High Performance Graphics (2012)
C. Dachsbacher, J. Munkberg, and J. Pantaleoni (Editors)

High-Quality Parallel Depth-of-Field Using Line Samples

Stanley Tzeng[†1], Anjul Patney[1], Andrew Davidson[1], Mohamed S. Ebeida[2], Scott A. Mitchell[2], and John D. Owens[1]

[1]University of California, Davis
[2]Sandia National Laboratories

Abstract

We present a parallel method for rendering high-quality depth-of-field effects using continuous-domain line samples, and demonstrate its high performance on commodity GPUs. Our method runs at interactive rates and has very low noise. Our exploration of the problem carefully considers implementation alternatives, and transforms an originally unbounded storage requirement to a small fixed requirement using heuristics to maintain quality. We also propose a novel blur-dependent level-of-detail scheme that helps accelerate rendering without undesirable artifacts. Our method consistently runs 4 to 5$\times$ faster than an equivalent point sampler with better image quality. Our method draws parallels to related work in rendering multi-fragment effects.

1. Introduction

Emulating the way our eyes perceive the world adds a significant amount of realism to 3D graphics. Recent research has demonstrated a great interest in methods that simulate blurry effects like motion-blur and depth-of-field. In this paper, we present a method for the latter. Traditionally, depth-of-field effects are computed using stochastic point samples along the lens in 4D (x,y,u,v) space, where (x,y) are coordinates in pixel space and (u,v) are coordinates along the camera lens. However, using point samples in this 4D space can lead to noise artifacts. Reducing the noise to visually acceptable levels requires hundreds of samples per pixel, making the operation quite expensive even on a powerful device like a modern GPU. We would prefer a method that required fewer samples to achieve an acceptable noise level and that could easily be parallelized.

For high-quality rendering at low cost on parallel hardware, we propose using continuous *line samples* instead of point samples. A line sample gathers all the information from a line in (x,y,u,v) space rather than a single point. Each of these line samples provide more information on the lens than a point sample. While each line sample is more expensive than a point sample, we will show that they converge faster than point sampling (Figure 1) with a lower amount of noise and can thus produce acceptable visual quality in a more efficient way. While line samples can add bias to the process, careful design minimizes the impact on the final image. Like point samples, each line sample can be evaluated in parallel, and this makes line samples a viable candidate for GPU rendering.

Though sampling is highly parallel, implementing a line-sample-based depth-of-field rasterizer suitable for the GPU is a significant challenge. The main difficulty stems from the memory constraints of the GPU. Sampling in 4D generates a massive amount of samples that, if not managed well, can lead to a significant performance hit due to memory bandwidth bottlenecks. Our heuristic-based prioritization scheme keeps only the most relevant samples that contribute to the final color of a pixel. We further optimize our rasterizer for the GPU by introducing a level-of-detail mesh resolution scheme to accelerate rendering. Careful kernel design, along with our heuristics, allow us to keep most major memory accesses on chip. We will demonstrate that just a few line samples can give results with less noise than hundreds of point samples while running four times faster. The result is a tiled rasterizer that can achieve depth-of-field effects with a significantly lower number of samples than point sampling. This speedup allows our rasterizer to run at interactive frame rates.

The main contributions of this paper are:

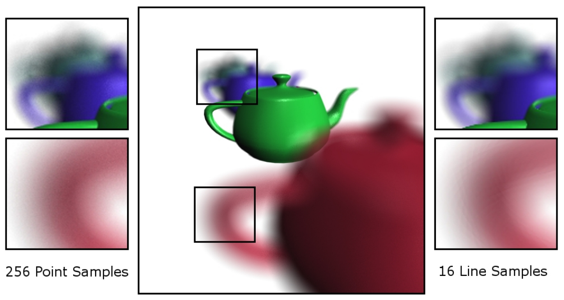
- A novel depth-of-field scheme using line samples that converges faster than traditional point sampling and lends well to an efficient data-parallel implementation.
- An efficiently parallelized line sampling algorithm on the

[†] stzeng@ucdavis.edu

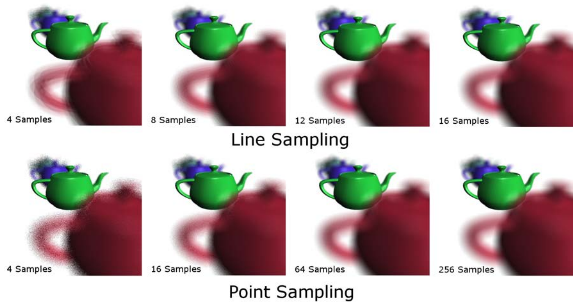
© The Eurographics Association 2012.

DOI: 10.2312/EGGH/HPG12/023-031

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



(a) Comparison between point and line sampling. Middle is ground truth done with 1600 point samples per pixel



(b) Quality improvement with increasing samples

Figure 1: *A quality comparison between depth-of-field effects rendered using conventional point sampling and those rendered using continuous line samples. Point sampling converges slowly, and is unable to generate noise-free images even at 256 samples per pixel. In contrast, using line samples provides acceptable quality with only 16 samples.*

GPU that fits within local memory limitations via heuristics that help keep only the most important information.
- A new level-of-detail (LoD) scheme based on circles of confusion. Our LoD scheme accelerates our depth-of-field method during high blur.

