High Performance Graphics (2010)
M. Doggett, S. Laine, and W. Hunt (Editors)

Real-time Stochastic Rasterization on Conventional GPU Architectures

M. McGuire[†1,2], E. Enderton[1], P. Shirley[1], and D. Luebke[1]

[1]NVIDIA
[2]Williams College

Abstract

This paper presents a hybrid algorithm for rendering approximate motion and defocus blur with precise stochastic visibility evaluation. It demonstrates—for the first time, with a full stochastic technique—real-time performance on conventional GPU architectures for complex scenes at 1920×1080 HD resolution. The algorithm operates on dynamic triangle meshes for which per-vertex velocity or corresponding vertices from the previous frame are available. It leverages multisample antialiasing (MSAA) and a tight space-time-aperture convex hull to efficiently evaluate visibility independently of shading. For triangles whose motion crosses the camera plane, we present a novel 2D bounding box algorithm that we conjecture is conservative. The sampling algorithm further reduces sample variance within primitives by integrating textures according to ray differentials in time and aperture.

1. Introduction

Photographs exhibit both motion blur and defocus blur because cameras integrate across non-zero shutter times and apertures. High-quality blur effects are ubiquitous and important in film and offline rendering algorithms, but are rare in real-time rendering algorithms, which rely on instantaneous shutters and pinhole cameras. Instead, most real-time systems approximate motion and defocus blur with image-space post-processing that suffers robustness problems. Despite advances in real-time ray tracing such as NVIDIA's OptiX engine, real-time distribution ray tracing remains challenging. Several researchers have proposed extending rasterization algorithms or hardware to stochastically sample shutter interval and aperture [WGER05, AMMH07, FLB*09], but these methods have not yet proven practical for interactive graphics. In this paper, we introduce several innovations that enable interactive stochastic rasterization on existing commodity GPUs today and will likely become increasingly effective on their descendants.

Several properties make a rasterization-based approach attractive. Unlike ray tracing, the time complexity of rasterization scales only weakly with image resolution, for a fixed scene tessellation. Doubling the number of samples in an image typically increases the time to rasterize a scene less than twofold, since much of the visibility computation is amortized over more samples. We seek an approach that extends this broad advantage of rasterization — weak scaling with resolution — to the domain of distributed visibility effects (motion blur, depth of field). On a pragmatic level, we seek a stochastic rasterization approach that exploits the tremendous progress of commodity graphics hardware, remains compatible with existing shaders and game art workflows, and coud be directly incorporated into future evolutions of hardware and APIs. In particular our test scenes dominated by "macro-triangles," triangles whose area is at least a couple of pixels.

We follow traditional rasterization and proceed in draw call order, generating for every triangle a conservative screen-space "footprint" bounding the image of the triangle when blurred. This footprint is triangulated and rasterized, invoking a pixel shader for each pixel in which the triangle is potentially visible. The pixel shader uses a stochastic time and lens position to generate a ray that it then casts against the original triangle. If the ray intersects the triangle, texturing and shading proceed as normal using the resulting barycentric coordinates. If desired, the pixel shader generates and casts multiple rays per pixel, achieving a higher sampling rate for visibility than for shading. This dovetails well with hardware MSAA, which does the same thing for

† {momcguire, eenderton, pshirley, dluebke}@nvidia.com

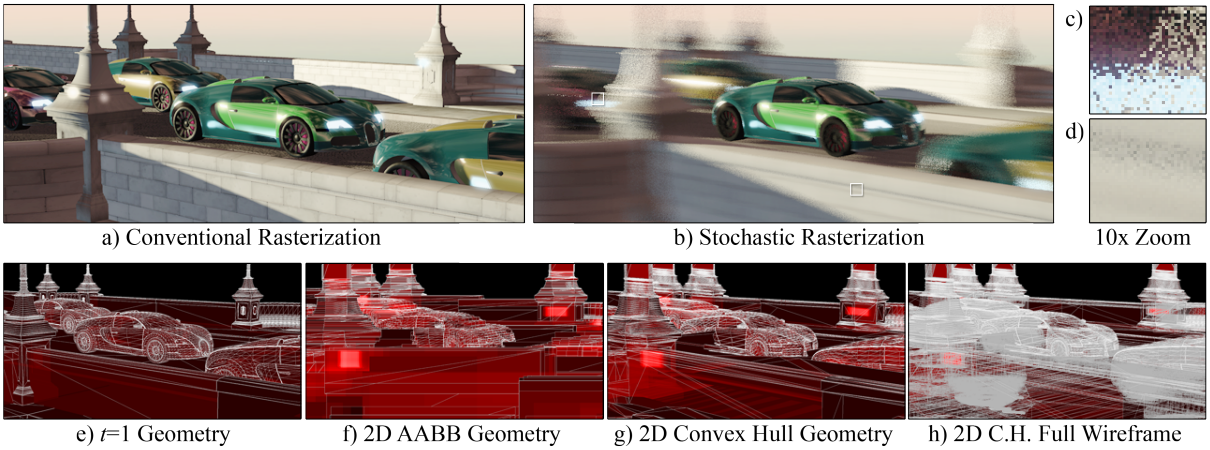

Figure 1: *A 1.8 million triangle scene at 1920×1080 resolution. (a) Conventional rasterization. (b) Motion blur rendered by stochastic rasterization at 9 frames per second with 1 shading, 1 visibility, and 4 texture samples per pixel on a GeForce GT 280 GPU. Details of sampling noise at (c) geometric and (d) texture edges. Colors in the second row indicate the number of fragments rasterized per pixel using (e) the shutter-close positions, (f) the screen-aligned bounding boxes, or (g) the convex hulls of the moving triangles. The convex hulls provide tighter bounds, resulting in much less overdraw. Those wireframes show only visible surfaces; (h) is the complete geometry streamed to the rasterizer. (Non-zoomed images in this paper have reduced resolution to print and display on screen accurately; see* `http://research.nvidia.com` *for the full source images.)*

conventional rasterization. The efficiency of the approach depends on the tightness of the screen-space bound. We use the projected convex hull of the blurred triangle as a good bound for motion, and bounding boxes as an acceptable although suboptimal bound for defocus. Those few triangles whose motion spans both the camera plane and the near clip plane require special handling; we propose a new method for finding a 2D bounding box that we conjecture conservatively covers the triangle's projection.

