can be seen, this guarantees that the set of time samples inside a pixel is different from neighboring pixels, which reduces the previously mentioned pixelation effect.

A common strategy to improve the quality of spatial antialiasing is to use larger filter kernels when computing the

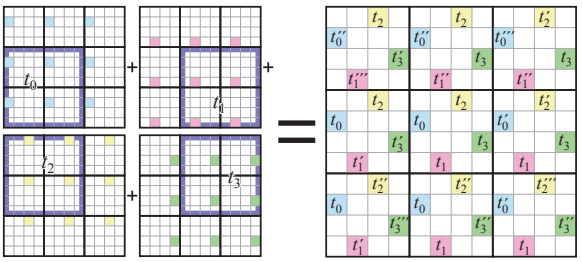


Figure 3: *By offsetting the quads (purple) for the different samples in time, we obtain a sampling scheme where neighboring pixels have different sets of times.*

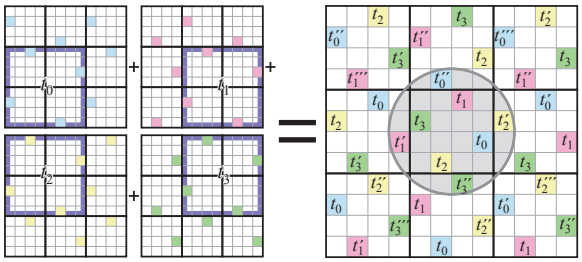


Figure 4: *We have redistributed the time samples inside the quads in order to avoid two samples with same color (i.e., belonging to the same time interval, T_i) being on the same subpixel column or row. To the right, this is clearly so for the samples inside the gray filter kernel. Note that only one quad is shown, while they in reality repeat over the entire pixel grid. Furthermore, the spatial sampling pattern repeats after 2×2 pixels, but the times of the samples have a longer period (typically, a 32×32 random table is used).*

final color of a pixel. When increasing the kernel for spatio-temporal filtering, we would ideally like to include samples with times different from the times inside the pixel, in order to improve the sampling resolution in the time dimension. In the following, we extend RGSS so that another four samples are used in the filter kernel, and we simply choose the four spatially closest samples. Note that our reasoning applies with minor modifications to any number of samples.

Assume we want to compute the final pixel color of the center pixel in Figure 3 by weighting together the samples with these times: $t_0, t_1, t_2, t_3, t_0', t_1'', t_2''$, and t_3'. From the figure, we notice that $t_0 \in T_0$ shares subpixel row with $t_0' \in T_0$, and $t_1 \in T_1$ shares subpixel column with $t_1'' \in T_1$, and so on. This is not ideal, at least not from an N-rooks perspective.

To remove this disadvantage, and thus improve sampling and filtering quality, we have devised a solution, which is shown in Figure 4. It is a straightforward task to verify that our sampling scheme gives eight different times for the eight spatio-temporal samples used for computing the final color of a pixel. As a final improvement of the time samples, consider two time samples belonging to the same time interval, T_i, inside the filter kernel. An example consists of t_0 and t_0'' (Figure 4), which both belong to the time interval $T_0 = [0, 0.25)$. To further improve the sampling quality, we make certain that $t_0 \in T_0^-$ and $t_0'' \in T_0^+$, where $T_0^- = [0, 0.125)$ and $T_0^+ = [0.125, 0.25)$. In general, we split T_i in the middle into T_i^- and T_i^+. This can be ensured when the sampling pattern is generated. The result is a sampling scheme with four generating samples per pixel, and with the larger filter kernel we obtain eight jittered time samples per pixel. Compared to RGSS, the added cost is essentially only more expensive filtering, which is done only once per pixel when the image has been rendered.

Note that the actual spatial positions can easily be redistributed to form another pattern. For example, we could use the pattern, inspired by Laine and Aila [LA06], shown to the right instead. In our experience, the *spatial* anti-aliasing would change a little bit compared to RGSS, but the *temporal* anti-aliasing remains very close to constant due to that we still get eight jittered time samples. Recall that the focus of our paper is not on the spatial sampling pattern.

Next, we describe how the filtering of the samples is done. Assume the colors of the samples inside a pixel are denoted, $\mathbf{c}_l^0$, where $l \in \{0,1,2,3\}$, and the colors of the four closest samples in the neighboring pixels by $\mathbf{c}_l^1$, again with $l \in \{0,1,2,3\}$. We use a *low-pass filter* to compute the final pixel color:

$$\mathbf{C} = w_0 \sum_{l=0}^{3} \mathbf{c}_l^0 + w_1 \sum_{l=0}^{3} \mathbf{c}_l^1 \qquad (1)$$

For all our tests, we use $w_0 = 5/32$ and $w_1 = 3/32$. This gives a good trade-off between spatial and temporal blurring. Naturally, it is simple to change the weights according to the purpose. We attempted to use another four samples from the neighboring pixels, but this did not give much of an effect on the quality.

It should be noted that the spatial positions can be jittered inside the subpixel using, for example, multi-jittering [Nie87]. By using a smaller grid of such spatial samples, we basically get a spatial interleaved sampling scheme [KH01]. Extending the ideas of this subsection to schemes with more samples per pixel is straightforward, and is therefore omitted.

3.3. Traversal of Time-Continuous Triangles

In this section, we describe how a time-continuous triangle (TCT) can be traversed, i.e., how the pixels inside a TCT can be found efficiently. Notice that the quadrilateral sides of a TCT are, in general, bilinear patches, and hence not necessarily planar. This makes clipping a TCT against the canonical view volume a complex procedure. Instead we decided to use edge functions [Pin88] derived directly from the homogeneous coordinates [OG97, MWM02], $\mathbf{q}_k$ and $\mathbf{r}_k$ with $k \in \{0,1,2\}$, of the TCT. This avoids clipping altogether. Using the two-dimensional axis-aligned bounding box of the TCT to limit the rasterization can make the traversal algorithm visit an excessive amount of pixels that are outside the TCT [WGER05].