2. Previous Work

Line Sampling Jones and Perry [JP00] experimented with line samples for anti-aliased polygon rendering. They shoot line samples across a pixel's spatial extent, analytically compute triangle coverage for each of them, and average them to obtain pixel colors. Unlike their method, our line samples lie in (u,v) space, do not span multiple pixels, and are much more parallel-friendly.

Recent research has also shown promise in rendering high-quality motion-blur using multi-dimensional samples. Gribel et al. [GDAM10] present the use of line samples in the (x,y,t) domain to analytically render motion blur effects in a scene and in their more recent work, proposed the use of planar samples in the (x,y,t) domain for motion-blur [GBAM11]. They also extend their implementation to render motion-correct ambient occlusion.

Depth of Field Recent work in depth-of-field rendering has explored efficiency from both sampling and reconstruction perspectives. Akenine-Möller et al. [AMTMH12] accelerate depth-of-field effects by culling tiles based on their positions in the lens domain. As opposed to tiles, Munkberg and Akenine-Möller [MAM11] accelerate motion blur and depth-of-field by culling backface geometry, resulting in a 5D culling test. On the other hand, Lehtinen at al. [LAC*11] optimize the reconstruction to require much fewer samples for high-quality results.

On the GPU, much sampling work has concentrated on motion blur with extended implementations to handle depth-of-field effects. Akenine-Möller et al. [AMMH07] presented multi-pass stochastic rasterization that uses point samples to render depth-of-field effects on the GPU. McGuire et al. [MESL10] construct a rasterizer on the GPU that can handle depth-of-field; Lee et al. [LES10] use depth peeling to accelerate their GPU ray tracer for lens effects.

The fundamental difference between previous work and ours is that we explore the use of line samples for depth-of-field within the constraints of a highly parallel environment but with limited local resources. Previous work was CPU-based without aggressive memory constraints, which eases the difficult task of storing line samples that require potentially unbounded space. Our problem also spans an extra dimension than Gribel et al. and thus requires careful treatment to ensure tractable memory usage. Our solution approximates the unbounded set of sample-triangle intersections using a static amount of memory per pixel, which we accomplish via several heuristics to ensure that we store only the most important intersections. This idea is analogous to the technique of Salvi et al. [SML11] in the domain of adaptive order-independent transparency. We also propose a novel blur-dependent LoD method that recovers some lost performance for scenes with high blur.

3. Line Sampling

Line samples are linear-domain counterparts of point samples which, due to their increased dimensionality, are able to capture more information about the domain per sample. For the same reason, one line sample is computationally more expensive than a point sample. This trade-off makes our exploration exciting, and we wish to find out whether the improvement in quality can account for the additional performance cost. In other words, we would like to use as few

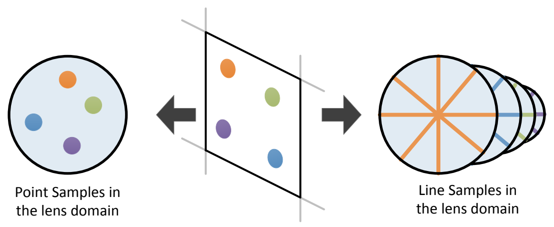

Point Samples in the lens domain
Line Samples in the lens domain

Figure 2: *Comparison of line samples against point samples. For each of the spatial pixel samples (middle), stochastic point sampling (left) considers a single point in the lens domain. In contrast, our method (right) instantiates several line samples (four in the picture) in the lens domain for each spatial sample. Each spatial pixel sample samples the screen using eight lines. Colors indicate which line samples belong to which spatial sample. For simplicity we organize line samples in a wagon-wheel fashion.*

line samples as possible to obtain high-quality results with an efficient implemention on modern parallel hardware. Figure 1 shows a quality comparison, which demonstrates that just 16 line samples can be sufficient to minimize any major artifacts.

3.1. Formulation of Our Method

Mathematically, computing the color of a pixel in the presence of depth-of-field can be expressed as a four-dimensional integral over the pixel's spatial (x,y) and lens's aperture (u,v) dimensions (see Figure 2). We construct several line samples in the (x,y,u,v) domain (as shown in Figure 3), and analytically compute and store triangle coverages along the line samples. Once we have processed all triangles, we resolve the list of coverages for each line sample to obtain the final sample as well as pixel color. This high-level description is analogous to Gribel et al. [GDAM10], though we extend it to a broader four-dimensional problem and also present a novel parallelization technique.

The following subsections describe the steps of our formulation. The input to our system is a list of triangles.

