Volume xx (200y), Number z, pp. 1–14

Efficient Depth of Field Rasterization using a Tile Test based on Half-Space Culling

Tomas Akenine-Möller[1,2] , Robert Toth[1] , Jacob Munkberg[1], and Jon Hasselgren[1]
[1]Intel Corporation [2]Lund University

Abstract

For depth of field rasterization, it is often desired to have an efficient tile versus triangle test, which can conservatively compute which samples on the lens that need to execute the sample-in-triangle test. We present a novel test for this, which is optimal in the sense that the region on the lens cannot be further reduced. Our test is based on removing half-space regions of the (u,v)-space on the lens, from where the triangle definitely cannot be seen through a tile of pixels. We find the intersection of all such regions exactly, and the resulting region can be used to reduce the number of sample-in-triangle tests that need to be performed. Our main contribution is that the theory we develop provides a limit for how efficient a practical tile versus defocused triangle test ever can become. To verify our work, we also develop a conceptual implementation for depth of field rasterization based on our new theory. We show that the number of arithmetic operations involved in the rasterization process can be reduced. More importantly, with a tile test, multi-sampling anti-aliasing can be used which may reduce shader executions and the related memory bandwidth usage substantially. In general, this can be translated to a performance increase and/or power savings.

1. Introduction

During the last few years, research activity has increased in the stochastic rasterization field [CCC87, AMMH07, TL08, FLB*09, MESL10, BFH10, MCH*11]. The goal is to use more realistic camera models, so that correct motion blur and depth of field (DOF) is generated as a result of the visibility computations. This is in contrast to the majority of current real-time rendering engines, which use pinhole cameras with infinitely fast shutters. In some cases, these engines use ad-hoc approaches for a subset of these effects under controlled forms. An example is to add motion blur only from camera movement with approximate visibility.

Efficient rasterization of correct depth of field and motion blur at the same time remains an elusive goal. We note that rasterization of depth of field alone is a substantially simpler problem, but still, specialized algorithms for this is not a well-explored field in computer graphics. In this paper, we therefore present a novel approach for efficient depth of field rasterization. Our target is current and near-future real-time rendering applications, and these will continue to contain triangles of varying sizes. As a consequence, our shading is not executed per-vertex (as done in micropolygon rendering), but rather on a per-pixel level. In addition, our traversal order enables multi-sampled anti-aliasing (MSAA).

We use the notion of separating planes, which has been used in cell-based occlusion culling [CT97], and inspiration from soft shadow rendering research [NN85], and apply it to the topic of rasterization. Using several simplifications and new insights, we are able to present the theory for an optimal tile versus defocused triangle test. In addition, we develop a practical depth of field rasterization algorithm based on our new theory. It reduces the total number of arithmetic operations needed for rasterization, and it is hierarchical in that it visits one tile of pixels at a time. In general, this traversal order gives many advantages for rasterization algorithms, and in our case, we note that it may be used to reduce the number of shader executions and in addition, it is likely that such a traversal order reduces texture bandwidth usage [HG97] and depth buffer bandwidth [MCH*11] compared to other algorithms. We believe our research is a step towards a hardware implementation of depth of field rasterization.

2. Previous Work

In our research, we are primarily interested in rasterization-based methods that render accurate depth of field. In the following, we therefore avoid discussing stochastic ray tracing based methods [CPC84] and post-processing techniques [BHK*03, Dem04].

To efficiently handle triangles of varying sizes, most triangle rasterization algorithms are hierarchical in nature. Such

algorithms use triangle versus tile testing, and for static triangles, this is particularly simple [MM00, AMA05]. However, they are not straightforward to extend to motion blur nor depth of field rasterization.

Cook et al. [CCC87] presented the famous REYES rendering architecture, which contains a stochastic rasterizer for *micropolygons* capable of both motion blur and depth of field (DOF). They use a technique called *interval*, which is described in the previous work section by Fatahalian et al. [FLB*09]. The idea for DOF is simply to split the domain of the lens into a number of smaller regions. For each region, the triangle is bounded, and rasterized only to the samples inside the region.

Fatahalian et al. [FLB*09] adapted *interleaved* rasterization [KH01], to micropolygon rendering with blur effects. Brunhaver et al. [BFH10] later investigated the efficiency of different hardware implementations of this technique. However, micropolygon rasterization approaches are not directly applicable to *macro-sized* triangles. Due to shading at the vertex level, all primitives must be tessellated down to the pixel level. This increases the visibility computations tremendously, especially with the high sampling rates required for defocus blur. In this paper, we address these shortcomings by deriving a tile vs. defocused triangle overlap test that bounds the visible area of the lens. This test enables a tile-based traversal order with multi-sampled anti-aliasing (MSAA).

Recently, we developed a tile versus moving triangle test [MCH*11] for hierarchical rendering of motion blurred triangles. The tile test computed a time interval for each tile, and only the samples inside that interval needed to be inside-tested. Advantages of this algorithm included the possibility to use MSAA, reduced depth buffer bandwidth usage, and reduced number of arithmetic operations for the rasterization procedure.

Akenine-Möller et al. [AMMH07] rendered depth of field with line samples on the lens, which required several passes. In essence, motion blur rasterization was used to compute an image with depth of field. Recently, several algorithms have been presented for stochastic rasterization on GPU hardware. For depth of field, Toth and Linder presented the first stochastic rasterization algorithm running on GPUs [TL08]. Recently, McGuire et al. presented an implementation of motion blur and depth of field for current GPUs [MESL10]. A specialized method for computing the convex hull of the triangle and a technique for robust handling of the case where the triangle intersects with the $w = 0$ plane were presented. Inside testing was done using a ray-triangle intersection test, and MSAA was used to reduce shading costs. Lee et al. [LES10] present a method where layers are constructed using depth peeling. In a later pass, an image with DOF is created by ray tracing through the layers on the GPU.

In parallel with our work in this paper, Laine et al. [LAKL11] presented a new rasterization technique that can handle both motion blur and depth of field at the same time, which we do not handle. However, we use two types of planes (both axis-aligned and non-axis-aligned), while they only use the axis-aligned planes for culling. In Section 6, we briefly compare our DOF rasterization technique to Laine et al.'s DOF method without motion blur.

The inspiration for our algorithm comes from both occlusion culling and soft shadow rendering. Coorg and Teller [CT97] used separating and supporting planes for cell-based visibility queries for large occluders, and we use their notion of separating planes in our research. In addition, our work is also related to soft shadow rendering. With modest geometry, Nishita et al. showed that it is conceptually simple to split a soft shadow geometrically into umbra and penumbra regions [NN85].

3. Motivation

In this section, we motivate the need for a tile-based traversal order with a tile versus defocused triangle overlap test for depth of field rasterization. First of all, we note that with tile-based traversal, for a particular triangle, each pixel is visited at most once. The resulting traversal algorithm, with a defocused tile test available, has several potential advantages, such as:

1. efficient rasterization of mixed triangle sizes,
2. the possibility to use multi-sampling anti-aliasing,
3. better texture cache usage, and
4. reduced depth buffer bandwidth.

Each of these advantages are discussed below.

1. *Mixed Triangle Sizes* The ability to handle mixed triangle sizes, and not only micropolygons, is important, since much content today and in the near future will contain large, medium, and small triangles. Both the *interval* and *interleaved* rasterization techniques (Section 2) are primarily for micropolygon scenes. In such contexts, inside tests are performed for all samples within an axis-aligned bounding box around each primitive. However, this becomes inefficient (compared to a standard hierarchical rasterizer) when the triangle sizes increase, as illustrated in Figure 1. It is hard to efficiently extend the *interval* approach with a hierarchical overlap test. One approach is to compute and test against a convex hull for each lens region, but that is rather expensive. In contrast, interleaved rasterization can be augmented with a hierarchical test for the triangle positioned for a lens coordinate, which is a discrete position, (u_i, v_i), on the lens. However, the number of different lens coordinates required for acceptable depth-of-field quality is large (typically 64–256, see Figure 13), which leads to many hierarchical tests. Also, for interleaved rasterization, the number of samples a hierarchical test can cull is low due to that the screen space sample positions for each lens coordinate are sparsely distributed (e.g., one per 2×2 pixels for 64 samples per pixel, whose lens coordinates are chosen from a set of 256). With

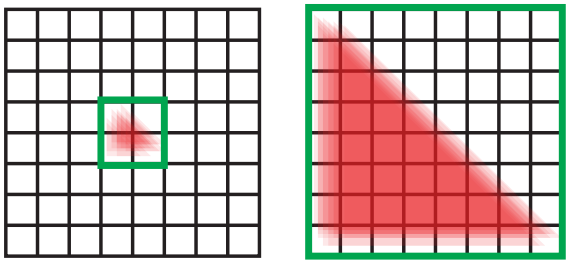


Figure 1: *Left: a pixel-sized triangle is positioned for a set of sample positions over the lens. All samples within the green bounding box will be tested. Right: a larger triangle is positioned for a set of sample positions over the lens. For many pixels in the upper right region, there is no overlap with the defocused triangle, and many unnecessary visibility tests are executed. A hierarchical rasterizer could easily discard many of these pixels.*

a tile test against a defocused triangle, on the other hand, an efficient hierarchical traversal order becomes possible.