Our stochastic rasterization algorithm achieves interactive performance on existing hardware for complex scenes like the one shown in Figure 1. We anticipate such an approach will be feasible in games within a couple of GPU generations, and suggest some hardware improvements that could bring it about sooner. That prediction is based on three trends that we currently observe in hardware rendering. These are an increasing ratio of coverage to shading samples (e.g., via MSAA and CSAA); more complex BRDF, indirect illumination, and shadowing algorithms that increase the cost of shading a sample; and the persistence of power-efficient fixed-function units supporting rasterization. If these continue, the advantages of our algorithm over accumulation buffering will grow because it adapts a conventional rasterizer to perform many visibility tests across time and lens position while holding shading rate as low as once per pixel. Because it stochastically samples eye rays, it enjoys the benefits over post-processing and ad hoc methods that different effects (e.g., stochastic transparency, motion blur, defocus) work together without any special-purpose tricks, and that increasing processing power directly increases image quality.

2. Related Work

We classify related works by how they sample the 5D space of sub-pixel spatial location (x,y), time t, and lens position (u,v).

Conventional Render a single image in which (x,y,t,u,v) are fixed for all pixels. Standard rasterization functions as an instantaneous pinhole camera, resulting in a strobing "stop-motion" effect and lack of defocus - objects that should be blurry are sharp. Images rendered using conventional sampling can be post-processed using image processing blurs. These post-processing techniques are currently the method of choice for interactive systems, but are prone to artifacts ([AMHH08] p. 490).

Accumulation Render multiple conventional images and average their pixel values [HA90, WGER05]. This brute-force approach fixes the sub-pixel (x,y) offset and (t,u,v) values across each image. It is powerful but slow, and undersampling can result in discrete "ghosting" instead of continuous blur.

Stochastic Render a single image in which each sample location has an independently chosen (x,y,t,u,v) position [CPC84]. The primary artifact is noise — blurry objects appear dithered — which is typically countered by supersampling. The most well-known example of this technique is the Reyes algorithm in RenderMan [CCC87], used to make many offline-rendered movies. More recently researchers have investigated how to approach interactivity using stochastic rasterization [AMMH07, TL08, FLB*09, RKLC*10], but interactive rates remain elusive. Making that leap is the goal of this paper.

Our approach is inspired by a comment by Wexler et al. [WGER05] in which they considered and then rejected using bounding volumes plus ray intersections for off-line GPU rendering of motion and defocus blur. With more advanced hardware and some algorithmic improvements, this idea is now practical. The first steps for defocus were demonstrated by Toth and Linder [TL08], who also identified the thread coherence problem that currently limits performance for all stochastic methods of this form. The notion of bounding geometry plus per-fragment visibility tests also appears in many non-stochastic rendering applications, such as displacement mapping [WWT*03], GPGPU ray tracing [HSHH07], and ambient occlusion [McG10, LK10]. Akenine-Möller, Munkberg, and Hasselgren [AMMH07] suggested bounding motion by the convex hull and identified the problem with crossing the camera plane ($z = 0$). We extend and refine previous ideas into a solution for motion and defocus blur, addressing robust and efficient 2D convex hull in detail, as well as texture sampling, multisampling, conservative ray tests, and the $z = 0$ problem, for which we propose a novel 2D bounding box solution.

3. Algorithm

3.1. Overview

Each sample represents a point in 5D space. We denote the ordinates of the sample under consideration as t^*, x^*, y^* on the unit space-time interval within a pixel and u^*, v^* on the unit disk of a normalized lens. Sample time is more complicated than a point in screen or lens space because it changes the actual scene, not just the camera's view of it. We refer to four times: $t = 0$ and $t = 1$ are the shutter open and close times; t^* is the time at which visibility is computed for a given sample; and t_s is the global time at which shading and shadowing occur.

Each input triangle is defined by its vertex positions at times $t = 0$ and $t = 1$. The camera transform is also defined at these two times; thus the ends of the interval are convenient choices for t_s. We assume the vertices move linearly during the interval. The camera may also carry depth of field parameters (which for simplicity we assume to be static).

Each triangle is converted into geometry that conservatively covers the pixels that the triangle could affect, taking into account its motion and defocus. We then conventionally rasterize this bounding geometry.

For each rasterized fragment, the algorithm performs a ray-triangle intersection test to determine actual visibility. These rays are stochastic samples over time t and lens position u, v. Samples that pass the intersection test are shaded and set to their correct depth (which is necessary because the default fragment depth is that of the bounding geometry, not the triangle). Visibility between triangles is determined by a conventional depth-buffer test, which operates correctly because (t, u, v) are fixed at each screen-space sample location.

In more detail, here is an implementation sketch of the stochastic rasterization algorithm for a conventional programmable pipeline API such as DirectX or OpenGL.