3.1.1. Triangle Setup

Figure 2 provides a brief overview of our design for line sampling the four-dimensional (x,y,u,v) domain. In order to span this domain, we could construct line samples in several ways. We can support arbitrary lines, but that is a computationally expensive alternative. Hence, we focus on choosing line samples that are not only efficient to process, but also lend to simple parallelization and straightforward parameterization across the lens. With these goals in mind, we have found that a simple separation of spatial and lens dimensions with lines in the latter works best. For each pixel sample (x,y), we consider several line samples in the lens (u,v) domain. Intuitively, our strategy uses lines to aggregate the light entering a given screen location from across the lens surface.

Our choice is driven by the fact that sampling in this fashion has three advantages: (1) it is computationally cheap; (2) due to symmetry, it has an efficient data-parallel implementation; and (3) it has minimal undesirable artifacts. Furthermore, our sampling process can be cleanly split into a setup phase and a coverage evaluation phase, which helps in reducing redundant computation across multiple line samples. To demonstrate this, we now derive the set of edge equations for rasterization in the presence of depth-of-field. While the derivation is more complex than for traditional rasterization, the resulting equations are similar to classic triangle setup.

For a given triangle, we start by computing a signed radius of circle-of-confusion (CoC) for each vertex, obtained using the following expression [Ham07]:

$$\text{CoC} = A\frac{f(z-z_f)}{z(z_f-f)},$$

where A and f are the camera aperture and focal length, respectively, and z_f and z indicate the respective depths of the focal plane and the given vertex. Note that z is simply the w coordinate of the vertex in clip space.

We assume a linear apparent motion of the vertex screen coordinates on varying u and v. For a given screen-space vertex $x_i, y_i : i \in \{0,1,2\}$,

$$\begin{aligned} x_i^u &= x_i + c_i u, \\ y_i^v &= y_i + c_i v, \end{aligned} \tag{1}$$

where c_i is the vertex's CoC and $u, v \in [-0.5, 0.5]$.

Assuming the ith oriented edge connects vertices i and j of a triangle, we substitute Eq. (1) into the equations for testing whether a point (x,y) lies in the half-space created by edge i. Let ES_i represent the edge-sum at point (x,y) for the ith edge:

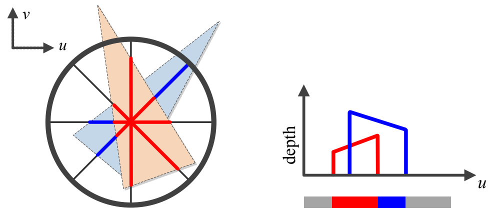
$$ES_i(x,y,u,v) = (y-y_i^v)(x_j^u - x_i^u) - (x-x_i^u)(y_j^v - y_i^v) \geq 0.$$

Expanding, we get the equivalent equations

$$\begin{aligned} \Leftrightarrow \quad & (y-y_i-c_iv)\left(x_j-x_i+u(c_j-c_i)\right) \\ - \quad & (x-x_i-c_iu)\left(y_j-y_i+v(c_j-c_i)\right) \geq 0 \\ \Leftrightarrow \quad & ES_i(x,y,0,0) \\ + \quad & u\left(c_j(y-y_i)-c_i(y-y_j)\right) \\ - \quad & v\left(c_j(x-x_i)-c_i(x-x_j)\right) \geq 0. \end{aligned}$$

This simplifies to

$$ES_i(x,y,u,v) = C_i(x,y) + uA_i(y) - vB_i(x) \geq 0, \tag{2}$$



(a) Line samples in the (u,v) domain. (b) Resolving depth for a sample.

Figure 3: *Our technique for computing analytical coverage using line samples. The (u,v) domain with transformed triangles and line samples is shown in (a). Bright segments indicate coverage. Depth resolution per sample is shown in (b). Without line samples, we would have to use many more point samples within the sampling region to achieve equivalent quality.*

where

$$\begin{aligned} A_i(y) &= c_j(y-y_i) - c_i(y-y_j) \\ B_i(x) &= c_j(x-x_i) - c_i(x-x_j) \\ C_i(x,y) &= (y-y_i)(x_j-x_i) - (x-x_i)(y_j-y_i). \end{aligned}$$

Here C_i is simply the traditional edge-sum. A_i and B_i can both be partially computed during triangle setup, and updated using MAD operations for every line sample. Note that A_i here is distinct from the camera aperture A.

During setup, we transform edges of each screen-space triangle into lens space, then generate line samples in this domain, computing coverage as described in the next subsection.

3.1.2. Coverage Computation

For each pixel sample, we instantiate several line samples along the u or v directions. We consider a circular (u,v) domain (corresponding to a circular lens shape) and line samples thrown in a wagon-wheel fashion (see Figure 3).

We chose a wagon-wheel sampling pattern for three major reasons. First, a wagon-wheel pattern in a circular domain ensures that all line samples have equal contribution to the final image. Should we have chosen, say, a grid pattern instead, then not all lines would have the same length. This would cause unnecessary asymmetry and would hinder a data-parallel implementation. Second, wagon-wheel line equations can be expressed in simple slope-intercept form $v = mu$, which greatly simplifies the math needed for coverage computation. Third, while there is a bias associated with the wagon-wheel sampling pattern, reweighting during reconstruction is rather trivial. A wagon-wheel pattern gives us excellent sampling quality while maintaining simplicity.