2. *MSAA* To support mixed triangle sizes, we shade after visibility at the pixel level, similar to current hardware rasterizers. In contrast, micropolygon pipelines shade at the vertex level and thus need to tessellate all scene geometry to pixel-sized primitives. McGuire et al. [MESL10] have shown that multi-sampling anti-aliasing (MSAA) can be used for both motion blur and depth of field with substantially reduced number of shader executions. Since interleaved rasterization iterates over lens samples, the same screen space region will be visited multiple times, which means that MSAA is not trivially supported. With interval rasterization, one can shade once per lens region. This means that the shading rate is dependent on the number of regions the lens is split into (we use 4×4 regions in our research in order to get good traversal efficiency). In contrast, with a tile test against the defocused triangle, one can choose from the minimal number (one) of shader executions up to one shader execution per sample. For motion blur rasterization, a tile test can avoid as many as $75-90\%$ [MCH*11] of the pixel shader executions, compared to interleaved rasterization with super sampled shading. Similar or better results are expected for depth of field since more samples are needed to reduce noise in highly defocused areas. Note that image quality may suffer when using MSAA. However, in cases with low-frequency shading (diffuse, ambient occlusion, etc), one shader execution per pixel is often sufficient. We simply note that with a tile test readily available, MSAA can be used, and image quality can be traded for rendering speed for high-frequency shaders. It is important to note that the savings in shader executions often are directly translated to performance and/or power savings.

3. *Texture Cache Usage* For normal rasterization, it has been shown that a screen space tile-based traversal order is key to getting good texture cache efficiency [HG97]. In contrast, the traversal order of interleaved rasterization [KH01] may thrash the texture cache for reasonably large triangles. With shading at the vertex level, this is not a problem, but for shading at the pixel level, this may negatively affect the texture cache performance. For interleaved depth of field rasterization [FLB*09], this will happen already when the screen space edge length of a triangle is 64 pixels for a 16 kB texture cache.† Note that when the texture cache is thrashed, the texture bandwidth usage goes up by up to a factor of N, where N is the number of unique lens samples. Hence, all triangles that are relatively large may substantially reduce texture performance. In all fairness, Fatahalian et al.'s work on interleaved rasterization targeted micropolygons with shading at the vertex level, which means that this argument falls short in their research. However, for the foreseeable future, real-time rendering will use a wide variety of triangle sizes in the same scene, and in those cases, the traversal order of interleaved rasterization is not optimal due to texture thrashing as argued above.

4. *Depth Buffer Bandwidth Usage* Finally, we also note that a tile-based traversal order may reduce depth buffer bandwidth usage. This is the case for motion blur rasterization [MCH*11], and similar effects are expected for depth of field rasterization. Again, reductions in memory bandwidth usage will increase performance and/or reduce power usage. In this paper, we leave such a study for future work though.

As shown in this section, our arguments for a tiled traversal with a tile versus defocused triangle test are strong. The arithmetic cost of the actual test is rather insignificant in comparison if a substantial amount of memory bandwidth usage can be avoided, and the number of shader executions be reduced to, e.g., 20%, or less. Next, we present a theoretically optimal tile versus defocused triangle test, which is followed by a proof-of-concept implementation and evaluation.

4. Half-Space Culling on the Lens

As usual for depth of field (DOF) rendering, the lens area is parameterized by $(u,v) \in \Omega \subseteq [-1,+1] \times [-1,+1]$, where Ω is the aperture shape and may, for instance, be hexagonal or circular. In general, we have n samples per pixel for stochastic rasterization, where each sample consists of a spatial position, (x_i, y_i), and a lens position, (u_i, v_i). A clip-space vertex of a triangle is denoted $\mathbf{p}^i = (p^i_x, p^i_y, p^i_z, p^i_w)$, and a triangle is then $\mathbf{p}^0\mathbf{p}^1\mathbf{p}^2$. We call the plane with $w = F$, where rendered geometry will appear in perfect focus, the *focus plane*.

† We assume 64 samples per pixel and 256 lens samples are used, and that an access to a texel costs 4 bytes (RGBA), that bilinear filtering (4 texel accesses) is used, and that two textures are used in the pixel shader. With these settings, it is straighforward to verify that the texture cache will be filled up by rasterizing a triangle for *one* lens coordinate when the screen space edge length is greater or equal to 64 pixels. For a 64 kB texture cache, this increases to 128 pixels, which still is not exceptionally large.

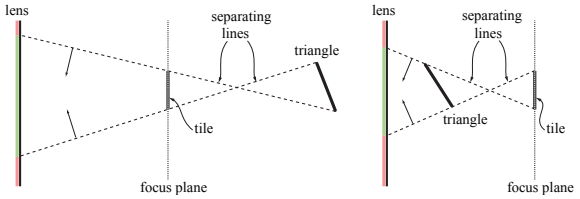


Figure 2: *From the red regions on the lens, it is impossible to "see" the triangle through the tile. The separating lines of a tile and a triangle can be used to find those regions, and this can in turn be used to reduce the number of computations needed for stochastic rasterization. Left: a triangle is beyond the focus plane. Right: a triangle is in front of the focus plane. As can be seen, when a vertex beyond the focus plane is used to define the separating plane, the positive half-space contains the tile, and vice versa.*

Not that in all our figures, the pixels and tiles are illustrated as being located exactly in the focus plane. In the text below, a *tile* is a rectangular block of pixels, and an *inside test* simply computes whether a certain sample, (x_i, y_i, u_i, v_i), is inside the triangle being rasterized. In general, the number of unnecessary inside tests should be minimized.

Next, we introduce the theoretical background and insights needed for half-space culling, which results in an *optimal*, in the sense of conservativeness, tile versus defocused triangle test. Then follows efficient ways of exploiting this new theory for depth of field rasterization.

4.1. Theoretical Background

In order to describe our algorithm, we will use the notion of *separating* planes [CT97]. A separating plane between two convex polyhedral objects is formed by an edge from one object and a vertex from the other object, and *at the same time*, the objects must lie on opposite sides of the plane. An illustration of this is shown in Figure 2 in two dimensions.

Our technique uses separating planes, derived from a tile and triangle, to remove half-space regions on the lens from further processing. We find regions on the lens which cannot "see" the triangle through any point in the tile. Samples in the tile with lens coordinates, (u_i, v_i), in such regions do not need any further inside-testing. Intuitively, this can be understood from Figure 2, where the *separating lines* (in two dimensions) of the tile and the triangle are used to cull regions (red) on the lens.

In three dimensions, however, there are two different types of separating planes that can be used to cull half-space regions on the lens. These are illustrated in Figure 3. The first set of separating planes are generated by a *tile edge* and a *triangle vertex*. Let us denote these planes by π_i, where the positive half-space of the plane consists of all points, $\mathbf{p}$, such that $\pi_i(\mathbf{p}) \geq 0$. Now, consider the example to the left in Figure 3, where the tile's left side creates a separating plane with the rightmost triangle vertex. This separating plane cuts the lens area into two half-spaces. We call the dividing line a *half-space line*, $h_i(u,v) = 0$.

Note that we choose the sign of the normal (i.e., $+\mathbf{n}$ or $-\mathbf{n}$) of the separating plane differently depending on whether the triangle vertex, forming the separating plane, is in front or behind the focus plane. The rationale for this is that we would like to cull regions where $h_i(u,v) < 0$, independent of vertex position. For vertices behind the focus plane, the separating plane's normal is chosen such that its *positive* half space contains the entire tile. In contrast, for vertices in front of the focus plane, the separating plane's normal is such that its *negative* half-space contains the entire tile. This is also illustrated in Figure 2. The direction of the two-dimensional normal of the half-space line is inherited from the corresponding separating plane's normal. These two-dimensional normals are illustrated as arrows in Figure 3. Geometrically, we can see that no point in the negative half-space on the lens can "see" the triangle through any point in the tile.

The second set of half-space lines are generated from separating planes formed from a *triangle edge* and a *tile corner*. An example is illustrated to the right in Figure 3. We denote these separating planes by Π_j to distinguish them from the planes (π_i) formed by tile edges and triangle vertices. The Π_j planes also generate half-space lines, which are denoted $H_j(u,v)$=0.