1. **Shadows**: Compute shadow maps from the geometry at time t_s.
2. **Host Program**: Bind the vertex stream and transformation matrices for $t = 0$, plus any texture coordinates and other shading attributes, as usual. Bind the vertex stream for $t = 1$ as an additional vertex attribute, and transformation matrices for $t = 1$ as additional state.
3. **Vertex Shader**: Transform the vertices to homogeneous clip space at both $t = 0$ and $t = 1$.
4. **Geometry Shader**: Each triangle now provides six camera-space vertices, including three from the $t = 1$ attribute. Except for special cases discussed in Section 3.3, perform the homogeneous division on all vertices and emit bounding geometry of the resulting 2D projected points.
   For in-focus triangles, we use the convex hull of the projected vertices (Section 3.2). For out-of-focus triangles, we need to bound the set of six circles corresponding to dilation of each vertex by the defocus radius at its depth. In this case we use a simple axis-aligned bounding box. In either case, the z of the emitted geometry is set to the minimum z of the projected vertices, so that portions of the hull may be z-culled by the rasterizer. (This z is clamped to the near plane, to prevent clipping the hull.) Triangles crossing $z = 0$ are handled specially, as described in Section 3.3.
   The pixel shader requires as input the complete time-continuous triangle in camera coordinates, with all six vertices plus shading attributes. We attach this complete description to every emitted vertex.
5. **Pixel Shader**:
   a. Read parameters $(x^*, y^*, t^*, u^*, v^*)$ for the current sample from a precomputed, tiled 128^2 buffer.
   b. Interpolate the triangle's camera-space vertices at t^*.
   c. Solve for the barycentric weights of the ray-triangle intersection (Section 3.4). Set the fragment depth to the depth of the intersection. (DirectX "conservative odepth" and the OpenGL idiom "max(gl_FragCoord.z, depth)" should both allow early z testing despite writing to depth.) If there is no intersection, then discard the fragment.
   d. Shade. Compute the texture coordinates, normal, tangent, and other shading inputs by barycentric interpolation. Note that for shadow map tests, specular calculations, etc., the surface position should be interpolated at shading time t_s, not sample time t^*.
   Here, the fragment shader is performing the attribute interpolation conventionally performed by the rasterizer.

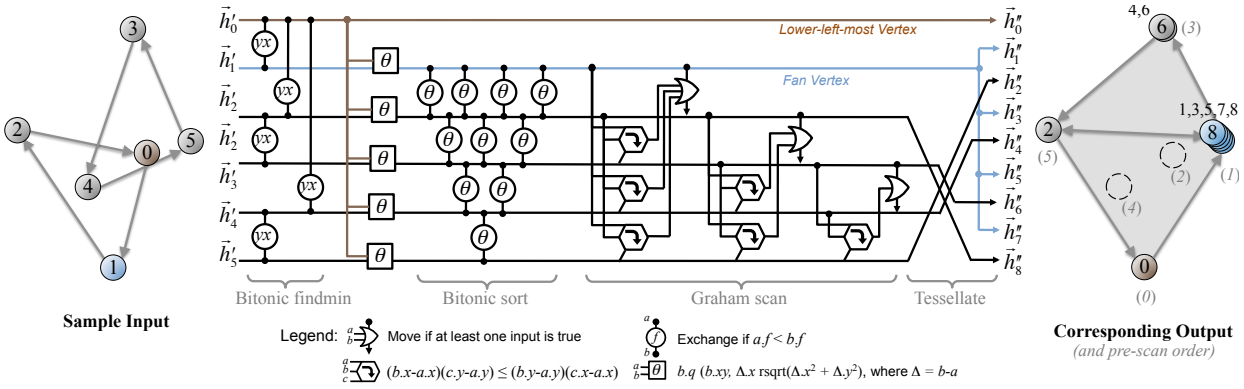

Figure 2: *Our bounding geometry algorithm. The six screen-space vertices of a time-continuous triangle enter on the left in any order. Vertices forming the tessellation of the 2D convex hull exit on the right in triangle-strip order.*

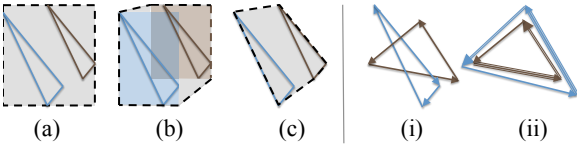

Figure 3: *Some choices for bounding geometry. We use the 2D convex hull (c) because the cost of computing this tight bound is repaid in the per-fragment threads that are conserved. Note that the triangles may have arbitrary orientation with respect to each other, such as (i) or (ii).*

We extend this base algorithm in two ways that introduce error but increase the perceived image quality by reducing variance. The first extension is temporal integration of texture samples using a single additional ray cast, as described in section 3.5. The second is increased visibility samples per shading computation via multi-sampled antialiasing (*MSAA*), as described in section 3.6.

The entire process can be thought of as a source code transformation on an existing non-stochastic shader chain. Our system doubles the vertex shader, inserts a fixed epilogue in the geometry shader, and inserts a fixed prologue into the pixel shader.

3.2. Convex Hull

For each triangle, we need to rasterize bounding geometry large enough to cover the triangle's entire motion [AMMH07]. The most conservative bound is the near plane of the entire view frustum. The tightest bound would be the front-facing surfaces of the triangle's swept volume, but the sides of this volume are curved in the general case (the start and end positions of an edge may be skew). A common choice in previous work is the 2D or 3D bounding box of the six vertices. For diagonal triangles or diagonal motion, the bounding box can cover $O(n^2)$ samples when only $O(n)$ are covered by the triangle. We instead use the 2D convex hull of the six vertices, which is equal to the 2D convex hull of the swept volume. (Proof: Since points in the swept volume are convex combinations of the six vertices, adding them does not change the convex hull.) See Figure 3.

Figure 2 shows our convex hull algorithm, expressed as a circuit diagram to demonstrate its suitability for inclusion in a hypothetical hardware stochastic rasterizer unit, as well as in a geometry shader where dynamic array indexing and recursion are inconvenient. This is the Graham scan algorithm [Gra72], but specialized to $N = 6$ with the stack and conditional branches eliminated in favor of conditional moves between vertex registers. These moves replace vertices that do not lie on the hull with vertices that do, in such a way that the resulting triangles are degenerate and therefore quickly eliminated by the GPU's rasterizer.

The hull algorithm takes vertices $\vec{h}'_0 \ldots \vec{h}'_5$ as input. These are the post-projective clip-space vertices of the swept volume, in any order. The `findmin` stage performs conditional exchanges to bring the vertex with the lowest y component to register 0. Ties are resolved in favor of the lowest x value. The θ field (which we store in the vector's z field in a geometry shader) increases with the counter-clockwise (CCW) angle of each vertex about this new vertex zero. The bitonic sort then arranges the six vertices in CCW winding order.

Following the sort, vertices 0, 1, and 5 must lie on the convex hull. The algorithm therefore only needs to classify the three remaining vertices. If the path $\vec{a} \rightarrow \vec{b} \rightarrow \vec{c}$ turns to the right for any three CCW-ordered (not necessarily consecutive) vertices, then $\vec{b}$ cannot be on the convex hull. The algorithm exhaustively tests vertices in registers 2, 3, and 4 against this property using the sub-circuit denoted by the hexagon with a right arrow.