Rendering consists of testing each incoming triangle against potentially covered line samples. For each conservatively covered pixel sample, we use equations from Section 3.1.1 to transform triangle edges to the (u,v) domain. Transformed edge equations in this domain behave similar to (x,y) edge equations in traditional rasterization, each dividing the (u,v) domain into positive and negative half-spaces. Our goal is to track the intersection of the positive half-spaces with each of our line samples. Given the transformed equations, we compute the coverage against a wagon-wheel line sample as follows:

1. Compute the point of intersection of the edge (using the equation from Section 3.1.1) with the line sample while tracking the part of dart which falls in the positive half-space. The result is an interval along the line sample, ranging from one extreme of the lens domain to the point of intersection.
2. Compute the intersection of intervals from each of the three edges per triangle to obtain an interval lying entirely inside the (u,v) domain. This interval represents the part of a line sample covered by the input triangle, illustrated by the bright red portions in Figure 3.

We call the above intersection interval as *segment*. Once segments for all triangles have been aggregated, we resolve the final color for each sample. Using an approach similar to Gribel et al.'s [GDAM10], we sweep across a sorted list of segments in the sample while aggregating segments closest in depth, and then use all pixel samples to compute the final color for the pixel.

Shading Although each segment covers a continuous parametric range across the triangle, storage as well as performance considerations make it impractical to compute a varying color across this range. Furthermore, we expect highly varying colors along a segment to correspond to primitives with high blur. As a result, we choose to shade each segment at its mid-point. In our implementation we shade at mesh vertices and interpolate to the mid-point of each segment, although per-segment shading is a straightforward extension.

In addition to being practical, the idea of shading once per sample is also inspired from decoupled sampling in modern graphics pipelines, where the goal is to shade each primitive at most once per pixel. In future work we wish to explore a reuse cache that will aid in sharing shading values across different line samples in a pixel.

We notice that such a decoupled shading scheme introduces some over-smoothing in blurry regions (see Figure 4), but we feel that this is an acceptable trade-off.

Bias Note that the wagon-wheel pattern will tend to bias reconstruction towards the center of the lens. Fortunately, we can eliminate this bias by non-linearly rescaling u and v such that the density of line samples (line samples per unit area) is constant across the lens surface direction. Because the

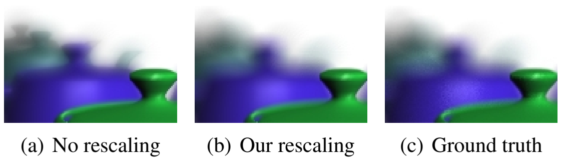


Figure 4: *Fixing bias due to our wagon-wheel sampling pattern. (a) shows undesirably sharp rendering due to bias towards the center of the lens. (b) shows the resulting image after using our non-linear weights to scale intervals. (b) is closer to our ground truth image (c), which was generated using 1600 stratified point samples per pixel.*

density of wagon-wheel samples reduces radially as $|1/r|$, we transform any point (r,θ) to $(2r|r|,\theta)$ before considering its contribution (2 is the normalizing factor). In practice, we compute the contribution κ of an interval (p_1,p_2) along any line sample $(p_1,p_2 \in [-0.5,0.5])$, instead of its length (p_2-p_1), as the following:

$$\kappa(p_1,p_2) = 2\times(p_2|p_2|-p_1|p_1|).$$

Figure 4 shows the impact of this modification.

4. Parallel Line Sampling

Since each pixel is independent, it is straightforward and natural to parallelize line samples along a per-pixel or per-sample basis. To correctly resolve depth, we must divide our rendering stage into a *sample* phase and a *composite* phase. This separation introduces the possibility of having a potentially very large number of triangle-sample interactions or segments per sample. In spirit, this is analogous to the problem faced in implementing other multi-fragment rendering effects like order-independent transparency [SML11].

Consider a straightforward implementation of the algorithm. We create separate GPU kernels for sample and composite. The sample kernel would test n line samples per pixel and write out the intersected segments into off-chip memory. The composite kernel would read in these segments then filter them into a color per pixel. However, implementing our rendering stage in this manner is quite inefficient for these reasons:

Producer-Consumer Locality Once a pixel has generated all necessary segments, we can move onto the composite stage. The above naive approach would not take advantage of the implicit producer-consumer locality in this sequence—this also avoids unnecessary copies to off-chip memory, since the resident segment values can be used directly in the composite stage.

Unbounded Segment Storage Storing all possible segments can lead to unbounded storage requirements. This storage is often wasted in segments that will make no meaningful contribution to the image. Modern GPUs have a fixed and limited amount of local storage, which further exacerbates the problem.

Excessive Work on Blurry Triangles Detail is less noticeable in triangles that are blurred. For high resolution meshes with large blur, we perform more work than needed in producing a faithful image. There should be a quality adjustment to the mesh to account for the loss of performance.