The first set of half-space lines, h_i, will be either horizontal or vertical, while in general, the second set of half-space lines, H_j, can have arbitrary orientation. When all tile edges each generate a separating plane, they may form a two-dimensional box in the plane of the lens. With such a box, it is simple and efficient to cull large portions of the samples in the entire tile from further processing. An example is shown to the left in Figure 4. To the right in the same figure, further regions on the lens have been culled away by the H_j lines. When all triangle vertices either are in front of or behind the focus plane, the remaining *active* region on the lens is defined by a convex region, where $h_i(u,v) \geq 0$ and $H_j(u,v) \geq 0$. The key to our tile test is that it is only for samples with their lens coordinates, (u_i, v_i), inside this active region (green region in Figure 4) that needs to be inside-tested.

Between a tile and a triangle, there are no more types of separating planes, and hence, when all the h_i and all H_j half-space lines are used to compute the active region, the active region cannot be further reduced in size on the lens. This implies that the size of the active region is *minimal*, which in turn means that it provides a limit for how efficient (in terms of active region area) a practical defocused tile test can become.

Note that it is not always possible to create separating planes. This happens, for example, when a triangle cuts through a tile, which implies that the triangle goes from

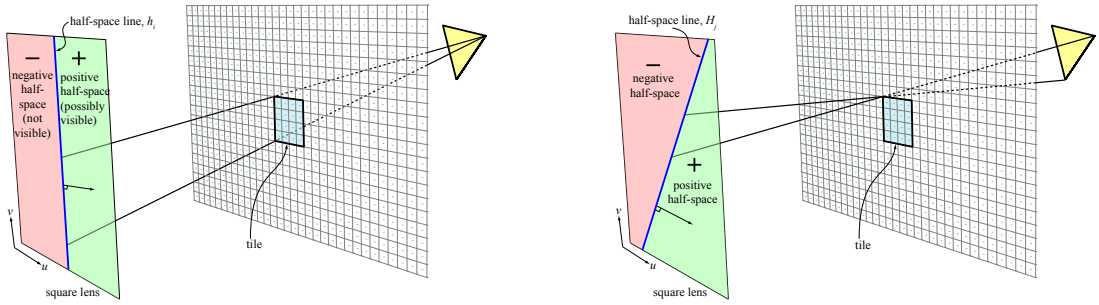


Figure 3: *Left: in this example, the rightmost vertex of the yellow triangle forms a separating plane with the leftmost tile edge. This plane intersects the lens area, and divides it into two half-spaces: the negative half-space (red), where its points on the lens cannot see the triangle through any point in the tile, and the positive half-space (green), whose points on the lens potentially can see the triangle. The positive half-space is the defined by the side of the plane where the tile itself is located. Right: a separating plane formed from a triangle edge and a tile corner.*

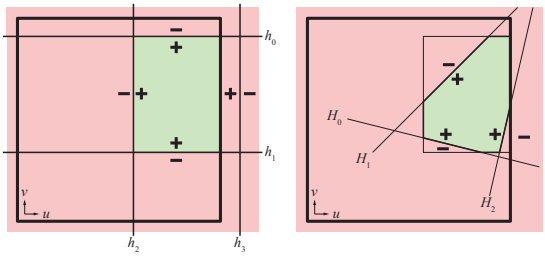


Figure 4: *The lens is the thick-lined square. Left: the first four separating planes generate horizontal and vertical half-space lines, which are defined by $h_i(u,v) = 0$. Together, they form a two-dimensional bounding box in the lens plane in this example. Only the samples with lens positions in the green area need to be further processed. As can be seen, using only the first four planes can cull away a substantial region of the lens. Right: the second set of half-space lines, $H_j(u,v) = 0$, can further reduce the green region.*

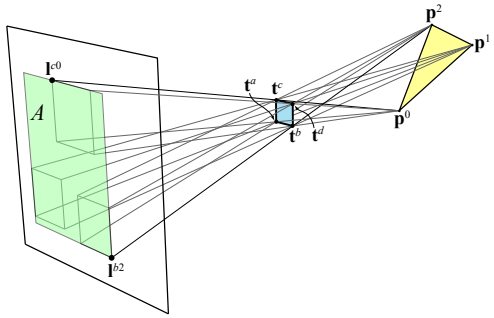


Figure 5: *Notation for computing half-space lines, h_i. The vertices, $\mathbf{p}^0$, $\mathbf{p}^1$, and $\mathbf{p}^2$, of a triangle are projected through the tile corners, $\mathbf{t}^a$, $\mathbf{t}^b$, $\mathbf{t}^c$, and $\mathbf{t}^d$, onto the lens to form the lens coordinates, $\mathbf{l}^{ij}$, where $i \in \{a,b,c,d\}$, and $j \in \{0,1,2\}$. For example, the vertex $\mathbf{p}^2$ is projected through $\mathbf{t}^b$ to generate $\mathbf{l}^{b2}$. This figure also shows an uv-box (green), denoted A, in the lens plane. The area outside the green box does not need any further inside testing.*

front- to backfacing in that tile. As a consequence, it is possible to add a half-space line, defined by the intersection of the plane of the triangle and the lens, to cull parts of the lens where the triangle is backfacing [MAM11]. This could be beneficial for triangles close to the silhouette of the object. In practice, there is usually only a fraction of such triangles in a scene, and as a consequence, we have not found that using this backfacing plane test for culling pays off.

In the following two subsections, we will show how to efficiently compute the two different types of half-space lines. In addition, these subsections will also explain how the half-space lines are computed when the triangle intersects the focus plane. Such situations are more complex and require special handling, as we will see.

4.2. Efficient Computation of Half-Space Lines h_i

In this subsection, we describe how we find the half-space lines, h_i, which are generated from separating planes, π_i, through a tile edge and a triangle vertex. A straightforward way to determine whether a plane is separating, is to test whether the two triangle vertices that were *not* used to define the plane, are both on the opposite side of the plane compared to the tile. We will describe an efficient way of doing this.

First, however, we present the notation used to describe our algorithm. As before, the triangle vertices are called $\mathbf{p}^0$, $\mathbf{p}^1$, and $\mathbf{p}^2$. The four corners of a tile are denoted by $\mathbf{t}^a$, $\mathbf{t}^b$, $\mathbf{t}^c$, and $\mathbf{t}^d$. The projection of a vertex, $\mathbf{p}^j$, through a tile corner, $\mathbf{t}^i$, onto the lens will be denoted by $\mathbf{l}^{ij} = (l_u^{ij}, l_v^{ij})$. See Figure 5 for an illustration of our notation.

In the following, we will describe how the projected points, $\mathbf{l}^{ij}$, are computed efficiently, and how they are used to, for example, compute areas on the lens where sample testing is not needed. An example of this is the region outside the green box in Figure 5. The projection of a vertex, $\mathbf{p}^j$, through a tile corner, $\mathbf{t}^i$, gives us the lens coordinates of

$\mathbf{l}^{ij}$ as:

$$\begin{aligned} l_u^{ij} &= -\frac{p_x^j F}{p_w^j - F} + t_x^i \frac{p_w^j}{p_w^j - F} = o_u^j + t_x^i \delta^j, \\ l_v^{ij} &= -\frac{p_y^j F}{p_w^j - F} + t_y^i \frac{p_w^j}{p_w^j - F} = o_v^j + t_y^i \delta^j. \end{aligned} \quad (1)$$

As can be seen in the equations above, the offsets, o, and the deltas, δ, are constant when rendering a particular triangle, and these can therefore be computed in a triangle setup. In addition, there is only a linear dependence on the tile coordinates, (t_x^i, t_y^i), implying that they can be evaluated efficiently using incremental updates when moving from one tile to the next. Note that if $p_w^j - F = 0$, then the corresponding coordinate can never generate a half-space line on the lens that will cull anything. The reason is that the projections, (l_u^{ij}, l_v^{ij}), will approach $\pm\infty$, which will never help in culling on a finite lens. Hence, such vertices are always ignored for the rest of the computations.

The half-space lines h_i will be either horizontal or vertical, so all computations can be done in two dimensions. Let us assume that we want to determine whether a plane from one of the triangle vertices, $\mathbf{p}^j$, is a separating plane through a tile edge located at $x = t_x^i$. In this case, all calculations can be done entirely in the xw-plane. In the following, recall that the focus plane is at $w = F$, and the lens is at $w = 0$.