Therefore, we propose a two-level algorithm for efficient rasterization of a TCT. First, a tight three-dimensional oriented bounding box (OBB) around the TCT is rasterized (Section 3.3.1). Second, for fragments inside the OBB, per-pixel evaluation of time-dependent edge functions (Section 3.3.2) follows. For samples inside the time-dependent edge functions, the pixel shader is executed.

3.3.1. OBB Traversal

We decided to use oriented bounding boxes (OBBs) around our TCTs to limit the number of pixels visited during traversal. To robustly handle cases where a TCT moves from in front of the viewer to behind the viewer, we rasterize only

the *backfaces* of the OBB without any depth testing (which is done in the next stage of our algorithm). This is similar to how shadow volume rendering [Cro77] handles the case when the viewer is inside a shadow volume and when a shadow volume intersects the near plane. For pixels covered by the OBB backfaces, we proceed to testing with time-dependent edge functions (next section).

Our method for computing a tight OBB is simple and gives very good results in the majority of all cases. All computations are done before division by w, and so we use the (x,y,w)-coordinates of the vertices of the TCT. The major axis of the OBB is computed as the difference between the center of the starting and ending triangle of the TCT. If this vector is near zero, an axis-aligned bounding box (AABB) is computed instead. Otherwise, we project the edges of the TCT onto the plane whose normal is the major axis. For the second axis of the OBB, we use the longest projected edge. Again, if there is no such non-zero vector, we revert to using an AABB. The third axis is obtained with a cross product. This algorithm can be implemented in a geometry shader.

Discussion Several different possibilities for this stage of the algorithm were explored. We tried using the convex hull of the homogeneous coordinates of the TCT, and we devised a hardware-friendly algorithm for this. However, it is very difficult to obtain a robust algorithm without handling a large set of special cases. In addition, the starting triangle of the TCT may be behind the camera, and in such situations, it is not even clear what the definition of the convex hull using homogeneous edge functions is. Another possibility is to use *bounding prisms* (BP) as used for caustic primitives [EAMJ05], for example. The construction algorithm for BPs works well for typical caustic rendering, but for more general settings, we have found that BPs with infinite size can result. In addition, the computation of BPs was more costly than OBBs. Hence, using OBBs is a good trade-off in terms of robustness, speed, and simplicity.

3.3.2. Time-Dependent Edge Functions

Due to the traversal from the previous section, we know that a quad overlaps with the OBB of the TCT. Now, we need to determine whether the samples, $\mathbf{s}_i$ (see beginning of Section 3.2 for the definition of samples), overlaps with the TCT. To be able to do this efficiently, we introduce *time-dependent edge functions*.

First, recall that the vertices, $\mathbf{q}_k$ and $\mathbf{r}_k$, $k \in \{0,1,2\}$, are in homogeneous clip space after application of the projection matrix (but before division by w), and that the camera is located in $(0,0,0)$. Furthermore, let us introduce a "truncated" variant of a vector $\mathbf{v}$ as $\hat{\mathbf{v}} = (v_x, v_y, v_w)$. This simply means that we create a three-dimensional vector from a four-dimensional by skipping the z-coordinate.[‡]

[‡] Note that due to the projection matrix (e.g., OpenGL or DirectX), this vector is in a scaled and translated camera space. This can be verified by examining the elements of the projection matrix.

The edge function through two vertices, say $\hat{\mathbf{p}}_0$ and $\hat{\mathbf{p}}_1$, is then [OG97, MWM02]:

$$e(x,y,w) = (\hat{\mathbf{p}}_1 \times \hat{\mathbf{p}}_0) \cdot (x,y,w) = ax + by + cw, \qquad (2)$$

where $(\hat{\mathbf{p}}_1 \times \hat{\mathbf{p}}_0) = (a,b,c)$. Now, since the vertices are functions of time, $\hat{\mathbf{p}}_k(t) = (1-t)\hat{\mathbf{q}}_k + t\hat{\mathbf{r}}_k$, we simplify the expression for the edge function parameters:

$$\begin{aligned}(a,b,c) = (\hat{\mathbf{p}}_1 \times \hat{\mathbf{p}}_0) &= ((1-t)\hat{\mathbf{q}}_1 + t\hat{\mathbf{r}}_1) \times ((1-t)\hat{\mathbf{q}}_0 + t\hat{\mathbf{r}}_0) \\ &= t^2\hat{\mathbf{f}} + t\hat{\mathbf{g}} + \hat{\mathbf{h}},\end{aligned} \qquad (3)$$

where:

$$\begin{aligned}\hat{\mathbf{m}} &= \hat{\mathbf{q}}_1 \times \hat{\mathbf{r}}_0 + \hat{\mathbf{r}}_1 \times \hat{\mathbf{q}}_0 \\ \hat{\mathbf{h}} &= \hat{\mathbf{q}}_1 \times \hat{\mathbf{q}}_0, \\ \hat{\mathbf{f}} &= \hat{\mathbf{h}} - \hat{\mathbf{m}} + \hat{\mathbf{r}}_1 \times \hat{\mathbf{r}}_0, \\ \hat{\mathbf{g}} &= -2\hat{\mathbf{h}} + \hat{\mathbf{m}}, \end{aligned} \qquad (4)$$

This means that we have simple expressions for all the edge function parameters, (a,b,c). For example, we have: $a(t) = f_x t^2 + g_x t + h_x$. Note that $\hat{\mathbf{f}}$, $\hat{\mathbf{g}}$, and $\hat{\mathbf{h}}$ can be computed in the triangle setup. For a specific time, t_i, and spatial sample position, (x_i, y_i), we now arrive at the time-dependent edge function:

$$e(\mathbf{s}_i) = e(x_i, y_i, t_i) = a(t_i)x_i + b(t_i)y_i + c(t_i), \qquad (5)$$

where $w = 1$ since we now are dealing with screen space (x,y)-coordinates.