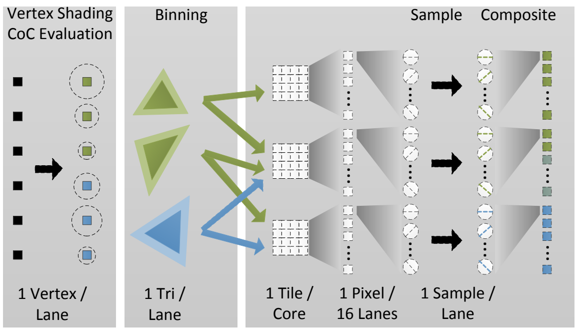


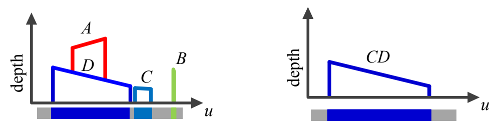
Figure 5: *Organization of our parallel renderer. Our GPU execution is divided into three phases, working in parallel on vertices, triangles, and line samples respectively. Separate gray squares indicate separate kernels. In the binning phase (middle kernel) the lighter color boundary around the triangles represent the triangle's CoC. Text at the top describes the action of the kernel while the bottom text has the granularity of the kernel. Here a lane is defined to be a SIMD lane. See section 4 for details.*

Based on the above observations, we have instead designed a tiled GPU renderer with fixed on-chip segment buffers and a blur-dependent level-of-detail scheme.

Main Kernel If we are to take advantage of producer-consumer locality, then the first step towards a better implementation is to merge the sampling and composite kernels together. We can merge the two kernels into one since their parallel granularity is now the same as both kernels process one sample per SIMD lane (see Figure 5).

Now, it is possible to store the generated segments from the sample stage and feed them to the composite stage without passing through global memory, saving useful bandwidth. We accomplish this by utilizing shared memory as a *segment queue* for a set of pixels.

The main drawback to using shared memory is that there is little of it. Thus, we tile the screen so that segments generated in a tile fit into shared memory. To avoid each tile rendering every primitive in the scene, we employ a preprocess stage that bins the primitives for each tile [LK11]. We are now able to describe the complete algorithm in detail.



(a) Incoming Segments. (b) Result.

Figure 6: *Example of the segment heuristics. Left: Segments A, B, C are incoming segments and D is a segment in the segment buffer. Segments A and B are rejected by our heuristic. A is rejected since it is occluded completely by D. B is rejected because it is too small to make any visible contribution. Right: C is similar enough in depth, color, and close enough in distance to D that it is merged together with D to produce a new segment CD. If C had a completely different color than D, then it would have been kept separate from D and C would be added to the segment buffer separately. Our heuristics aggressively winnow incoming segments to help minimize shared memory usage.*

Algorithm Our final algorithm consists of two preprocessing kernels and a main kernel. Figure 5 presents an overview.

Our first kernel transforms vertices into screen space, shades them, and then calculates their CoCs. A second kernel assembles triangles from these vertices and intersects them with tiles. A triangle that intersects a tile is placed in that tile's primitive bin and passed to the main kernel.

The main kernel then generates line samples per pixel. Each line sample tests the primitives in its bin, and generates line segments with the samples that intersect primitives. Each line sample has a segment buffer in local (shared) memory which it uses to store line segments. Once all the primitives have been sampled and their respective segments generated, the composite phase of the main kernel begins. During this phase, each sample walks along its parametric dimension and iteratively yet analytically aggregates the contribution of the closest segment on every step as illustrated in Figure 3.

We have described our algorithm on the GPU, but as it stands we have not handled what happens when a line sample generates too many segments for its segment buffer. In these cases we wish to evict segments with little or no contribution, and keep segments that may make the largest contributions to the final color of the pixel.

4.1. Segment Heuristics

We define several strict heuristics that determine whether or not an incoming segment is qualified to occupy the limited segment buffer.

However, before we consider choosing the more valuable segments, we compress each segment to 16 bits of storage to save space. Then we perform three tests:

Occlusion Given our current segment buffer. If we are to add a new segment, we check if this segment is already covered by any other segment. If so we disregard it. Alternatively, if the incoming segment completely covers an existing segment, we replace the latter's contents with the incoming segment (evicting the previous segment).

Merge If an incoming segment is sufficiently close to an existing segment within our segment buffer, we would like to try to merge the two in order to save space. Gribel et al. [GDAM10] utilized a similar technique to save space in their analytic motion blur renderer.
In order to merge two segments, we require three conditions must be met:
(1) The Euclidean distance between the two colors is less than a certain threshold.
(2) The difference in depth between these two segments is below a certain threshold.
(3) The orientation and slope of both segments are similar enough to merge.

Length If the incoming segment is too short to have a significant contribution to the final color, it is discarded.