To determine whether $u = l_u^{ij}$ actually defines a half-space line from a separating plane, we want to determine in which half-space the other two triangle vertices are located. If they both are *not* in the same half-space as the tile itself, we have found a separating line. More specifically, this is done as follows. We set $\mathbf{q} = \mathbf{p}^j$ and let $\mathbf{r}$ be one of the other triangle vertices. A two-dimensional line equation from (q_x, q_w) to (t_x^i, F) is derived, and the two other triangle vertices are inserted into the line equation. We find that the line equation evaluated at a point, (r_x, r_w), is:

$$\begin{aligned} e(\mathbf{q},\mathbf{r}) &= r_x(q_w - F) + q_x(F - r_w) + t_x^i(r_w - q_w) \\ &= O_{qr} + t_x^i \Delta_{qr}, \end{aligned} \quad (2)$$

which is linear in t_x^i. Note also that $e(\mathbf{q},\mathbf{r}) = -e(\mathbf{r},\mathbf{q})$, and so for a given tile, only $e(\mathbf{p}^0,\mathbf{p}^1)$, $e(\mathbf{p}^1,\mathbf{p}^2)$, and $e(\mathbf{p}^2,\mathbf{p}^0)$ need to be evaluated. In general, for $u = l_u^{ij}$ to define a separating line, the two other triangle vertices should be in the negative half-space when $p_w^j > F$, and in the positive half-space when $p_w^j < F$, as can be seen in Figure 2. The case when $p_w^j - F = 0$ is ignored, as described previously, because such projections will not provide any culling.

For example, given the vertex $\mathbf{q} = \mathbf{p}^0$, and the leftmost tile edge, $x = t_x^a$, we test if $u = l_u^{a0}$ is a separating half space-line by evaluating the line equation (Equation 2) for $\mathbf{r} = \mathbf{p}^1$ and $\mathbf{r} = \mathbf{p}^2$. If it is separating, the corresponding half-space line,

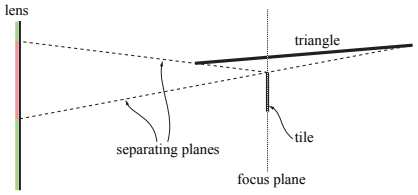


Figure 6: *When a triangle intersects the focus plane, a tile side can generate two separating planes as shown in this illustration. Note that culling is done in the "inner" region (red) in this case, in contrast to when the triangle does not intersect the focus plane (see Figure 2). In this case, we generate one uv-box for the topmost green region and one for the bottommost region on the lens.*

$h(u,v)$, is defined by:

$$h(u,v) = \begin{cases} u - l_u^{a0}, & \text{when } p_w^0 > F, \\ l_u^{a0} - u, & \text{when } p_w^0 < F, \end{cases} \quad (3)$$

which is a vertical half-space line on the lens. Note that the culling "direction" changes depending on whether the vertex is in front of the focus plane, F, or behind it. In addition, the $p_w > F$ tests are reversed when testing against the rightmost tile edge, $x = t_x^d$. Similar equations are created for all lens coordinates, l_u^{ij} and l_v^{ij}, which have been verified to be real separating planes (using Equation 2). This is all that is needed for computing the horizontal and vertical half-space lines, $h_i(u,v) = 0$.

Finally, we describe how the half-space lines are combined to form, what we call, uv-boxes in the lens plane. Only samples inside a uv-box need further processing (and can be further culled using the H_j lines). Note that we use the term *box* broadly because they can extend infinitely in some directions, as we will see. For this discussion, we split the triangle vertices into two sets. The first set consists of all vertices located in front of the focus plane, and the second set consists of all vertices behind the focus plane. When one set is empty, all triangle vertices are located on one side of the focus plane. In these cases, it is straightforward to generate a box in the uv domain on the lens using the h_i lines. An example is shown in Figure 5.

However, when both sets are non-empty, and there are separating half-space lines generated from both sets, *two* uv-boxes will be generated. This can be seen in two dimensions in Figure 6. When there are two separating half-space lines generated in one set, both these will be used to define that set's uv-box. This is illustrated in Figure 7, where the uv-boxes extend infinitely in some directions because these boxes are only generated from two half-space lines.

The uv-boxes determine active regions on the lens, and for all pixels within the tile, only samples within these regions need to be tested against the triangle. In the next subsection, we will show how these active regions can be further reduced.

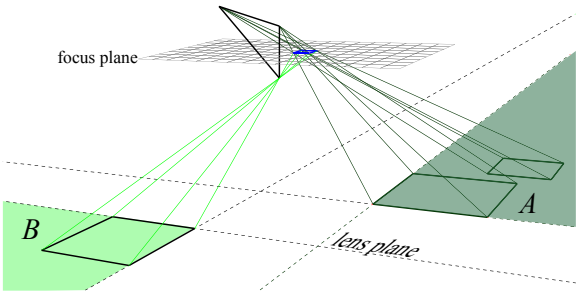


Figure 7: *Visible regions for the triangle seen through the blue tile with one triangle vertex in front of the focus plane, and two behind. In this case, we generate two uv-boxes (green), A and B, on the lens, and inside testing is only done for samples with uv-coordinates in these boxes.*

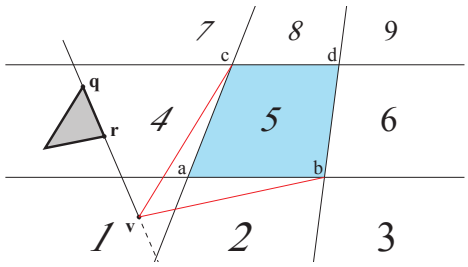


Figure 8: *The triangle edge,* **qr**, *is intersection tested against the focus plane, which generates the intersection point,* **v**. *The tile abcd (shown in blue) divides the focus plane into nine regions, numbered 1–9. In the example above, the intersection point,* **v**, *lies in region 1, and therefore the candidate corners to form separating planes are b and c. In this illustration, those are shown as red lines, which form tangents between* **v** *and the tile.*

4.3. Efficient Computation of Half-Space Lines H_j

To find the second type of half-space lines, H_j, we use an approach, based on Coorg and Teller's [CT97] method, for finding separating planes from a triangle edge and tile corner. This includes two steps, namely, finding a set of candidate planes and then finding out whether those candidates indeed are *separating*.

To find candidate planes given a triangle edge, **qr**, we form the ray $\mathbf{q}+t(\mathbf{r}-\mathbf{q}) = \mathbf{q}+t\mathbf{d}$. Referring to Figure 8, we compute the intersection point, **v**, between the ray and the focus plane. We divide the focus plane into nine regions and identify which region **v** falls into. The intersection point, **v**, lies in a certain tile, whose tile coordinates can be precomputed and used for all subsequent computations. During traversal, the only work needed per tile is thus a comparison of the tile coordinate with these pre-computed tile coordinates.

With **v** categorized into one of the nine regions, we identify the two tile corners m and n which form the largest angle $\angle\mathbf{t}^m\mathbf{v}\mathbf{t}^n$. These can be tabulated as follows:

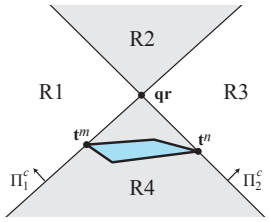


Figure 9: *The two candidate planes, Π_1^c and Π_2^c, divide space into four regions, shown as R1–R4. By construction, the triangle edge,* **qr**, *is included in both these planes. Whether Π_1^c and Π_2^c are separating depends on which region the third triangle vertex,* **s**, *is in. If* **s** *is in region R1 or R2, Π_1^c is a separating plane. Π_2^c is separating if* **s** *lies in R2 or R3. A triangle edge,* **qr**, *can thus produce zero, one, or two separating planes.*

Region	1	2	3	4	5	6	7	8	9
m	b	b	d	a	-	d	a	c	c
n	c	a	a	c	-	b	d	d	b

Using this table, we can form two *candidate* planes $\Pi_1^c : (\mathbf{q},\mathbf{r},\mathbf{t}^m)$ and $\Pi_2^c : (\mathbf{r},\mathbf{q},\mathbf{t}^n)$. Using the sign of the d_w-component, i.e., whether the edge, **qr**, points towards the camera, we can choose plane normal directions such that the tile is in the negative half-space of the respective plane. For edges parallel to the focus plane (i.e., $d_w = 0$), we use (d_x, d_y) to determine m and n and the sign of $q_w - F$ to determine the normal direction. Note that there are no candidate planes for region five, since in this region, the edge is pointing straight at the tile and there cannot exist any separating planes between the edge and a tile corner. Likewise, any edge **qr** where $\mathbf{q}_w = \mathbf{r}_w = F$, cannot produce a useful half-space line and is thus ignored.

Given two candidate planes, Π_1^c and Π_2^c, it is now time to determine whether they are separating planes. To determine this, the third vertex of the triangle is tested against these planes. If the vertex is in the positive half-space of a plane, that plane is a separating plane. Each triangle edge can generate up to two separating planes, as can be seen in Figure 9.