Finally, the interleaving pattern on the far right converts the resulting CCW convex polygon into a triangle strip con-

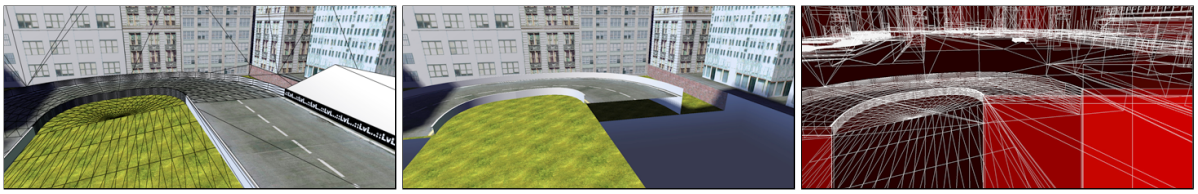

Figure 4: *Left: a roadway, modeled with exceptionally large triangles that frequently trigger the $z = 0$ crossing case. Middle: scene with triangles that cross $z = 0$ removed. Right: visualization of overdraw using our motion blur algorithm (with zero velocity). Maximum overdraw is 6.*

taining some degenerate triangles. (DirectX and OpenGL geometry shaders cannot emit triangle fans.)

3.3. Crossing $z = 0$

At the geometry stage, if all six vertices are culled by one view frustum plane, the triangle can be culled. We conjecture that if both the shutter-open and shutter-close triangles are backfacing, a triangle under linear motion is always backfacing on the entire interval, so we also cull in that case.

Source triangles entirely in front of the camera plane ($z = 0$) are processed as described in the previous section.

The remaining case is a set of vertices that spans both the $z = 0$ singularity and the near plane. Almost all triangles crossing $z = 0$ are outside the view frustum (which shrinks to zero width near the singularity); the few remaining typically result from a camera moving low and fast over a ground plane. Projecting this geometry to 2D is problematic. Straightforward 2D convex hull or bounding box algorithms would require the geometry to be clipped against the near plane before projection. But since the side "faces" of a time-continuous triangle are bilinear patches, precise clipping is complicated and produces curved objects. A full-screen quadrilateral is of course a conservative bound, but would make rendering impractically slow.

We attempt to improve that bound as follows. A time-continuous triangle has six vertices. There are 15 ways of connecting these into edges. We conjecture that those edges are a superset of the edges in the 3D convex hull (this is why our algorithm for the common case works). Therefore the hull that contains both the vertices in the positive half-space and any intersections of the 15 edges with the near plane is the hull of the part of the time-continuous triangle that lies in the positive half-space. It is hard to compute either the 3D or 2D projection of that hull in a geometry shader because doing so requires consideration of 15 potential intersections in addition to the original six vertices. However, it is easy to compute a 2D bounding box of the projections of those points.

For triangles that cross z=0, we compute this 2D bounding box, viewport-cull it in 2D, and then for the few boxes that pass the cull, rasterize the result with our usual fragment shader. This algorithm produces correct images with practical performance in the cases we have tried, including hard scenes like the one in figure 4. We conjecture, but have not proved, that it produces a conservative bound in all cases.

3.4. Conservative Ray Cast

We encode the statically chosen time t and camera-space ray direction and origin (from (x,y,u,v)) for each screen-space sample in two texture maps. The fragment shader reads these, interpolates the camera-space triangle vertices to the chosen time t and then solves for the barycentric weights of the ray-triangle intersection ([AMHH08] p. 750). We conservatively accept any intersection for which these weights are in the expanded range $[-\varepsilon, 1+\varepsilon]$, in order to accommodate different precision between the (typically fixed-point) rasterizer and the fragment shader computation. To make ε roughly proportional to half a pixel, we chose the computationally efficient approximation

$$\varepsilon = \min(\max(k_1 \cdot d, 0), k_2), \tag{1}$$

for Manhattan distance d between the ray origin and the triangle centroid. If ε is too large, then object contours that should be blurry will appear solid. We chose $k_1 = 0.002, k_2 = 0.1$ empirically for our test scenes, which are at 1920×1080 and have 45° to 70° fields of view. Note that setting the constants on the smaller side or skipping this step entirely may be acceptable for many applications–at $\varepsilon = 0$ we observed about 1 visibility error per 100,000 pixels.

3.5. Texture Integration

MIP-map sampling computes an average texture at texture coordinate $\vec{s}$ using an area defined by $\partial\vec{s}/\partial x$ and $\partial\vec{s}/\partial y$. Averaging the *inputs* to shading rather than its output creates artifacts when the shading function is nonlinear, which it almost always is. Yet this practice, like bilinear texture filtering, is widespread because error from nonlinearity is less perceptible than that from spatial texture aliasing.

To support selection of MIP levels, GPUs compute screen-space (x,y) derivatives of arbitrary expressions by finite differences between parallel subexpressions across 4-pixel *quads*. Under stochastic rasterization, such automatic differences may compare samples with different values of

(x,y,t,u,v). This causes surfaces that should not be blurred at all to correctly exhibit no texture blur; however, it can cause overblurring of surfaces that should already be somewhat blurry. This is primarily visible under motion blur, where details become incorrectly blurred perpendicular to the axis of motion. Whether this artifact is undesirable depends on the application; it may be considered a feature in some cases, but we chose to address it.