Once the three edge functions, $e_j(\mathbf{s}_i)$, have been computed, we can determine whether a sample, $\mathbf{s}_i$, is inside the TCT at time t_i. If this is true, we linearly interpolate the vertex attributes of the starting and ending triangle of the TCT with respect to t_i, and pass them on downwards the pipeline.

Note that since each time-dependent edge function is defined by four vertices, cracks "in time" between two TCTs sharing an edge can be avoided using a simple tie-breaker rule [MWM02]. However, to avoid small numerical inaccuracies when evaluating the expressions in Equation 4, we also make sure that two TCTs sharing an edge always compute the parameters $\hat{\mathbf{f}}$, $\hat{\mathbf{g}}$, $\hat{\mathbf{h}}$ in exactly the same way. This is done by swapping $\mathbf{q}_1$ and $\mathbf{q}_0$ so that the first point is always the one with smallest x-value before calculation of the parameters starts. If the x's are equal, testing continues with y, and so on.

Discussion Another possible solution would be to interpolate edge functions in *screen space*. Consider one edge function, $e_0(x,y) = a_0x + b_0y + c_0$, for the first triangle, $\Delta\mathbf{q}_0\mathbf{q}_1\mathbf{q}_2$, and the corresponding edge function, $e_1(x,y) = a_1x + b_1y + c_1$ for triangle $\Delta\mathbf{r}_0\mathbf{r}_1\mathbf{r}_2$. To find the edge function for a specific time, $t \in [0,1)$, one could interpolate the edge functions parameters, e.g., $a(t) = (1-t)a_0 + ta_1$, and so on. However, this does not take perspective foreshortening into account, and in addition, it requires the TCT to be clipped, which we also want to avoid.

For simplicity, we have limited ourselves to linear interpolation of vertex positions and attributes. To get curved motion blur, we can use our technique together with an accumulation buffer for faster performance. Higher-order interpola-

tion, such as quadratic or cubic Bézier curves, is of course also possible. Besides the actual interpolation, only the OBB computation need to be altered, since more vertices need to be processed.

3.4. Zmin/Zmax-Culling

Zmin- and Zmax-culling [AMS03, Mor00] are crucial for good depth buffer and texture access performance. Therefore, one of our goals has been to make stochastic rasterization work with this type of algorithms. Hence, a conservative estimate of minimum and maximum depth inside a tile (often 8×8 pixels) for a time-dependent triangle is needed.

We limit our discussion here to Zmax-culling, where a conservative estimate of the minimum depth value, denoted z^{tri}_{min}, of a triangle inside a tile is needed. The maximum of the depth values inside a tile is denoted z_{max}. If $z^{tri}_{min} > z_{max}$ we can avoid processing the triangle in that tile. Extending this to Zmin-culling is straightforward.

A conservative estimate of the minimum depth value of a triangle inside a tile is simply the minimum of the vertices of the triangle being rendered. Let us denote this value by z^{v}_{min}, where the superscript indicates *vertices*. However, this can become overly conservative, for example, when rendering a large triangle with a normal almost perpendicular to the view direction. To improve this, one can also compute the depth at the tile corners using the plane equation of the triangle, and computing the minimum of these. Let us call this value z^{c}_{min}, where the superscript indicates *corners*. An improved estimate of the minimum depth of the triangle inside a tile is then:

$$z^{tri}_{min} = \max(z^{v}_{min}, z^{c}_{min}). \tag{6}$$

This is a commonly used technique. In the case of rendering a TCT, we again evaluate Equation 6, but the computation of the terms in the max-function becomes a bit more complex. The value z^{v}_{min} is computed using the six vertices of the TCT, and z^{c}_{min} is computed using the plane equations of the starting and the ending triangles of the TCT. This is conservatively correct as long as the orientations of the starting and ending triangle are the same. When this is not true, you may not always get a correct conservative value. One reason for this, is that the depth at the corners of a tile can become unbounded when the orientation of a TCT changes from, for example, backfacing to frontfacing. An example is illustrated in Figure 5.

However, there is a straightforward solution to this. If there is *no* change in orientation, we compute z^{tri}_{min} using Equation 6. In the case of a change in orientation, we simply use the minimum of the depths at the vertices of the starting and ending triangles, i.e., $z^{tri}_{min} = z^{v}_{min}$.

Discussion In the description above, we have assumed that we store one z_{max}-value per tile for all different times of the sample inside a tile. An alternative would be to store, for example, four values per tile: z^{i}_{max}, $i \in \{0,1,2,3\}$, where z^{i}_{max} is the max-value of the depths belonging to the time interval, T_i. In a sense, the low-resolution depth buffer that contains z_{max}-values, is extended in the time dimension. While this is clearly possible and would provide more efficient culling, we have decided to leave this for future work, since it does not fit well with contemporary GPUs as they store only one z_{max}-value per tile.