For a given a plane, $\Pi : \mathbf{n}\cdot\mathbf{p}+c = 0$, the corresponding half-space line, H, on the lens, $(u,v,0)$, is:

$$H(u,v) = \mathbf{n}\cdot(u,v,0)+c = n_x u + n_y v + c. \qquad (4)$$

This equation varies non-linearly and must therefore be computed for each tile. To see this, consider again the triangle edge, **qr**, and a tile corner, **t**, in the focus plane. A plane through these points is defined by $\mathbb{P} : \mathbf{n}\cdot\mathbf{x}+c = 0$, where:

$$\begin{aligned} \mathbf{n} &= (\mathbf{q}-\mathbf{t})\times(\mathbf{r}-\mathbf{t}) = \mathbf{q}\times\mathbf{r}+(\mathbf{r}-\mathbf{q})\times\mathbf{t}, \\ c &= -\mathbf{n}\cdot\mathbf{t} = -\mathbf{t}\cdot(\mathbf{q}\times\mathbf{r}). \end{aligned} \qquad (5)$$

Note that the normal **n** and c change as we move in screen space. For example, the change in n_x and n_y is proportional to the difference in depth between **r** and **q**.

Finally, the normals of the planes, Π_i, need to be handled

carefully when culling samples on the lens. When culling regions from *uv*–box A (illustrated in Figure 7), samples in the positive half-space of Π_i can be culled. However, when culling regions from *uv*–box B, samples from the *negative* half-space of Π_i can be culled.

5. Implementation

Section 4 reveals the theory for an exact tile vs defocused triangle test. It is very likely that this theory can be turned into practice in a variety of ways. In this section, we present *one* such implementation as a proof of concept. However, we believe that other implementation variants may improve performance further.

In addition, we outline our two depth of field rasterizer implementations based on INTERVAL and INTERLEAVE [FLB*09], adapted to support mixed triangle sizes, instead of just micropolygons. All algorithms, including a brute-force DOF rasterizer, have been implemented in a functional software simulator focusing on the rasterization part of the graphics pipeline. Similar to current GPU pipelines, we assume that shading is computed after visibility at the pixel level in all our implementations. This is in contrast to micropolygon rasterizers, which usually shade at the vertex level. Our simulator is missing a complete shader system, but the important pieces are in place in order to be able to measure DOF rasterization efficiency, which is the objective of our research.

5.1. Our Algorithm

Up until now, the half-space lines, h_i and H_j, which lie in the plane of the lens, have been computed in an exact manner. In this subsection, we will first describe how they can be exploited to quickly cull samples for faster depth of field rasterization. The half-space lines are computed on a per-tile basis, and hence culling opportunities will be shared between all the pixels in a tile. However, the process still needs to be very efficient. As will be seen, determining which samples lie within the active subspace of the lens can be implemented as a rasterization process in itself.

In our conceptual implementation, we superimpose a square grid on top of the lens shape and in an initialization step, we create a table of the number of samples falling into each grid cell. This is illustrated in Figure 10. These sample distributions vary from pixel to pixel, and we use a small set (32×32) of distributions that are repeated over the screen.

The half-space lines, h_i, which are generated from triangle vertices and tile edges, provide an easy means of culling since they are axis-aligned. We can simply clamp them down and up to the nearest grid cell and use the clamped rectangular extents to quickly traverse relevant samples. Additionally, we test each sample that passes the grid-cell test against the h_i half-space lines.

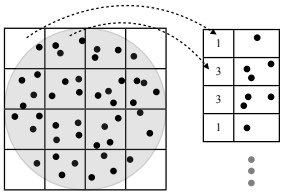


Figure 10: *Lens grid with corresponding storage. For each grid cell, we store the number of samples within that cell, and an offset pointing to the first sample in that grid cell. With this layout, we can efficiently cull large sets of samples against the separating half-space lines.*

For the non-axis-aligned half-space lines, H_j, we iterate over all grid cells in the rectangular extents computed from the h_i lines and conservatively test for overlap between the grid cells and the H_j lines. This essentially boils down to a micro-rasterization process for every screen space tile in order to cull the samples. One way to optimize this is to exploit the limited resolution of the lens grid and use a precomputed rasterizer [KLA04, LK10]. However, we instead exploit the fact that two-dimensional edge equations can be updated incrementally from one pixel to the next [Pin88], and evaluate each H_j using additions for a small grid to get conservative micro-rasterization.

In the following pseudo-code, we outline the full algorithm for rendering depth of field using half-space line culling.

```
compute BBox of defocused triangle Δ
compute initial h0, ..., h3 half-space lines
for all tiles T in BBox do
  inexpensive update of h0, ..., h3
  compute UVBoxes from h0, ..., h3
  if any UVBox overlaps lens shape then
    compute Hj half-space lines
    bitmask = compute overlap bitmask for all Hj
    for all pixels p in T do
      for all grid cells C in UVBoxes do
        if bitmask[C] == True then
          test samples in C(p)∩UVBoxes against Δ
        end if
      end for
      shade pixel (multisampling)
    end for
  end if
end for
```

As a further optimization, we can also disable the H_j line test (the grey lines in the pseudo code) for highly defocused or near axis-aligned edges. To estimate the culling efficiency of an H_j line, we use the screen space area from the axis-aligned bounding box of the screen space projected edge. Given the screen space projections of two triangle vertices, (x_i, y_i), $i \in 0, 1$, as seen from the center of the lens, we form $A_{cull} = |x_1 - x_0||y_1 - y_0|/2$. If A_{cull} is small compared to

the the screen space bounding box of the defocused triangle, the H_j test for that edge is disabled. This test is performed in the triangle setup. For more details, please refer to Appendix A. For our test scenes, this selective H_j test reduced total arithmetic cost with up to 10% compared to always testing against all h_i and H_j lines.

5.2. Interval DOF Rasterizer

For evaluation purposes, we have also implemented an interval-based DOF rasterizer [FLB*09]. Here, the *uv*-lens domain is divided into a grid, and we perform one DOF rasterization pass for each grid cell. Within each pass, the defocused triangle is bounded for the active *uv*-grid cell, and only the lens samples within the current cell need to be inside-tested. We include pseudo-code for the algorithm below.

```
for all grid cells C do
  compute BBox of defocused triangle Δ over C
  for all tiles T in BBox do
    for all pixels p in T do
      test samples in C(p) against Δ
      shade pixel (multisampling within C)
    end for
  end for
end for
```

The main differences between our method and interval-based DOF rasterization are the iteration order and the lack of a tile overlap test in the interval-based rasterizer. For the interval-based traversal order, the same screen space tile may be visited multiple times. With shading at the pixel level, each pixel is shaded once for each grid cell, unless there is a shader cache [RKLC*11, BFM10] available. A shader cache is a very attractive option for inexpensive shading in the future. However, adding hardware support for depth of field in the rasterizer may be a smaller change, and so it is likely that happens first. A natural next step is then to add the shader cache.

5.3. Interleave DOF Rasterizer

In the interleave-based DOF rasterizer [FLB*09], we rasterize the triangle at N discrete *uv*-lens coordinates. We perform N rasterization passes, but in each pass, only a $1/N$ fraction of the screen space sample positions are evaluated. Within each pass, the triangle is bounded for the active lens coordinate. We include pseudo-code for the algorithm below.

```
for all lens coordinates (u_i, v_i) do
  compute BBox of triangle Δ at lens coordinate (u_i, v_i)
  for all tiles T in BBox do
    for all samples with (u, v) = (u_i, v_i) in T do
      test sample against Δ
      shade sample (supersampling)
    end for
  end for
end for
```

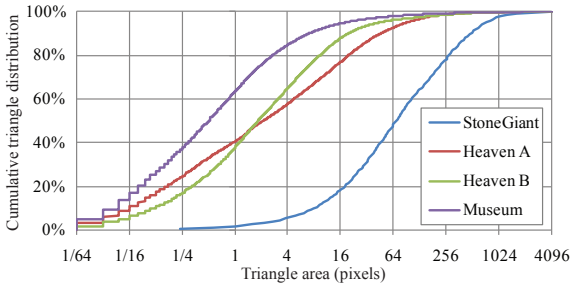


Figure 11: *Cumulative distribution function of the triangle sizes in our test scenes.*

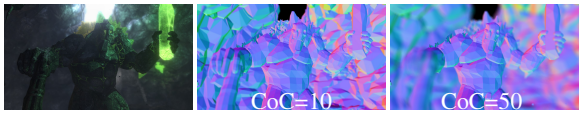
Note that the iteration order is over lens coordinates, so the same screen space region will be visited multiple times for one defocused triangle. We have also implemented a variant of this algorithm with an overlap test for each coarse tile. With 256 tuples and 64 samples per pixel, we execute a standard tile vs static triangle (for a particular lens coordinate) test on 8×8 pixel tiles. Each test can cull up to 16 sample tests, since for a given lens coordinate, there is only one sample per 2×2 pixels.