We investigated locking (t,u,v) parameters across quads to allow built-in derivatives to operate correctly, but found the resulting increase in variance to be undesirable. So we instead compute t,u,v derivatives explicitly. We cast a second ray per fragment at $t+\Delta t$ (ignoring whether it hits inside the triangle) and estimate $\partial\vec{s}/\partial t$ by finite differences, $|\partial\vec{s}/\partial(x,y)|$ by z-distance from the eye, and $|\partial\vec{s}/\partial(u,v)|$ by the radius of the circle of confusion [PH04, 490] scaled by $|\partial\vec{s}/\partial(x,y)|$. Applying a ray differential inside a rasterizer's shader is our only contribution here; after that we follow ideas by Loviscach [Lov05] and others. The GLSL call for the texture integral from gradients is:

$$\texttt{textureGrad}\left(T,\vec{s},\frac{\partial\vec{s}}{\partial t},\left(\left|\frac{\partial\vec{s}}{\partial(x,y)}\right|+\left|\frac{\partial\vec{s}}{\partial(u,v)}\right|\right)\vec{1}\right). \quad (2)$$

For platforms on which `textureGrad` is unavailable or slow, we substitute

$$\frac{1}{N}\sum_{i=0}^{N}\texttt{texture}\left(T,\vec{s}+k_3\frac{i-N}{N}\frac{\partial\vec{s}}{\partial t},\left|\frac{\partial\vec{s}}{\partial(x,y)}\right|+\left|\frac{\partial\vec{s}}{\partial(u,v)}\right|\right). \quad (3)$$

We found that $\Delta t = 0.01, N = 6, k_3 = 0.8$ gave good results on our test scenes. Constant $0 < k_3 \leq 1$ jitters the time interval to avoid discrete ghosts from the N point samples. We used the approximation in eqn. 3 for all of our results. Note that the algorithm only blurs texture fetches by the ray differentials. There is no post-processed motion or defocus blur.

3.6. Multi-Sample Antialiasing

The efficiency of multi-sample antialiasing (MSAA) rasterization extends to ray casting as well. We cast one ray per MSAA sample, keeping a bit mask to store hit/miss outcomes for each sample. If the bit mask is all zero, we discard the pixel. Otherwise we set the MSAA sample mask to the bit mask and shade the last intersection point. Using the last point allows us to keep a minimum of state.

Shading only once per pixel per fragment is a great advantage over accumulation methods. Pixels within the intersection of the shutter-open and shutter-close triangles will have all their samples covered by the triangle, at various times and barycentric positions. Just as a single shade suffices for static MSAA, and those texture look-ups are filtered over the triangle's intersection with the pixel, a single shade suffices for stochastic MSAA, particularly when the texture look-ups are filtered over the blurred triangle's intersection with the pixel.

The improvement in shading rate depends on the area of the blurred triangle relative to the area of the original triangle (i.e., the speed of the triangle in proportion to its size). For 8x MSAA, triangles that cover 8x their area will receive no advantage, as every sample they cover will tend to be in a different pixel. Micropolygons thus gain less advantage from MSAA than larger triangles, for a given speed.

		Motion			*Defocus*	
		Race	**Bridge**	**Fairy**	**Fairy**	**Cubes**
		130 ktri	1.8 Mtri	174 ktri	174 ktri	50 tri
Vis	Shade	Fig 6.	Fig. 1	Fig. 5ur	Fig. 5ll	Fig. 4
1	1	43.8	9.5	22.8	5.8	104.4
4	1	36.2	4.8	15.5	2.6	59.2
4	4	16.4	4.3	14.1	3.2	31.0

Table 1: *Frames per second for test scenes at 1920×1080 resolution. Rows vary the visibility samples per pixel and shading samples per pixel, corresponding to no AA, 4x MSAA, and 4x SSAA.*

4. Experimental Results

We implemented stochastic rasterization in OpenGL on an NVIDIA GTX 280 GPU system. As expected the algorithm computes motion blur from combined camera motion (Figure 1), object motion (Figure 8), and object deformation (Figure 6), with correct occlusion (Figure 5) and camera blur of shadows. Figure 6 demonstrates a strength of stochastic rasterization: motion and defocus blur can be computed correctly and simultaneously.

Table 1 demonstrates that the algorithm can achieve interactive performance for a variety of scenes. Race and Bridge are high-speed car scenes at two levels of complexity. Fairy is a standard benchmark scene, and Cubes is a simple scene of cubes receding through the focus field. The bounding boxes used for defocus are a weakness of our algorithm and substantially degrade performance on highly tessellated scenes. Cubes and Fairy shade approximately the same number of samples, yet Fairy has many thin triangles that generate millions of extra failed visibility tests and thus unused pixel processing units.

As Figure 1 illustrates, using the convex hull algorithm reduces overdraw by providing much tighter bounds than an axis-aligned bounding box. The exact performance impact is highly scene- and animation-dependent, but in practice we observe that using convex hull decreases frame time significantly; for example, motion blur on the Fairy Forest scene renders in 107 ms using convex hulls versus 310 ms using bounding boxes.

Table 2 shows the performance impact of disabling stages of the algorithm. This gives intuition for, but is not the same as, the cost of a specific stage. That is because GPUs are massively parallel and share units between stages. For example, an increase in geometry workload decreases the cores available for pixel shading. The pixel shader shading time is

amplified by the fact that a thread group must wait for all pixels to complete shading, even if only one visibility sample passed the ray test. Since a triangle's bounding geometry grows large under motion and defocus blur, this means that the total cycles devoted to shading can increase dramatically, even though the total number of shading samples in the final image remains constant. This can be seen by comparing the shading cost reported in table 2 for the four subimages in figure 6, which vary only in defocus and motion blur. This is a drawback of stochastic rasterization originally identified by Toth and Linder [TL08] and is a strong incentive to build stochastic sampling directly into the rasterizer unit in the future. The "Overhead" column measures the cost of the draw calls and vertex shader invocation when the vertex shader and later stages do no work. The "Bound" time is the relatively negligible cost of the geometry shader plus the larger cost of the rasterizer in the case where the pixel shader returns a constant color.