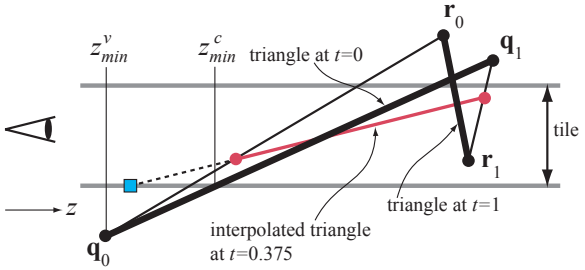


Figure 5: *A time continuous triangle (TCT) defined by a starting triangle, $\Delta\mathbf{q}_0\mathbf{q}_1\mathbf{q}_2$, and an ending triangle, $\Delta\mathbf{r}_0\mathbf{r}_1\mathbf{r}_2$, here shown in two dimensions. Due to that these triangles do not have the same orientation, problems in Zmax-culling can occur. Normally, we compute $z^{tri}_{min} = \max(z^{v}_{min}, z^{c}_{min})$. In this case, this is not correct, since at, e.g., $t = 0.375$, the true depth at the tile corner (blue square) is smaller than z^{c}_{min} which is computed using the plane equations of the two triangles of the TCT. Our solution is simply to use $z^{tri}_{min} = z^{v}_{min}$ when the orientation of the triangles changes. This gives a conservative estimate.*

3.5. Time-Dependent Textures

Motion blurred geometry without motion blurred shadows spoils the entire concept, almost. Hence, we would like to support motion blurred shadows in our spatio-temporal framework. Shadow mapping [Wil78] is a commonly used technique for (static) shadow generation. Lokovic and Veach [LV00] introduce deep shadow maps, where motion blur is handled by associating a random time with every shadow map sample within a texel. The time samples are averaged together, which means that the time dimension is reduced to a single blurred value. As a consequence, the authors concluded that this approach will be correct only for static shadow receivers as seen from the light source.

We alleviate this problem by introducing time-dependent textures, which holds a set of time samples per texel and support time-dependent reads and writes. When generating the shadow map, we use the sampling strategy of Section 3.2 and store n depth values per texel, each associated with a unique time, t_s. When rendering from the camera, the visible sample will be associated with a time t_i. During time-dependent texture lookup, we ensure that the screen space sample, $t_i \in T_i$, access the shadow map sample with time t_s also in T_i. This will reduce self-shadowing artifacts for cases with moving receivers. With n jittered time samples per texel in screen and light space, our approach guarantees that $|t_i - t_s| < 1/n$. If more time samples are added per pixel, the result converges towards the correct image. With uniform time sam-

pling, $t_i = t_s$, the images instead contain apparent strobing artifacts.

In general, time-dependent textures are useful as render targets for dynamically generated effects, where we need to store time-dependent depth or color values. A simple technique for generating reflections for curved geometry is to first render a cube map from the position of the object, and then access this map with the reflection vectors during rendering of reflecting objects. If we use time-dependent textures for cube map generation and lookups, we can handle correct motion blurred reflections, even when both the reflection vector and the cube map changes over time. See Figure 8 for an example.

4. Results

We have implemented a subset of OpenGL 2.0 in a functional simulator in C++. Currently, there are two ways to specify vertex positions. For the first method, you set all your transforms (model + view + projection), and then ask the API to "remember" the composite matrix. This is the transform matrix for $t = 0$. After that you set the all the matrices again (this time for $t = 1$), and then render your objects. The other method simply specifies a double set of vertex positions. We call one such rendering an SR pass.

Note that we use the abbreviation ABT for accumulation buffering of static images. However, we can also accumulate images rendered with SR. We call this *stochastic rasterization accumulation* (SRA).

We emphasize the fact that still images only reveal a small part of the perceived image quality. Since our target is real-time rendering, we refer the reader to the videos of this submission in order to judge the quality of our motion blur, depth of field, and glossy reflections.

For Zmax-culling, we have not gathered statistical results. We note that if the geometry is static, the algorithm works as well as the old Zmax-algorithm. For moving geometry, culling will occur when possible, but there is really no algorithm to compare to, so this has been omitted for now.

In the following, we report our results for motion blur, depth of field, and glossy planar reflections. It should be noted that our framework can only handle one extra dimension at a time, and therefore only one effect at a time. For example, we cannot handle DOF and motion blur in the same image and pass.

4.1. Motion Blur

For our motion blur rendering results, we use only a single SR pass with four samples per pixel, except where otherwise mentioned.

Cook et al. [CPC84] point out a number of hard cases of motion blur: specular highlights, intersecting objects, shadows and reflections. As seen in Figure 6A and B, our algorithm handles these cases due to its stochastic nature. The chain elements intersect, and have complex motion, and the staircase scene shows specular highlights and blurred shadows using time-dependent shadow maps. Note that these images were rendered using only four samples per pixel. As the algorithm allows sampling at arbitrary times within the frame, strobing artifacts are replaced by (less noticeable) noise without increasing the sampling cost. It should be noted that the algorithm correctly handles scenes where both the camera and geometry are animated as the total motion simply becomes composite transform matrices applied at $t = 0$ and $t = 1$.

In Figure 7, a simple model of a textured wheel is shown. The model is translated and its texture coordinates rotated, which means that motion blur is both obtained due to the translation and rotation. This kind of effect is not handled correctly by methods where a static image is rendered first, and then that image is blurred according to motion vectors [SSC03]. This example clearly shows the flexibility and power of our method, and indicates that the quality converges towards the reference solution (bottom row in Figure 7) in this case, which is a major advantage.

An example of blurred reflections from moving objects using a time-dependent cube map is shown in Figure 8.

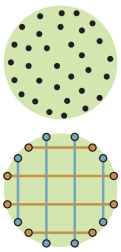
Since the TCT uses linear interpolation, the algorithm cannot render higher order movement directly. For example, a rotating sphere gets a blurry edge where the relative motion is largest, and a fast circular arc movement of, say, a sword will get a triangular motion trail. Artifacts from such non-linear motion can be found in our video. These situations can be improved using an SRA technique, and generating TCTs for uniform subintervals of the time inside a frame. Our video shows that stochastic rasterization quickly resembles the ground truth, while accumulation buffering techniques suffer severely from strobing artifacts in these cases.