6. Results

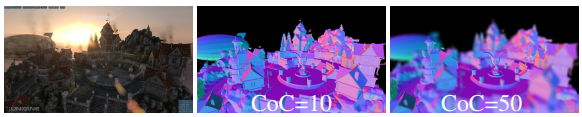
In Figure 12, we present the four test scenes that we have used in order to gather statistics for various depth of field rasterization algorithms. StoneGiant has relatively large triangles and a small triangle count. Heaven A has 6× the number of triangles and shows a zoomed-out view with relatively small triangles. Heaven B has 2× the number of triangles compared to Heaven A with a close-up view. These demos can be run with different tessellation settings, and for this study we use the setting *disabled* (StoneGiant, Heaven A) and *normal* (Heaven B) to reach a rich variety of triangle distributions. The Museum scene contains a mix of both many small and some large triangles and has a total of 1.5M triangles. Figure 11 shows an overview of the triangle size distribution in the different scenes. To find out more about the properties about the different rasterization algorithms, we render all test scenes with varying amounts of defocus blur. We use the term circle of confusion (CoC) to denote the defocus blur in pixels when the depth approaches infinity.

Our results contain both a comparison of DOF rasterization algorithms that allow for *arbitrary* sampling patterns, and a comparison of DOF rasterization algorithms that use *interleaved* sampling patterns [KH01]. All these algorithms are summarized next.

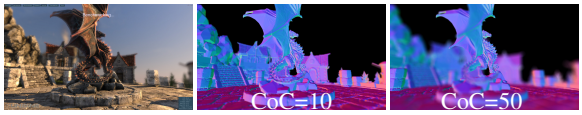
The DOF rasterization algorithms that allow for arbitrary sampling patterns are BRUTEFORCE, INTERVAL, and OUR. BRUTEFORCE computes the bounding box of a triangle as seen through the four corners of the square lens, and then performs inside testing for all samples in the bounding box. As described in Section 5.2, INTERVAL dices the



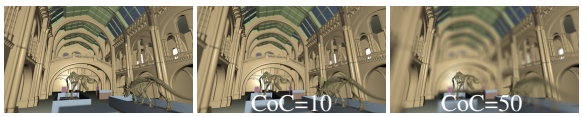
StoneGiant, Courtesy of BitSquid,
65k triangles, 1280 × 720 pixels, 3.6× overdraw



Heaven A (v 2.1), Courtesy of Unigine Corp.,
0.41M triangles, 1920 × 1080 pixels, 2.4× overdraw



Heaven B (v 2.1), Courtesy of Unigine Corp.,
0.83M triangles, 1920 × 1080 pixels, 3.1× overdraw



Museum, Courtesy of Alvaro L. Bautista and
Joel Anderson, 1.5M tris, 1920 × 1080 pixels, 7× overdraw

Figure 12: *The test scenes that we use for our measurements.*

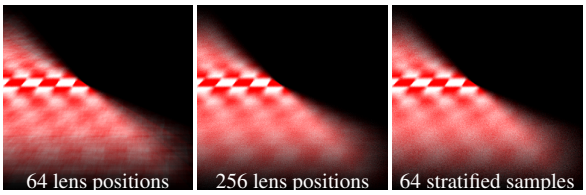


Figure 13: *Quality of using interleaved sampling (with 64 and 256 lens positions) compared to 64 unique stratified samples per pixel (right). All images were generated with 64 samples per pixel, and permuted (x,y) sample positions for each block of 2×2 pixels.*

	StoneGiant	Heaven A	Heaven B	Museum
CoC=0	1.00	1.00	1.00	1.00
CoC=10	1.20	1.22	1.27	1.25
CoC=50	1.47	1.29	1.27	1.56

Table 1: *Efficiency of our traversal algorithm presented in Section 5.1. The numbers show the ratio between the number of samples tested with our traversal algorithm and the optimal number of samples that need to be tested according to our theory presented in Section 4. The comparison is using 2×2 pixel tiles. Even for large blurs, we only test a modest amount of samples with (u,v)-coordinates that cannot possibly hit the triangle within a screen space tile.*

lens into a number of grid cells, and performs BRUTEFORCE-rasterization for each grid cell's corresponding *uv*-domain. OUR algorithm is described in Section 5.1, and uses half-space lines to cull samples for each screen space tile. Except where otherwise mentioned, we use a grid of 4×4 cells on the lens both for INTERVAL and OUR. BRUTEFORCE, INTERVAL, and OUR can use any type of sampling pattern, and all use the sample access method described in Section 5.1. For OUR, we use a tile size of 2×2 pixels, which was found to generate best results. Also, unless mentioned otherwise, the images were rendered with 64 samples per pixel.

In addition, we compare three algorithms that exploit an interleaved sampling pattern with 256 fixed lens coordinates. We choose 256 lens coordinates since the resulting image quality is close to using 64 unique stratified samples per pixel as shown in Figure 13. We implemented two versions of the interleaved rasterization algorithm (Section 5.3). These are called INTERLEAVE and HINTERLEAVE, where the latter has a tile vs. triangle test at the 8×8 pixel level. An interleaved sampling pattern for our algorithm results in that the per-sample test uses a standard two-dimensional edge test instead of using four-dimensional edge equations. This reduces the per-sample cost by 50%, but increases the triangle setup cost.

6.1. Comparison with our theory

Table 1 shows the efficiency of our traversal algorithm (Section 5.1) for various CoC radii in terms of sample tests. The results are relative to the samples within the lens region of the optimal tile test given by the H_j and h_i half-space lines. The source of inefficiency is the quantization of the lens into grid cells, and as a result, the region on the lens visible though a tile grows. While the exact h_i half-space lines are used for per-sample culling, the H_j half-space lines are not. Still, our implementation is reasonably close to the theoretical optimum, even for very large CoC radii.

6.2. Arithmetic Intensity

We have gathered results in terms of arithmetic intensity for varying amounts of defocus blur. In our measurements, arithmetic intensity includes the operations needed for triangle setup, tile testing, and sample testing, and disregards from operations common to all algorithms, such as near-plane and backface culling. To reason about the general trend of arithmetic intensity, we count instructions, such as ADD, MADD, etc, as one operation, and exclude control logic and simple bit manipulation. This is similar to the methodology used by Fatahalian et al. [FLB*09]. In Figure 14, we show a cost breakdown for the algorithms for one frame from the Museum scene.

The arithmetic intensity results are presented in Figure 15. We start by interpreting the results from the top row, i.e., for the algorithms using arbitrary sampling patterns. For this set of test scenes, OUR is the most efficient technique for all aperture sizes. For all scenes, the BRUTEFORCE algorithm

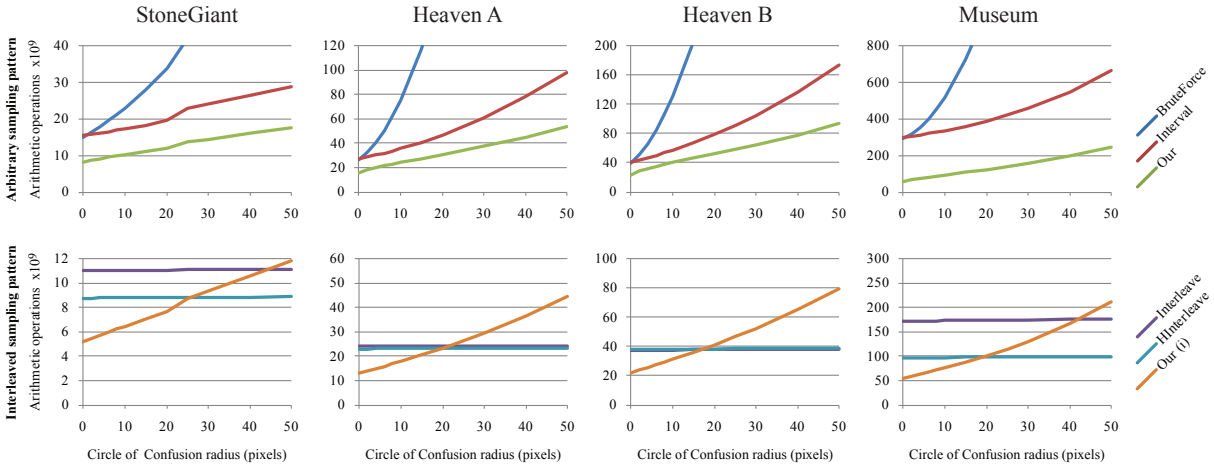


Figure 15: *Results in terms of arithmetic intensity for the different depth of field rasterization algorithms. All figures show the arithmetic intensity as a function of the limit circle of confusion radius. The top row shows algorithms that can have an arbitrary sampling pattern. The bottom row shows algorithms that rely on an interleaved sampling pattern.*