Scene	Bound	Tex	Vis	Fig	Overhead	Transform (VS)	Bound (GS+Rast)	Visibility (PS)	Shade (PS)	Frame
Bridge	2D Hull	4	1	1e	11 ms	+ 0	+ 1	+ 5	+ 19	= 36 ms
	2D Hull	4	1	1f	11 ms	+ 0	+ 1	+ 17	+ 75	= 104 ms
	AABB	4	1	1g	14 ms	+ 0	+ 1	+ 41	+ 118	= 171 ms
Cubes	AABB	6	4	4l	1 ms	+ 0	+ 0	+ 15	+ 3	= 19 ms
Race	2D Hull	4	1	6	2 ms	+ 1	+ 0	+ 8	+ 22	= 33 ms
Fairy	2D Hull	1	1		3 ms	+ 0	+ 1	+ 9	+ 20	= 32 ms
	2D Hull	6	4	5ul	3 ms	+ 0	+ 1	+ 16	+ 30	= 53 ms
	2D Hull	6	4	5ur	3 ms	+ 0	+ 4	+ 55	+ 98	= 160 ms
	AABB	6	4	5ll	3 ms	+ 0	+ 7	+ 347	+ 154	= 513 ms
	AABB	6	4	5lr	3 ms	+ 0	+19	+ 546	+ 127	= 695 ms

Table 2: *The performance impact of each stage, measured by progressively disabling subsequent pipeline stages. At 1920×1080, varying texture and visibility samples but always shading 1x per pixel.*

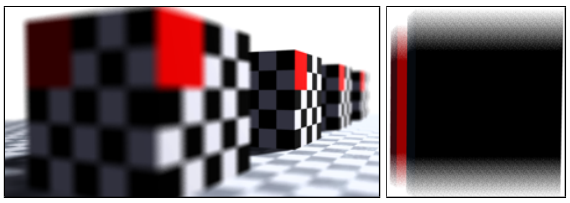

Figure 5: *Stochastic rasterization images showing (a) correct defocus blur and (b) correct motion blur (two rising boxes). Post-process techniques are challenged by surfaces revealed by rotation, disocclusion, or defocus, because color and velocity information are missing.*

Image quality is affected by several factors. Figure 8 shows how image quality improves with increased sampling. Note that stochastic noise, like film grain, is less visible

Figure 6: *(upper left) The Fairy Forest scene, with (upper right) motion only, (lower left) defocus only, and (lower right) simultaneous motion and defocus at 4x MSAA.*

when animated, so these figures look noisier than the associated animations. Figure 7 shows that texture blur can have a bigger effect even than MSAA samples.

Figure 9 compares accumulation buffering to stochastic rasterization for fixed rendering budgets in a scene with significant camera and object motion. As render time increases both converge towards a smooth result. We primarily increased the number of time samples for accumulation buffering and the number of total samples for stochastic rasterization. Accumulation buffering performs best with simple shaders, since it shades once per time per pixel. Where there is no overdraw stochastic rasterization only needs to shade once per pixel. However, stochastic rasterization requires more texture samples at each pixel and presents an incoherent workload to the GPU. We expect the advantages of stochastic rasterization to be amplified in the future by current GPU programming trends: more complex shaders, higher MSAA or CSAA sampling rates, and better work scheduling and compaction.

Figure 10 shows the results of our $z = 0$ crossing case algorithm for an adversarial scene. Empirically we have not yet found a case where the algorithm fails, but as future work we would like to prove the conjecture that it is always conservative.

5. Summary and Discussion

We have introduced a stochastic rasterization method built on conventional rasterization hardware and APIs, compatible with the existing ecosystem of real-time shaders, art assets, and game engines. We use a hybrid renderer in which

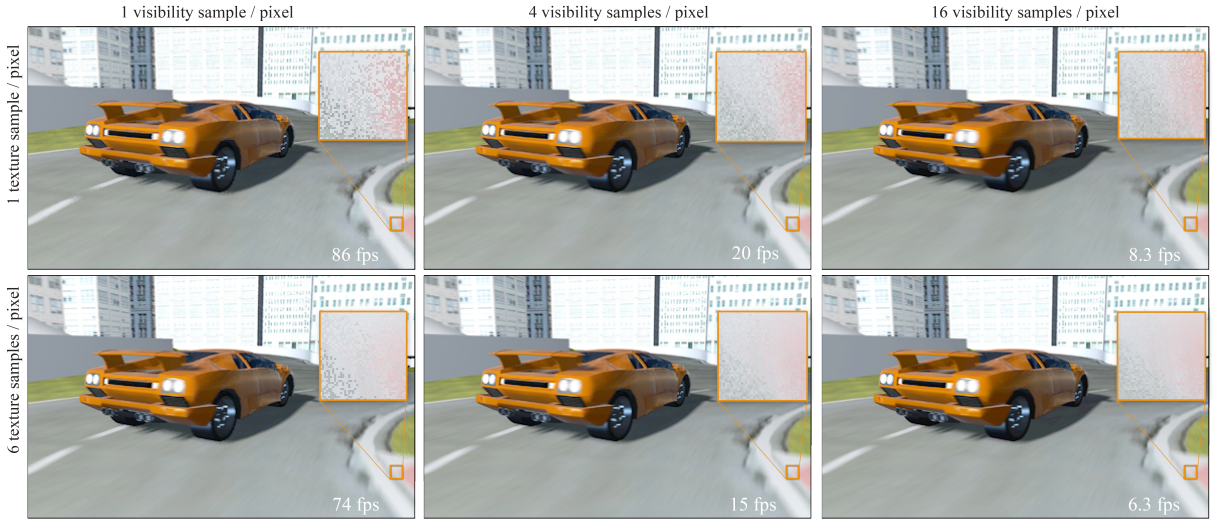

Figure 7: *130ktri racing scene with camera and object motion, rendered with motion blur. Along rows: Increasing texture filter size by the amount of motion or defocus provides improves sampling at texture edges, such between the red and white stripes in the 10x zoom insets. Along columns: Increasing visibility samples improves noise everywhere, including the geometric edge against the gray road on the lower-left of the insets, but at higher cost.*