Figure 6 shows an example of the heuristics on incoming segments. If a segment passes all tests then it is placed in the segment buffer. Should the buffer be full, then we evict one segment from the buffer. Eviction is based on the contribution of the segment based on its length. The segment with the minimum length is evicted to make room for the incoming segment. If the incoming segment has the smallest contribution of all segments, then it is discarded.

Our current set of heuristics does not account for intersecting segments. Our implementation simply ignores this corner case. We have found that implementing a sensible heuristic for this corner case is inefficient and inconsequential. Further, in our renderings even at very high depth complexities we have observed no visual artifacts from this.

4.2. Level of Detail

While designing our renderer, we noticed that scenes with a high degree of depth-of-field blur tended to put an enormous pressure on the number of triangles in each rendering bin. This is due to triangles over-intersecting with tiles due to a large CoC of the triangle.

This hurt our performance and also created problems for managing tile storage. To alleviate these issues plaguing triangles with a high degree of blur, we added multi-resolution meshes in a level of detail (LoD) scheme. However, unlike a traditional rasterizer that determines an object's LoD by its screen size, we determine it by using its blurriness or a measure of minimum CoC. The intuition behind our scheme is that meshes whose vertices have a high CoC are going to be severely blurred, and so a coarser mesh will suffice.

LoD Determination To determine the LoD for a scene object, we maintain two parameters: minimum CoC (c_{min}) of the object bounding box, and the screen-space length of an average edge (d_{avg}). We compute c_{min} using the equations presented in Section 3.

To efficiently compute d_{avg}, we take the center of the object bounding box p and create two points p_1 and p_2 such that p_1p_2 is parallel to the camera plane and is in object space the same length as the average base object edge. Then, d_{avg} is simply the distance between p_1 and p_2 after transformation to screen-space.

We compute the LoD score for an object as the ratio $s_{lod} = (c_{min}/d_{avg})$ and this is our metric for selecting the appropriate LoD mesh. s_{lod} gives us an idea of the blurriness of the average triangle in the object as compared to its average edge length. A large s_{lod} indicates that the object is either too blurry, too small, or both. In all these cases, we are better off selecting a lower resolution mesh instead, saving precious cycles and tile storage. On the other hand, low s_{lod} indicates an object that is either sharply focused, too big, or both. In these cases, we should try to select a higher resolution mesh to maintain visual fidelity.

Our simple LoD scheme allows us to render tiles more efficiently and at the same time reduces the loss in performance due to excessive blur. Furthermore, we are confident that it will robustly extend to tessellated primitives, where we can adjust the tessellation factors based on s_{lod}. We plan to experiment with this idea in the near future.

5. Results

To test our formulation, we designed two GPU-based renderers. One is our line sampler implementation as described in Section 4, and the second is a stochastic sampler based on point samples. Both renderers are set up as tiled renderers, only differing in their sampling and compositing procedures, thus allowing us to directly compare point- and line-sampled approaches. We wish to investigate the number of samples that both renderers need to return a high-quality image and the time it takes for them to render such a scene. We ran our experiments on an NVIDIA GTX 580 running CUDA 4.1, with two main tests: line-sampling vs. point-sampling rasterizers in terms of speed and quality, and the LoD vs. the non-LoD line-sampling rasterizer.

Figure 9 shows the resulting image quality and their respective performance is given in Table 1. All of our images are rendered at 800×800 resolution and they are embedded into the electronic version of this paper. Interested readers are encouraged to zoom in and observe the noise differences between the line and point samplers.

Our line sample renderer is consistently 4–5 times faster

Mesh	Point (fps)	Line (fps)
Dragon	0.64	2.86
Cessna	2.12	9.01
Hand	1.12	4.47
Geocheck	11.54	49.31
SDCC	0.49	2.04
Teapot	1.93	5.11

Table 1: *Performance of our point sampler against our line sampler. Numbers are given in frames per second. The point sampler uses 256 point samples per pixel while the line sampler uses 16 lines per pixel. Images of these test scenes are shown in Figure 9. Line sampling is consistently 4–5 times faster than point sampling.*

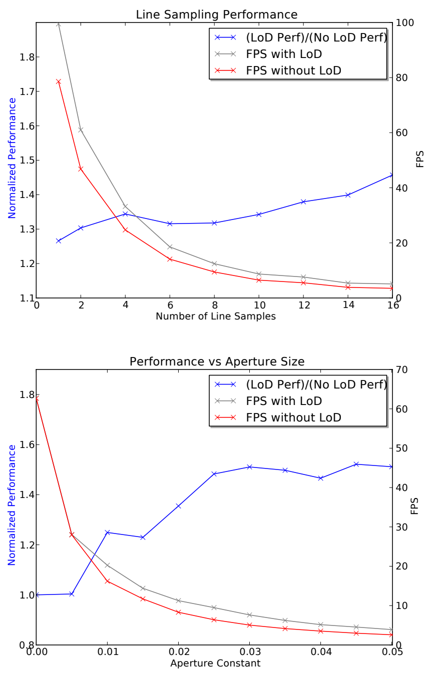


Figure 7: *Performance behavior of our renderer. Top: Normalized performance and FPS with varying number of line samples. Bottom: Normalized performance and FPS with varying aperture size for 16 line samples per pixel.*