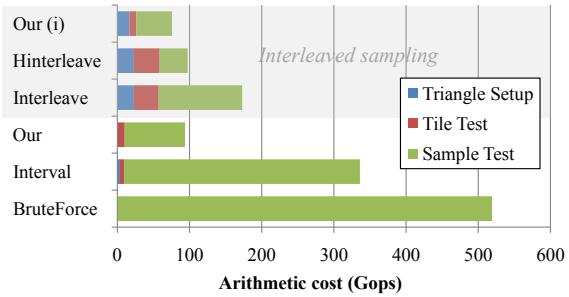


Figure 14: *Cost breakdown of the different algorithms for the Museum scene with a circle of confusion radius of 10 pixels. The top rows (gray background) show the algorithms using interleaved sampling, and the bottom rows show algorithms with arbitrary sampling patterns.* OUR *(i) is* OUR *algorithm with an interleaved sampling pattern.*

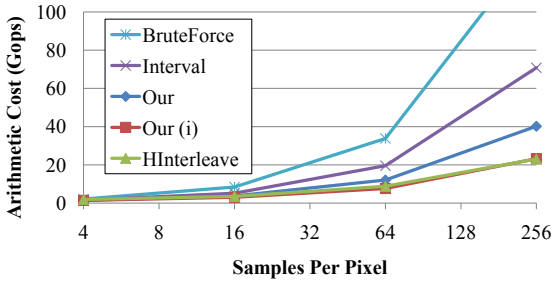


Figure 16: *Results in terms of arithmetic operations versus the number of samples per pixel for the StoneGiant scene with a circle of confusion radius of 20 pixels. Our algorithm has a consistently lower cost than* INTERVAL. *With interleaved sampling patterns, our algorithm scales similar to* HINTERLEAVE. *For the 256 spp data point, we use a grid resolution of* 8×8 *for* OUR, OUR (I), *and* INTERVAL *(and* 4×4 *for all other sampling rates).*

quickly becomes very expensive. It is interesting to see that even though Heaven B contains more than twice the number of triangles, the efficiency of OUR compared to INTERVAL does not change drastically. For the Museum scene, the triangle distribution is more varied, with both many small triangles and a set of large triangles covering many pixels. The triangles larger than 4k pixels are very few, but still, they represent 40% of the total coverage in this scene. As can be seen, tile-based half-space culling is very efficient for this workload.

In the lower row of Figure 15, we compare the algorithms based on interleaved sampling patterns. For StoneGiant and Museum with many large triangles, HINTERLEAVE is clearly more efficient than INTERLEAVE, and our algorithm is competitive up to a CoC radius of about 20 and 40 pixels, respectively. In the Heaven scenes, most triangles are very small, and the benefit of a tile-based overlap test is low. OUR algorithm is competitive up to a CoC radius of about 20 pixels. In all scenes, our algorithm has about half the arithmetic cost compared to HINTERLEAVE for small defocus blurs.

In Figure 16, we show how the algorithms scale with increasing number of samples per pixel (spp). For higher sampling rates, the cost of our tile tests is amortized over more samples, and the benefit of half-space culling increases. For a small number of samples (e.g., 4 spp), half-space culling is less beneficial. However, stochastic DOF without local reconstruction filters [SAC*11,LAC*11] at 4 spp contains significant noise even for modest aperture sizes.

Table 2 shows the culling efficiency obtained from the two set of half-space lines, h_i & H_j, and their combination. The h_i lines are most effective for triangles with large circle of

		None	h_i	H_j	h_i+H_j	INTERVAL
R=10	Culled	0%	30%	62%	65%	27%
	Cost (Gops)	23	19	10	10	17
R=50	Culled	0%	79%	72%	86%	66%
	Cost (Gops)	83	24	26	18	29

Table 2: *Culling efficiency (higher is better), and total rasterization arithmetic cost, of h_i, H_j, and their combination. Numbers are measured with the StoneGiant scene with two different circle of confusion radii (R). The combination of the two types of half-space lines efficiently reduces the number of samples tested. Tile edge tests were not adaptively disabled (Appendix A) for these measurements.*

confusion compared to the triangle size, while the H_j lines are most effective on triangles with modest blur.

6.3. Comparison with Laine et al's DOF algorithm

A similar test to our h_i lines was developed independently by Laine et al. [LAKL11]. Here we present a brief comparison on a controlled scene with varying triangle sizes and blur amount. In Figure 17, we show both arithmetic intensity and sample test efficiency (STE) [FLB*09] using 2×2 pixel tiles and 64 spp. Our algorithm has slightly higher STE on all scenes, with a significant difference in Scene A and C, where the H_j lines are very beneficial. The arithmetic intensity is about 50% of Laine's algorithm on all four scenes. Our algorithm may reject entire tiles (using the H_j lines) and with the grid over lens samples, cells of samples are quickly discarded. In contrast, Laine's algorithm (for DOF only) tests each individual sample against the valid lens region, and there is no early out at the tile level. For both algorithms, we count a test: $u < u_{min}$, as one arithmetic operation. The arithmetic intensity is dominated by the sample cost. With 2×2 pixel tiles, our algorithm spends 9% on evaluating tile tests (Laine: 0.4%). With 1×1 pixel tiles, our algorithm spends 20% on tile tests (Laine: 2%), and the total instruction count increases with 9% (Laine decreases with 7%). Still, comparing Laine's algorithm using 1×1 pixel tiles with our algorithm at 2×2 pixel tiles, there is still a $1.9\times$ instruction ratio in our favor when measured over all four scene configurations.

6.4. Potential for Multi-Sampling Anti-Aliasing

In micropolygon rasterization, shading is executed per vertex. With varying triangle sizes, however, per-vertex shading is too coarse, and standard GPU pipelines therefore shade per pixel. In the presence of depth-of-field with many stochastic visibility samples per pixel, it is critical to reduce the number of shader evaluations in a similar way. One way to do this is to implement a shading cache, where visibility samples mapping to similar barycentric regions of the triangle share a shaded value [RKLC*11, BFM10]. A coarser but simpler approximation is to extend standard multi-sampling

		A	B	C	D
Gops	Our	0.77	1.6	1.9	3.7
	Laine	1.4	3.9	4.0	7.3
STE	Our	27%	18%	42%	26%
	Laine	18%	17%	22%	21%

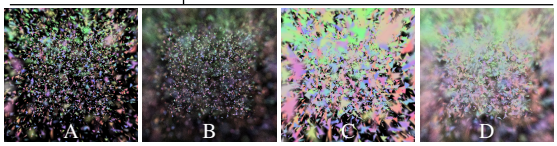


Figure 17: *A controlled test scene with 20k triangles with varying triangle sizes and defocus blur. We report the arithmetic intensity in Gops and the sample test efficiency (STE) using 64 spp and 2×2 pixel tiles.*

CoC radius:	0	10	20	30	50
OUR	1%	2%	4%	6%	9%
INTERVAL	10%	12%	14%	16%	19%

Table 3: *Number of quad shading requests (lower is better) for the Museum scene with varying circle of confusion (CoC) radii relative to supersampling (256 samples per 2×2 pixel block at 64 samples per pixel). Even for large defocus blurs, where fewer samples within a quad hit the same triangle, there is still significant MSAA potential for our algorithm.*

anti-aliasing (MSAA) to handle depth-of-field, where all visibility samples within a pixel from the same triangle share a common shaded value [MESL10]. Below, we measure the potential for MSAA in the different algorithms.

The iteration order in the traversal algorithms affects the potential for multi-sampling substantially. INTERLEAVE and HINTERLEAVE iterate over lens positions and this results in supersampling. INTERVAL iterates over grid cells on the lens, and will therefore shade (at least) once per grid cell. This implies 16 shader executions per pixel for 4×4 grid cells when the triangle is visible from all points on the lens. In contrast, with the screen space tile iteration order in OUR algorithm, we have the possibility to execute the shader only once per pixel for a triangle. The MSAA potential in the Museum scene is presented in Table 3. Sufficient image quality may not always be obtained by shading once per pixel in defocused areas. Note, however, that the shading rate could be controlled per shader and by the amount of defocus blur for the pixel being rasterized. For example, a diffuse shader may need very few (even just one) shading samples, while a specular shader may need more. In addition, unless the shader is highly view-dependent, there is little reason to super sample shading for surfaces that lie in perfect focus or very near the focus plane. Hence, we have made it clear that MSAA has potential to reduce shader computations and related memory bandwidth usage substantially for depth of field rasterization. A thorough investigation of when and how the number of shading samples can be reduced with retained image quality is out of the scope of our work here, and is therefore left for future work.