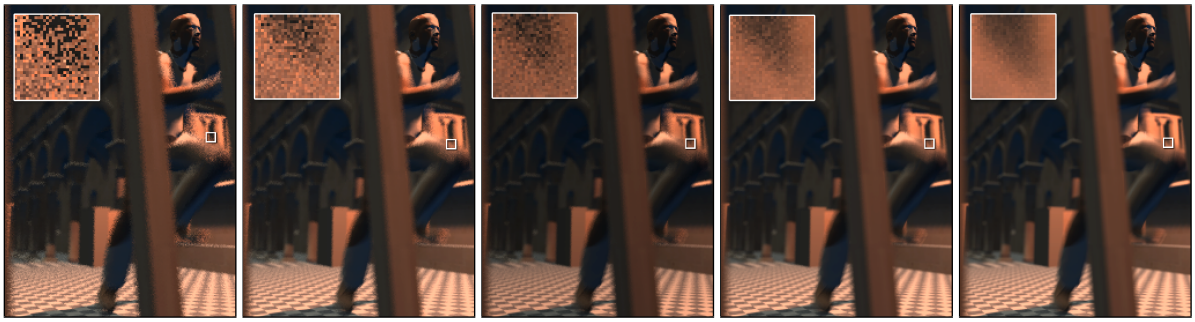

Figure 8: *The "Ben" animated character in the Sibenik Cathedral, with simultaneous motion and defocus blur, with 10x zoom of the outlined areas as insets. Variance due to stochastic sampling decreases with visibility samples per pixel. We find motion noise to be less apparent when viewed on screen under animation, and consider 4x to 16x visibility sampling to be reasonable for games. For still and film rendering, many more samples are required – only around 256x sampling is noise reduced to the level one would tolerate anyway from (real or CGI) film grain or camera sensor noise.*

2D rasterization conservatively identifies pixels potentially covered by a time- and lens-continuous triangle, and 5D ray-casting determines exact visibility and barycentric coordinates for shading. This simple idea, combined with several algorithmic extensions such as temporal texture filtering and careful interaction with MSAA, enables us to render complex game-like scenes with depth of field and motion blur, achieving acceptable quality at high resolution and interactive frame rates. We have suggested features for future hardware that would improve these frame rates further.

We observe that motion and defocus blur differ in two important practical respects. First, noise from motion blur is generally less noticeable than noise from defocus blur because it only occurs on moving objects. Second, motion blur spreads the object footprint in 1D while defocus blur spreads it in 2D. These differences will likely affect the cost-benefit decisions of developers implementing stochastic rasterization in games and other real-time applications.

5.1. Limitations and Assumptions

We make several simplifying assumptions, most of them typical for rendering systems with motion blur. Each assumption implies associated limitations.

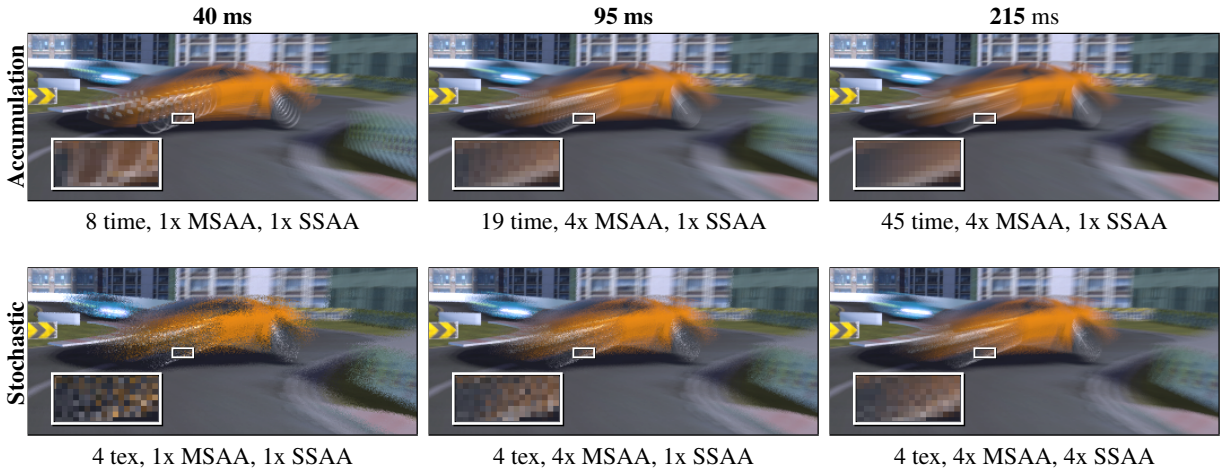

Figure 9: *Comparison of accumulation buffering and stochastic rasterization for the same render times. Accumulation buffering renders a number of discrete ghosts equal to the number of time samples; stochastic rasterization creates noise. Insets are zoomed 5x. Images are cropped and downsized for printing from the full 1920×1080 results; see our web page for the original data.*

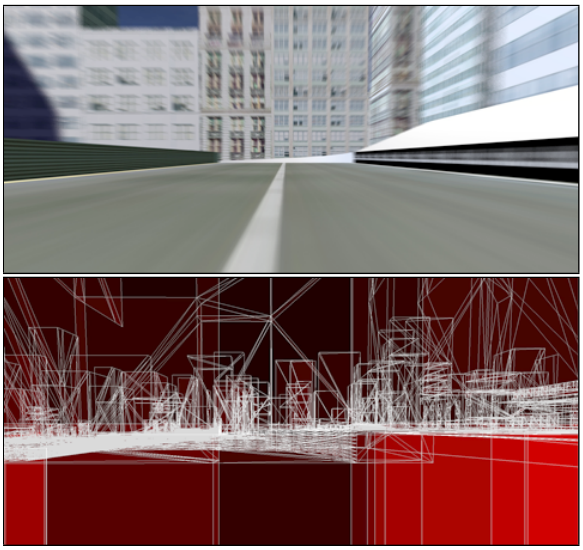

Figure 10: *Top: our stress test for our $z = 0$ algorithm includes very large triangles under large relative motion, which frequently triggers the case. Bottom: visualization of overdraw that shows moving triangles are worse, but not substantially worse, than static triangles. The camera for this image is centimeters above the poorly-tessellated area shown on the lower right in Figure 4.*

Constant topology. We assume that every triangle exists throughout the exposure interval, and that the position of every vertex is known at shutter open and shutter close. Our implementation generates $t = 0$ and $t = 1$ positions by skinning and transforming each vertex twice every frame and passing two camera transformations to the vertex shader. One could avoid the double transformation in a system where vertex velocities are available from a physics or animation system. Either choice requires augmenting a non-motion-blurred renderer with one extra attribute per vertex.