In Figure 9, we compare motion blur renderings with 4 samples per pixel against 8 samples per pixel in a single pass. Naturally, the quality is higher the more samples being used.

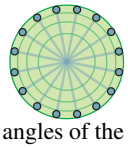
Blurred shadow maps inherit the shadow bias problem from standard shadow maps, which is somewhat enhanced by the added uncertainty in time. However, already with four jittered time shadow samples per pixel, we can render nice-looking, blurred shadows suitable for real-time content.

4.2. Depth of Field

Computing images with depth of field (DOF) is computationally expensive. Haeberli and Akeley [HA90] render DOF using an accumulation buffer with point sampling on the aperture of the camera lens, which is illustrated as a light green area to the right. In this example, we use 32 uniform random points. For DOF with our algorithm, we use an SRA approach, i.e., we accumulate several images from SR passes. With our SR algorithm, we can instead sample an *entire line* on the lens area in a single pass. This is illustrated to the right with four horizontal and four vertical



lines. Doing this, is a simple matter of setting the camera matrix for the start point of the line, then ask the API to "remember" the composite matrix, and then set the camera matrix for the end point of the line. This gives a DOF-effect in the direction of the line. For example, if we use a horizontal line, the DOF-effect will only be horizontal, but it will be stochastically sampled, i.e., with good quality. Using a multi-pass technique, we can average the results from a number of "line samples."

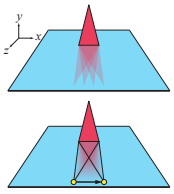


Using the line sample scheme above, it is quite clear that banding artifacts can appear, both horizontally and vertically. For best results, we need to sample using as long lines as possible, while also maximizing the number of angles of the lines. One such sampling pattern is shown to the right. However, this scheme has increased sample density the closer to the center you get. Our solution is to redistribute the sample times, t_i, which is illustrated by the circles. Theoretically, this should be done with a $\sqrt{t}$-like function being reflected around $(0.5, 0.5)$. In practice, we do it with a `smoothstep`-function, $s_i = t_i^2(3-2t_i)$, which is accurate enough. When this transform has been applied, the time-dependent edge functions use s_i (instead of t_i) for inclusion testing as usual.

To the best of our knowledge, we have not seen any DOF algorithm using line samples on the lens aperture. In our experience, this works really well already using only eight lines, i.e., eight passes, with four samples per pixel. This should be compared to grid point sampling the lens, which can require more than 100 samples to get stable results during animation [HAM06a]. In our experience, however, similar results to ours can be obtained with uniform random sampling over the lens using 32 image passes. Again, note that such an approach requires the scene geometry to be processed 32 times. See Figure 6C for an example of DOF rendering using our algorithm.

4.3. Glossy Reflections

For rendering planar glossy reflections, Diefenbach and Badler [DB97] suggested that the reflected object is sheared in the x- and z-directions, with increased shearing effect the smaller y gets. This is illustrated for shearing in x to the right. Rendering the scene many times with different amounts of shear gives glossy reflections in the accumulated image. Our stochastic rasterizer can again be used to advantage even in this case, using an SRA approach, as shown in the bottom right illustration. By using the concept of line samples from the previous subsection, we realize that a shearing pass in x is done as a line sample, where the outer vertex points are sheared the maximum amount in both directions. In practice, the shearing effect can be computed in the vertex shader. Figure 6D shows this effect using stochastic rasterization in the x-direction, and only four passes in z. As shown in the animation, no banding artifacts are noticeable.



4.4. Bandwidth Analysis

One can easily imagine that the random nature of our algorithm can break several of the features in a modern GPU, which exploits coherency in the rendering. This includes buffer compression and texture cache performance, for example.

The two scenes with most motion on textured objects are the Sponza DOF (Figure 6C), and the rotating/translating wheel (Figure 7). For all our tests, we used a single 6 kB texture cache, which is perfectly reasonable (for comparison, a Geforce 8800 has 8 kB per multi-processor). Also, our algorithm used four samples per pixel, while ABT used four passes, which also gives four samples per pixel. In the wheel scene, our algorithm used 225 MB of off-chip texture bandwidth, while ABT used 314 MB. For Sponza DOF, the advantage of our algorithm becomes more pronounced: ABT used 2314 MB, while our algorithm used only 1056 MB. In addition to this improvement, we believe that a texture cache coherent rasterization order can improve our numbers further. This is left for future work at this point.

For the depth buffer, we implemented depth offset compression (DOC) [HAM06b]. When using this on the chain scene (Figure 6A), the depth is compressed down to 62.5%. Hasselgren and Akenine-Möller report compression rates of about 60% on a set of static scenes [HAM06b]. This gives an indication that depth buffer compression can work quite well. However, we believe that clever new algorithms can be implemented to further increase compression. For example, using four layers in DOC could help quite a bit. An interesting avenue for future work would be to compress all samples in each time subinterval separately. For example, we can compress all samples inside the time interval [0.0,0.25) separately. This would increase the coherence, and improve compression. We note that this is important, and we would like to investigate buffer compression for SR in the future.