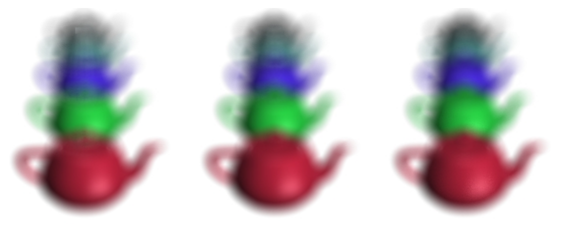


Figure 8: *We demonstrate the impact our blur-dependent LoD scheme by showing 5 teapots rendered under high blur. Due to the level of blur we automatically select lower resolution meshes for the teapots. While excessive blur makes visual impact negligible (in fact, without LoD we often experience more artifacts due to tile overflow), our scheme increased the performance for this scene by* 2×. *We used 1600 point samples per pixel for ground truth.*

than its point-sampling counterpart. Although each line sample is more expensive than a single point sample, line samples use dramatically fewer samples to converge faster. Even at 256 point samples per pixel, the noise is reduced but still evident and computing such a large amount of samples require a significant amount of time.

Figure 8 shows our rendering of the same mesh with varying LoDs. Figure 7 shows two plots on how our LoD scheme affects rendering. We vary the number of line samples used per pixel, and also adjust the aperture of the camera. The scene used is the teapot scene shown in the far right of Figure 9. We used three levels of detail and they had 100%, 75%, and 50% of the original mesh vertices, respectively.

As the number of samples varies, the LoD renderer performs only marginally faster. This is mainly due to rendering fewer primitives. However, when the aperture is high (i.e. when there is heavy blur), then the LoD scheme is about 50% faster than without. Higher aperture implies larger CoCs for each primitive, and thus each tile has more primitives to process. As our LoD score is based on a combination of the mesh's CoC and the distance from camera, it is able to select the appropriate meshes based on blur. The result is a significant speedup in overall rendering time of our system with a minimal effect on quality.

Our implementation of LoD uses only three levels of resolution as a proof-of-concept. Though our results from these tests are promising, we are sure that there is room for more aggressive LoD schemes to improve performance. Further, our LoD scheme can translate naturally to other adaptive mesh schemes (such as tessellation). Our LoD scoring system can also be used to determine the tessellation level for a tessellation-based depth-of-field pipeline.

Limitations The main limitation of our technique is the severely limited amount of available shared memory. Given the heuristics that bound our line segment buffers, the quality and number of samples is currently bound by the amount of shared memory available. With a larger shared memory pool, we could implement larger tiles for better performance, or larger buffers for more complex scenes. This would allow fewer bins with a smaller number of overlapping triangles and thus fewer accesses from global memory overall.

Shading each segment at mid-point is another limitation of our renderer since it is vulnerable to losing detail in extreme cases (e.g. high-frequency textures). We believe it is an important piece of future work to further investigate complex shading effects for line samples.

Finally, we feel the need for further investigation of strobing artifacts in some of our images. While we know the cause of these to be the use of the same (x,y) position across pixels, our experiments at randomizing this position adversely affect the results by adding noise.

Artifacts Our GPU renderer performs aggressive approximations to achieve high performance. This may sometimes involve discarding segments that have a nontrivial contribution to the final image. Our heuristics help minimize such cases by carefully discarding segments with least expected contribution to the final color, and thus we are able to maintain reasonable quality against a point sampling based method.

Further, since our tile bins have a fixed size, having too many triangles in a tile can cause bin thrashing. Thus, too much geometry in a tile will result in potential artifacts in our renderer. An example of artifacts in our rendering system is shown in Figure 8. In the leftmost figure, tiles overflow and result in errors due missing triangles.

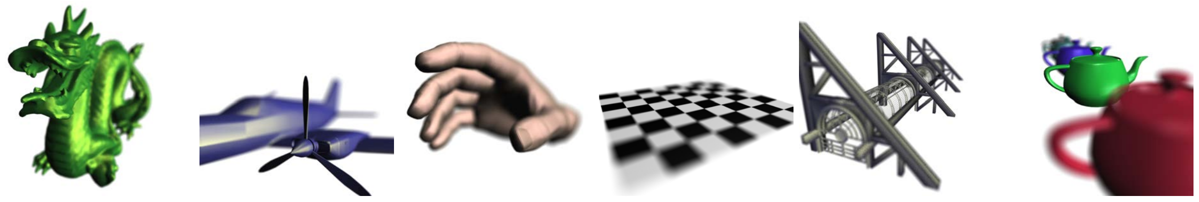
Fortunately, most such artifacts are associated with blur and tend to be less noisy than a point-sampled renderer. We have designed our heuristics and LoD scheme to help mitigate most rendering errors, but in some complicated scenes they may still be noticeable.