Constant shading. Following previous work [CCC87, FLB*09, RKLC*10], we shade at a single time per frame (shutter close in our implementation). This can produce smeared highlight artifacts but is a generally accepted practice often used in films. Surprisingly, shadows of moving objects computed once per frame are often satisfactory [RSC87], perhaps because percentage-closer-filtered shadows are already blurry. For increased shadow accuracy, one alternative is to compute time-varying shadow maps [AMMH07].

Moderate defocus. Under extreme defocus, a tiny triangle has the potential to contribute to any pixel on the screen. In that case our algorithm would be too slow for real-time. When the largest defocus point spread radius is on the order of three pixels, our algorithm can still produce reasonable performance. Our implementation currently uses a screen-space axis-aligned box to bound the Minkowski sum of the triangle with its circle of confusion. This is correct but overly conservative, further limiting our performance. Note that the circle of confusion for a single point on the near plane quickly encompasses the whole screen as the near plane approaches zero; a large near plane depth is thus essential for efficient defocus blur under any stochastic rasterization algorithm. We use 0.5 meters.

Shutter time. Varying the shutter duration increases and

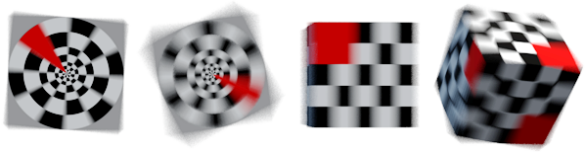

Figure 11: *The linear motion approximation gives plausible results even for modest rotation.*

decreases the size and overlap of swept triangles, which directly impacts performance. This provides a nice knob for developers to trade off performance for motion blur. Curiously, a shutter open for the entire frame time produces motion blur that appears too strong to a human viewer. Our images and videos use a half-frame exposure interval, a typical value for film rendering.

Linear motion. We assume that all vertex motion is linear during an exposure. Figure 11 shows that this gives surprisingly good results even for moderate rotational motion.

Interaction with alternate rendering strategies. Our algorithm is directly compatible with traditional forward rendering; super-sampling; bloom, gamma-correction, and tone map post-processing; stochastic transparency; and deferred shading (in which case one renders stochastic G-buffers instead of radiance). Screen space methods that rely on the depth buffer, such as screen-space ambient occlusion, would have to be altered to consider only adjacent framebuffer values that correspond to the same (t,u,v) sample. Given the small number of samples those methods tend to take, we suspect that such an alteration is impractical in many cases.

5.2. Future work

We plan to explore several enhancements including generalization to other distribution ray tracing effects, such as reflectance function sampling and soft shadows; improvements to bounding geometry for defocus blur; and methods for objects with highly non-linear motion during the shutter interval. We would also like to use interactive stochastic rasterization to study the effects of sample density, resolution, etc. on interactive tasks, and the practical ranges of motion and defocus blur parameters for typical applications.

References

[AMHH08] AKENINE-MÖLLER T., HAINES E., HOFFMAN N.: *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008.

[AMMH07] AKENINE-MÖLLER T., MUNKBERG J., HASSELGREN J.: Stochastic rasterization using time-continuous triangles. In *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware* (Aire-la-Ville, Switzerland, Switzerland, 2007), Eurographics Association, pp. 7–16.

[CCC87] COOK R. L., CARPENTER L., CATMULL E.: The Reyes image rendering architecture. *SIGGRAPH Comput. Graph. 21*, 4 (1987), 95–102.

[CPC84] COOK R. L., PORTER T., CARPENTER L.: Distributed ray tracing. *SIGGRAPH Comput. Graph. 18*, 3 (1984), 137–145.

[FLB*09] FATAHALIAN K., LUONG E., BOULOS S., AKELEY K., MARK W. R., HANRAHAN P.: Data-parallel rasterization of micropolygons with defocus and motion blur. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), ACM, pp. 59–68.

[Gra72] GRAHAM R.: An efficient algorithm for determining the convex hull of a finite planar set. *Information processing letters 1* (1972), 132–133.

[HA90] HAEBERLI P., AKELEY K.: The accumulation buffer: Hardware support for high-quality rendering. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques* (1990), ACM, p. 318.

[HSHH07] HORN D. R., SUGERMAN J., HOUSTON M., HANRAHAN P.: Interactive k-d tree gpu raytracing. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2007), ACM, pp. 167–174.

[LK10] LAINE S., KARRAS T.: Two methods for fast ray-casted ambient occlusion. In *EGSR 2010* (June 2010).

[Lov05] LOVISCACH J.: Motion blur for textures by means of anisotropic filtering. In *Proceedings of the Eurographics Symposium on Rendering* (2005), pp. 7–14.

[McG10] MCGUIRE M.: Hardware-accelerated ambient occlusion volumes. In *Proceedings of High Performance Graphics 2010* (Saarbrucken, Germany, 2010), ACM SIGGRAPH and Eurographics Association.

[PH04] PHARR M., HUMPHREYS G.: *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[RKLC*10] RAGAN-KELLEY J., LEHTINEN J., CHEN J., DOGGETT M., DURAND F.: Decoupled sampling for real-time graphics pipelines, 3 2010. MIT Computer Science and Artificial Intelligence Laboratory Technical Report Series MIT-CSAIL-TR-2010-015.

[RSC87] REEVES W. T., SALESIN D. H., COOK R. L.: Rendering antialiased shadows with depth maps. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1987), ACM, pp. 283–291.

[TL08] TOTH R., LINDER E.: *Stochastic Depth of Field using Hardware Accelerated Rasterization*. Master's thesis, Lund University, Sweden, June 2008.

[WGER05] WEXLER D., GRITZ L., ENDERTON E., RICE J.: Gpu-accelerated high-quality hidden surface removal. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (New York, NY, USA, 2005), ACM, pp. 7–14.

[WWT*03] WANG L., WANG X., TONG X., LIN S., HU S., GUO B., SHUM H.-Y.: View-dependent displacement mapping. *ACM Trans. Graph. 22*, 3 (2003), 334–339.