5. Discussion

Our algorithm for stochastic rasterization (SR) should be seen as a complement to standard rasterization. It is a feature that the programmer can turn on exactly when needed. For parts of a scene with little or no relative motion, standard rasterization can be used together with multi-sampling.[§] This gives spatial antialiasing at a low cost. However, for parts with faster relative motion, the more expensive stochastic rasterization can be activated with supersampling to obtain spatio-temporal antialiasing. Thus, the rendering can be seen as a combination of multi-sampling and supersampling. Note that for motion-blurred regions, the spatial positions of the samples do not matter that much. Instead, it is the temporal distribution of the samples that determine quality. For

[§] We use the following terminology: for multi-sampling, the pixel shader is executed only *once per pixel*, while for supersampling, the pixel shader is executed *once per sample*.

static parts, we get the same quality as using spatial antialiasing only, and our algorithm is not directly dependent on any particular scheme. We choose RGSS because it gives good spatial antialiasing, and is accepted in the industry.

We also want to emphasize a few very important features of using stochastic rasterization. First, if stochastic rasterization use n samples per pixel, we can compare this to accumulation buffering techniques (ABT) rendered with n passes, where each pass renders a static image. Our video clearly shows that the strobing artifacts of ABT are more noticeable. However, there is also a significant advantage in terms of sending geometry over the bus and geometry processing. With stochastic rasterization we only send the geometry once, and transform that geometry twice in the geometry shader (using different matrices for $t = 0$ and $t = 1$). In contrast, ABT would send the geometry n times over the bus, and process the geometry n times. This makes for a substantial improvement already at four samples per pixel.

Note that ABT (as defined in the previous paragraph) converges to a correct result the more static images that are accumulated. Our SR algorithm can render n images and accumulate them as well, but in our case convergence will be much quicker due to the stochastic nature of our algorithm. Furthermore, SR degrades more gracefully than ABT, which makes SR useable over a wider range of sampling rates.

A further use of SR is "practically frameless rendering" [WZM95], which is described briefly in Section 2. Assuming that it is possible to disable writing to a specific set of pixels or samples, we can use SR to render motion blurred triangles into, say, only every 4th pixel. This would give better image quality compared to the original approach, since SR can provide stochastic sampling of the geometry in each rendering pass.

Direct hardware support of our stochastic rasterization algorithm would require rather moderate additions since we could leverage on existing supersampling and multisampling hardware in contemporary GPUs. Transforming and setting up a time-continuous triangle can be done in the geometry shader, as well as computing an OBB. For a full implementation, our sampling/filtering, Zmin/Zmax-culling, and time-dependent texture lookups would require some small hardware modifications. We did a partial implementation of the "inner loop" of our algorithm in a fragment program. The time of the sample is computed through a texture lookup, and we interpolate the time-dependent edge functions based on that time, evaluate the interpolated edge functions based on spatial coordinates, and finally compute the perspective-correct barycentric coordinates for the sample. We assumed that the edge-function setup was done in a previous step, and used uniform parameters to pass per-triangle data in lack of better alternatives. By analyzing this program using a shader performance tool, `nvshaderperf`, we found that this shader program took 11 clock cycles to execute on a GeForce7800, with an expected fillrate of 872.73 Mpixels/s. This kind of performance can fill the screen eight times in 100 fps at 1024×1024. With a GeForce 8800, this would be much higher, but `nvshaderperf` did not support this card when we did our tests. Our conclusion is that we need native hardware support for time-dependent edge functions and interpolation using these to reach higher performance. In our current implementation, we practically perform inside-outside test and interpolation twice (first using native hardware, and then in the pixel shader), and it would be nice to avoid that.

For the pseudo-random time pattern, we use a fixed time table of 32×32 random numbers in the interval [0,1), as described in Section 3.2. We have not seen any visual difference between a 128×128 and a 32×32 table. Smaller tables start to alter the image quality. Due to our sampling strategy with the interval [0,1) split into eight subintervals, and random sampling done inside each such subinterval, we already have three bits of the random number implicitly. Empirically, we found that adding another three bits is enough per x and y for the sampling locations. This means that we need a table of $32 \times 32 \times 6$ bits constant pseudo-random numbers. In our experience, such fixed tables can be realized with very few gates.

6. Conclusion and Future Work

One could argue that all we do in this paper is to implement "stochastic rasterization" (SR)—a 20-year old technique [Coo86,CCC87]. However, we contribute with several techniques well-suited for GPUs. We develop tight-fitting robust OBBs around moving triangles, and introduce time-dependent edge functions for efficient inside-test and interpolation. In addition, we construct a clever scheme using only four samples per pixel, which gives eight samples (perfectly jittered in time) for pixel reconstruction, and still comply to using 2×2 quads. Furthermore, we create a Zmin/Zmax-culling variant, which is crucial to good performance today. We show that SR can be used for depth-of-field, glossy reflections, and motion blur with shadows, highlights, & reflections. In conclusion, we strongly believe our research advances the field of rasterization.

Even though we think that rasterization and ray tracing are somewhat complementary techniques, there is an ongoing debate about which is *the* preferred rendering algorithm. We have showed that SR is a powerful alternative for motion blur, since we can sample the moving triangles at any particular time. For ray tracing, spatial data structures need to be partly rebuilt for every instant of time where we want to sample the geometry, and this is expensive and impractical.

For future work, we want to investigate how texture-cache coherent rasterization order can be adapted to the case of SR, and work on depth and color buffer compression. Furthermore, we want to combine SR with delay streams [AMN03] for better culling. It would also be important to analyze shader branch efficiency. Also, when motion blur is used, the acceptable frame rate can, in general, be lower compared to not using motion blur. It would be interesting to see whether this can be used to conserve energy in mobile devices.

Acknowledgements

We acknowledge support from the Swedish Foundation for Strategic Research, Vetenskapsrådet, and NVIDIA's Fellowship program. Thanks to Timo Aila & anonymous reviewers for their helpful comments.

References

[AMN03] AILA T., MIETTINEN V., NORDLUND P.: Delay Streams for Graphics Hardware. *ACM Transactions on Graphics, 22*, 3 (2003), 792–800.