6. Conclusion

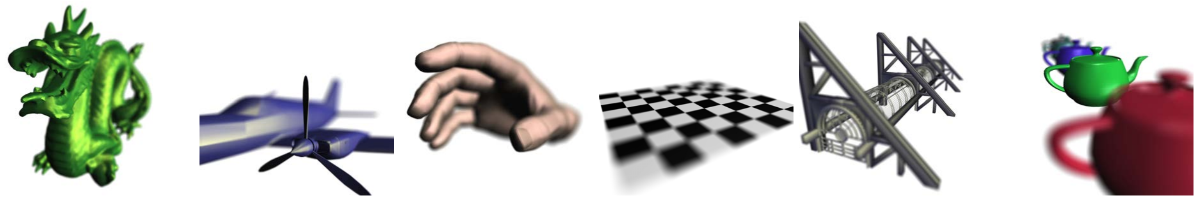
High-quality depth-of-field rendering is a continuing challenge, especially under the constraints of interactive performance. Since it requires a prohibitively large number of samples to be noise-free, point sampling is simply not practical in such a scenario.

In this paper we proposed an alternative in the form of line samples. We have shown that we can greatly reduce sampling noise with fewer line samples than point samples, and achieve interactive performance on a modern GPU despite severely limited local storage.

There are many avenues of future work from our algo-



(a) Example 800 × 800 scenes rendered using 256 point samples.



(b) Example 800 × 800 scenes rendered using 16 line samples.

Figure 9: *Examples of both renderers. Top row is point samples and bottom row is line samples. From left to right, the scenes are Dragon, Cessna, Hand, Geocheck, San Diego Convention Center (SDCC) and Teapot.*

rithm. Extensions to and combinations with other dimensions in rendering, for instance soft shadows and global illumination, should serve as intriguing explorations. We also plan to extend our LoD scheme to dynamically-tessellated primitives.

We hope that our work sparks readers' interest in the area of analytical line sampling and that design choices in our work will provide key guidelines for continuous-domain sampling research on parallel hardware. We also hope that our blur-dependent-LoD scheme will prove useful in other avenues of real-time rendering.

Acknowledgments

The authors appreciate the support of the Intel Science and Technology Center for Visual Computing, an Intel Ph.D. Fellowship, and National Science Foundation Award # CCF-1017399.

References

[AMMH07] AKENINE-MÖLLER T., MUNKBERG J., HASSELGREN J.: Stochastic rasterization using time-continuous triangles. In *Graphics Hardware 2007* (Aug. 2007), pp. 7–16. 2

[AMTMH12] AKENINE-MÖLLER T., TOTH R., MUNKBERG J., HASSELGREN J.: Efficient depth of field rasterization using a tile test based on half-space culling. *Computer Graphics Forum 31*, 1 (June 2012), 3–18. 2

[GBAM11] GRIBEL C. J., BARRINGER R., AKENINE-MÖLLER T.: High-quality spatio-temporal rendering using semi-analytical visibility. *ACM Transactions on Graphics 30* (Aug. 2011), 54:1–54:11. 2

[GDAM10] GRIBEL C. J., DOGGETT M., AKENINE-MÖLLER T.: Analytical motion blur rasterization with compression. In *High Performance Graphics* (June 2010), pp. 163–172. 2, 3, 4, 6

[Ham07] HAMMON JR. E.: Practical post-process depth of field. In *GPU Gems 3*, Nguyen H., (Ed.). Addison-Wesley, Aug. 2007, ch. 28, pp. 583–605. 3

[JP00] JONES T. R., PERRY R. N.: Antialiasing with line samples. In *Proceedings of the Eurographics Workshop on Rendering Techniques* (June 2000), pp. 197–206. 2

[LAC*11] LEHTINEN J., AILA T., CHEN J., LAINE S., DURAND F.: Temporal light field reconstruction for rendering distribution effects. *ACM Transactions on Graphics 30*, 4 (Aug. 2011). 2

[LES10] LEE S., EISEMANN E., SEIDEL H.-P.: Real-time lens blur effects and focus control. *ACM Transactions on Graphics 29*, 4 (July 2010), 65:1–65:7. 2

[LK11] LAINE S., KARRAS T.: High-performance software rasterization on GPUs. In *High Performance Graphics* (Aug. 2011), pp. 79–88. doi:10.1145/2018323.2018337. 5

[MAM11] MUNKBERG J., AKENINE-MÖLLER T.: Backface culling for motion blur and depth of field. *Journal of Graphics, GPU, and Game Tools 15* (Sept. 2011), 123–139. 2

[MESL10] MCGUIRE M., ENDERTON E., SHIRLEY P., LUEBKE D.: Real-time stochastic rasterization on conventional GPU architectures. In *High Performance Graphics* (June 2010). doi:10.2312/EGGH/HPG10/173-182. 2

[SML11] SALVI M., MONTGOMERY J., LEFOHN A.: Adaptive transparency. In *High Performance Graphics* (Aug. 2011), pp. 119–126. doi:10.1145/2018323.2018342. 2, 5