[AMS03] AKENINE-MÖLLER T., STRÖM J.: Graphics for the Masses: A Hardware Rasterization Architecture for Mobile Phones. *ACM Transactions on Graphics, 22*, 3 (2003), 801–808.

[ATI04] ATI: *Radeon X800: High Definition Gaming*. ATI Technologies White Paper, 2004.

[BFMZ94] BISHOP G., FUCHS H., MCMILLAN L., ZAGIER E. J. S.: Frameless Rendering: Double Buffering Considered Harmful. In *Proceedings of ACM SIGGRAPH 1994* (1994), pp. 175–176.

[Cat84] CATMULL E.: An Analytic Visible Surface Algorithm for Independent Pixel Processing. In *Computer Graphics (Proceedings of ACM SIGGRAPH 84)* (1984), pp. 109–115.

[CCC87] COOK R. L., CARPENTER L., CATMULL E.: The Reyes Image Rendering Architecture. In *Computer Graphics (Proceedings of ACM SIGGRAPH 87)* (1987), pp. 96–102.

[Coo86] COOK R. L.: Stochastic Sampling in Computer Graphics. *ACM Transactions on Graphics, 5*, 1 (1986), 51–72.

[CPC84] COOK R. L., PORTER T., CARPENTER L.: Distributed Ray Tracing. In *Computer Graphics (Proceedings of ACM SIGGRAPH 84)* (1984), pp. 137–145.

[Cro77] CROW F.: Shadow Algorithms for Computer Graphics. In *Computer Graphics (Proceedings of ACM SIGGRAPH 77)* (July 1977), pp. 242–248.

[DB97] DIEFENBACH P., BADLER N.: Multi-Pass Pipeline Rendering: Realism for Dynamic Environments. In *Symposium on Interactive 3D Graphics* (1997), pp. 59–70.

[Dem04] DEMERS J.: *Depth of Field: A Survey of Techniques*. GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics, 2004, ch. 23, pp. 375–390.

[DWS*88] DEERING M., WINNER S., SCHEDIWY B., DUFFY C., HUNT N.: The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics. In *Computer Graphics (Proceedings of ACM SIGGRAPH 88)* (1988), pp. 21–31.

[EAMJ05] ERNST M., AKENINE-MÖLLER T., JENSEN H. W.: Interactive Rendering of Caustics using Interpolated Warped Volumes. In *Graphics Interface* (2005), pp. 87–96.

[Gra85] GRANT C. W.: Integrated Analytic Spatial and Temporal Anti-Aliasing for Polyhedra in 4-Space. In *Computer Graphics (Proceedings of ACM SIGGRAPH 85)* (1985), pp. 79–84.

[HA90] HAEBERLI P., AKELEY K.: The Accumulation Buffer: Hardware Support for High-Quality Rendering. In *Computer Graphics (Proceedings of ACM SIGGRAPH 90)* (1990), pp. 309–318.

[HAM06a] HASSELGREN J., AKENINE-MÖLLER T.: An Efficient Multi-View Rasterization Architecture. In *Eurographics Symposium on Rendering* (2006), pp. 61–72.

[HAM06b] HASSELGREN J., AKENINE-MÖLLER T.: Efficient Depth Buffer Compression. In *Graphics Hardware* (2006), pp. 103–110.

[JK01] JONES N., KEYSER J.: Real-Time Geometric Motion Blur for a Deformning Polygonal Mesh. In *Computer Graphics International* (2001), pp. 26–31.

[KB83] KOREIN J., BADLER N.: Temporal Anti-Aliasing in Computer Generated Animation. In *Computer Graphics (Proceedings of ACM SIGGRAPH 83)* (1983), pp. 377–388.

[KH01] KELLER A., HEIDRICH W.: Interleaved Sampling. In *Eurographics Workshop on Rendering* (2001), pp. 269–276.

[LA06] LAINE S., AILA T.: A Weighted Error Metric and Optimization Method for Antialiasing Patterns. *Computer Graphics Forum, 25*, 1 (2006), 83–94.

[Lov05] LOVISCACH J.: Motion Blur for Textures by Means of Anisotropic Filtering. In *Eurographics Symposium on Rendering* (2005), pp. 7–14.

[LV00] LOKOVIC T., VEACH E.: Deep Shadow Maps. In *Proceedings of ACM SIGGRAPH 2000* (2000), pp. 385–392.

[Mor00] MOREIN S.: ATI Radeon HyperZ Technology. In *Workshop on Graphics Hardware, Hot3D Proceedings* (August 2000), ACM Press.

[MWM02] MCCOOL M. D., WALES C., MOULE K.: Incremental and Hierarchical Hilbert Order Edge Equation Polygon Rasterization. In *Graphics Hardware* (2002), pp. 65–72.

[Nie87] NIEDERREITER H.: Point Sets and Sequences with Small Discrepancy. *Monatshefte für Mathematik*, 104 (1987), 273–337.

[OG97] OLANO M., GREER T.: Triangle Scan Conversion using 2D Homogeneous Coordinates. In *Workshop on Graphics Hardware* (1997), pp. 89–95.

[Pin88] PINEDA J.: A Parallel Algorithm for Polygon Rasterization. In *Computer Graphics (Proceedings of ACM SIGGRAPH 88)* (August 1988), ACM, pp. 17–20.

[Shi90] SHIRLEY P.: *Physically Based Lighting Calculations for Computer Graphics*. PhD thesis, University of Illinois at Urbana Champaign, December 1990.

[SPW02] SUNG K., PEARCE A., WANG C.: Spatial-Temporal Antialiasing. *IEEE Transactions on Visualization and Computer Graphics, 8*, 2 (2002), 144–153.

[SSC03] SHIMIZU C., SHESH A., CHEN B.: *Hardware Accelerated Motion Blur Generation*. Tech. Rep. 05-03, Computer Science Department, University of Minnesota at Twin Cities, 2003.

[WGER05] WEXLER D., GRITZ L., ENDERTON E., RICE J.: GPU-Accelerated High-Quality Hidden Surface Removal. In *Graphics Hardware* (2005), pp. 7–14.

[Wil78] WILLIAMS L.: Casting curved shadows on curved surfaces. In *Computer Graphics (Proceedings of ACM SIGGRAPH 78)* (1978), ACM Press, pp. 270–274.

[WZ96] WLOKA M., ZELEZNIK R.: Interactive Real-Time Motion Blur. *The Visual Computer, 12*, 6 (1996), 283–295.

[WZM95] WLOKA M. M., ZELEZNIK R. C., MILLER T.: *Practically Frameless Rendering*. Tech. rep., 1995.

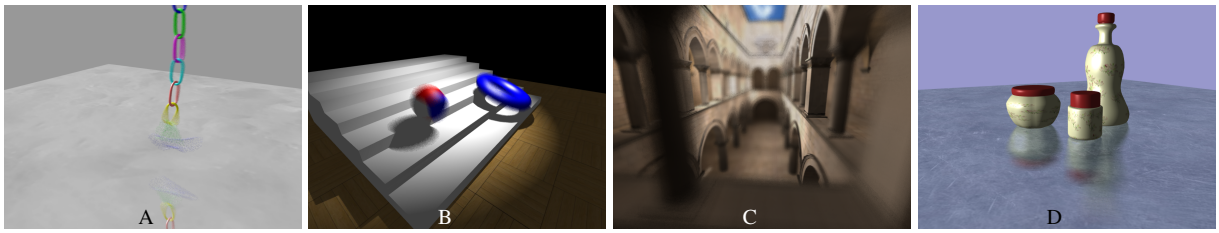


Figure 6: *Our stochastic rasterization algorithm can render images with a variety of effects. A+B: motion blur with only four samples per pixel. Notice the motion blurred reflections in A, and the motion blurred shadows and highlights in B. C: depth of field rendered with only eight passes with four samples per pixel. D: glossy planar reflections using four passes. Note that the target is real-time graphics, and so to be fair, the quality is best judged from our video.*

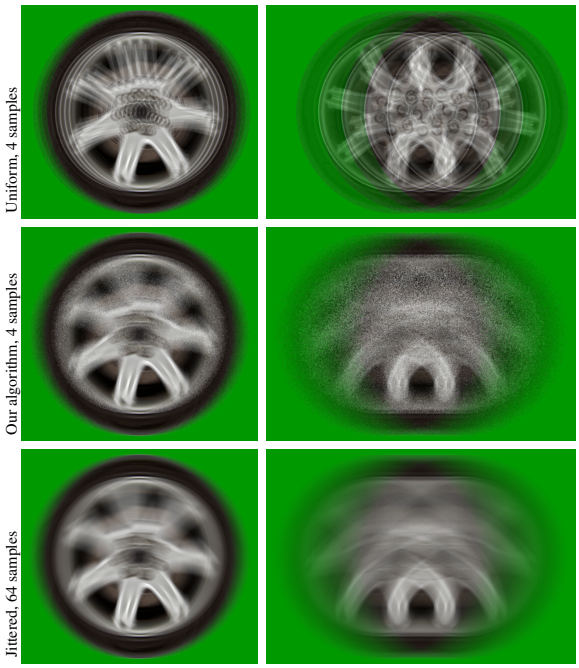


Figure 7: *Motion blur caused by both translation and rotation. Note the strobing artifacts obtained using four samples per pixel with uniform sampling, i.e., similar to Wexler et al's method [2005]. The left column shows a slow motion, while the right shows a five times faster motion.*

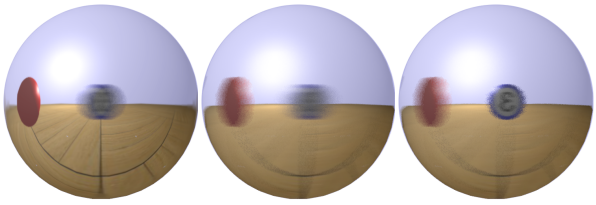


Figure 8: *A moving blue ball and a static red ball are reflected in a chrome sphere using cube mapping.* ***A****. Static camera. Notice the blurred blue ball and the sharp red ball.* ***B****. The camera is moving in the same path as the blue ball so that there is no relative motion between them. With a standard cube map, both balls appear blurred.* ***C****. With a time-dependent cube map, the reflected blue ball approaches the correct result, which is a sharp reflection. Four samples per pixel are used in all these examples.*

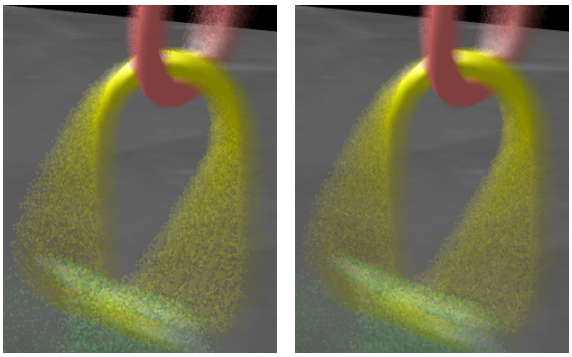


Figure 9: *Motion blur rendering in a single pass. Left: 4 samples per pixel. Right: 8 samples per pixel. Note that the motion is about 100 pixels wide in the fastest moving region. Recall that to halve the variance, we need to quadruple (4×) the number of samples.*