7. Conclusions and Future Work

Our new theory for the tile versus defocused triangle test provides a limit on how efficient such a tile test ever can be. As such, we hope that our research makes for a deeper understanding about depth of field rasterization. In our work, we have also presented a conceptual implementation of depth of field rasterization, and shown that our algorithm uses significantly fewer arithmetic operations when arbitrary sampling patterns are used. As we have seen, our algorithm also scales better than previous algorithms with increased sampling rates and our algorithm is better at handling mixed triangle sizes. The arithmetic intensity reveals some properties of the algorithms, but an additional important advantage of our algorithm is that it visits a tile only once per triangle. This enables multi-sampling anti-aliasing, which can reduce shader computation and memory bandwidth usage, and this in turn may increase performance substantially.

We want to emphasize that we have implemented only one variant of depth of field rasterization based on the new theory. However, getting closer to the bounds is surely possible, and we believe that future research will reveal new depth of field rasterization algorithms based on our new theory. We also hope that our research brings us a little closer to getting efficient depth of field rasterization for real-time graphics. As a next step, it would be interesting to devise an efficient fixed-point implementation of our algorithm. We also believe that our theory could be used as the basis for a novel 5D rasterizer, including both motion blur and depth of field at the same time, and this is a clear direction to continue doing research in. For future work, we also want to measure texture bandwidth usage and depth buffer bandwidth usage since these are very likely to be reduced due to the traversal order that our new tile test enables. In addition, it would be interesting to investigate level-of-detail algorithms specifically for depth of field rasterization in order to improve performance. This could make the advantage of our algorithm even more pronounced.

Acknowledgements Thanks to Tobias Persson from BitSquid for letting us use the StoneGiant demo, and to Denis Shergin from Unigine for letting us use images from Heaven 2.0. Tomas Akenine-Möller is a Royal Swedish Academy of Sciences Research Fellow supported by a grant from the Knut and Alice Wallenberg Foundation. In addition, we acknowledge support from the Swedish Foundation for Strategic Research.

References

[AMA05] AKENINE-MÖLLER T., AILA T.: Conservative and Tiled Rasterization Using a Modified Triangle Set-Up. *Journal of Graphics Tools, 10*, 3 (2005), 1–8. 2

[AMMH07] AKENINE-MÖLLER T., MUNKBERG J., HASSELGREN J.: Stochastic Rasterization using Time-Continuous Triangles. In *Graphics Hardware* (2007), pp. 7–16. 1, 2

[BFH10] BRUNHAVER J., FATAHALIAN K., HANRAHAN P.: Hardware Implementation of Micropolygon Rasterization with Motion and Defocus Blur. In *High-Performance Graphics* (2010), pp. 1–9. 1, 2

[BFM10] BURNS C. A., FATAHALIAN K., MARK W. R.: A Lazy Object-Space Shading Architecture with Decoupled Sampling. In *High-Performance Graphics* (2010), pp. 19–28. 9, 12

[BHK*03] BARSKY B. A., HORN D. R., KLEIN S. A., PANG J. A., YU M.: Camera Models and Optical Systems Used in Computer Graphics: Part II, Image-Based Techniques. In *International Conference on Computational Science and its Applications: Part III* (2003), pp. 256–265. 1

[CCC87] COOK R. L., CARPENTER L., CATMULL E.: The Reyes Image Rendering Architecture. In *Computer Graphics (Proceedings of SIGGRAPH 87)* (1987), vol. 21, pp. 95–102. 1, 2

[CPC84] COOK R. L., PORTER T., CARPENTER L.: Distributed Ray Tracing. In *Computer Graphics (Proceedings of SIGGRAPH 84)* (1984), vol. 18, pp. 137–145. 1

[CT97] COORG S., TELLER S.: Real-Time Occlusion Culling for Models with Large Occluders. In *Symposium on Interactive 3D Graphics* (1997), pp. 83–90. 1, 2, 4, 7

[Dem04] DEMERS J.: Depth of Field: A Survey of Techniques. In *GPU Gems*, Fernando R., (Ed.). Addison-Wesley, 2004, pp. 375–390. 1

[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 *High-Performance Graphics* (2009), pp. 59–68. 1, 2, 3, 8, 9, 10, 12

[HG97] HAKURA Z. S., GUPTA A.: The Design and Analysis of a Cache Architecture for Texture Mapping. In *International Symposium on Computer Architecture* (1997), pp. 108–120. 1, 3

[KH01] KELLER A., HEIDRICH W.: Interleaved Sampling. In *12th Eurographics Workshop on Rendering Techniques* (2001), pp. 269–276. 2, 3, 9

[KLA04] KAUTZ J., LEHTINEN J., AILA T.: Hemispherical Rasterization for Self-Shadowing of Dynamic Objects. In *Eurographics Symposium on Rendering* (2004), pp. 179–184. 8

[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 (2011), 55:1–55:12. 11

[LAKL11] LAINE S., AILA T., KARRAS T., LEHTINEN J.: Clipless dual-space bounds for faster stochastic rasterization. *ACM Transaction on Graphics, 30*, 4 (2011), 106:1–106:6. 2, 12

[LES10] LEE S., EISEMANN E., SEIDEL H.-P.: Real-Time Lens Blur Effects and Focus Control. *ACM Transactions on Graphics, 29*, 4 (2010), 65:1–65:7. 2

[LK10] LAINE S., KARRAS T.: Two Methods for Fast Ray-Cast Ambient Occlusion. *Computer Graphics Forum (Eurographics Symposium on Rendering), 29*, 4 (2010), 1325–1333. 8

[MAM11] MUNKBERG J., AKENINE-MÖLLER T.: Backface Culling for Motion Blur and Depth of Field. *journal of graphics, gpu, and game tools, 15*, 2 (2011), 123–139. 5

[MCH*11] MUNKBERG J., CLARBERG P., HASSELGREN J., TOTH R., SUGIHARA M., AKENINE-MÖLLER T.: Hierarchical Stochastic Motion Blur Rasterization. In *High Performance Graphics* (2011), pp. 107–118. 1, 2, 3

[MESL10] MCGUIRE M., ENDERTON E., SHIRLEY P., LUEBKE D.: Real-Time Stochastic Rasterization on Conventional GPU Architectures. In *High Performance Graphics* (2010), pp. 173–182. 1, 2, 3, 12

[MM00] MCCORMACK J., MCNAMARA R.: Tiled Polygon Traversal using Half-Plane Edge Functions. In *Graphics Hardware* (2000), pp. 15–21. 2

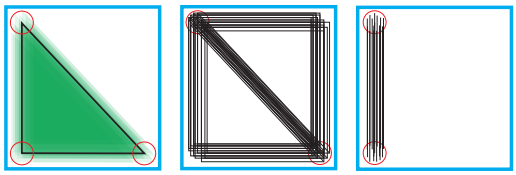


Figure 18: *A slightly defocused triangle projected on screen, with the circle of confusion of each triangle vertex highlighted in red. The middle illustration shows the bounding boxes of the diagonal edge for a set of lens coordinates. Each bounding box is large, and a cull test using the H_j line from this edge is likely to be beneficial. The bounding boxes of the vertical edge (right) are degenerate (i.e., lines) for each lens position. Here, culling using this edge is not beneficial.*

[NN85] NISHITA T., NAKAMAE E.: Continuous Tone Representation of Three-Dimensional Objects Taking Account of Shadows and Interreflection. In *Computer Graphics (Proceedings of SIGGRAPH 85)* (1985), vol. 19, pp. 23–30. 1, 2

[Pin88] PINEDA J.: A Parallel Algorithm for Polygon Rasterization. In *Computer Graphics (Proceedings of SIGGRAPH 88)* (1988), vol. 22, pp. 17–20. 8

[RKLC*11] RAGAN-KELLEY J., LEHTINEN J., CHEN J., DOGGETT M., DURAND F.: Decoupled Sampling for Graphics Pipelines. *ACM Transactions on Graphics, 30*, 3 (2011), 17:1–17:17. 9, 12, 14

[SAC*11] SHIRLEY P., AILA T., COHEN J., ENDERTON E., LAINE S., LUEBKE D., MCGUIRE M.: A Local Image Reconstruction Algorithm for Stochastic Rendering. In *Symposium on Interactive 3D Graphics and Games* (2011), pp. 9–13. 11

[TL08] TOTH R., LINDER E.: *Stochastic Depth of Field using Hardware Accelerated Rasterization*. Master's thesis, Lund University, 2008. 1, 2

Appendix A: Selective Tile-Edge Culling

In this section, we present a technique to estimate the efficiency of culling using the H_j lines. When the efficiency is expected to be low, we want to disable these tests on a per-edge basis. The motivation for this is illustrated in Figure 18.

To achieve our goal, we derive an estimate of the amount of samples that can be culled by a half-space line H_j derived from a tile corner and an edge, $E_{\mathbf{pq}}$, between vertices $\mathbf{p} = (p_x, p_y, p_z, p_w)$ and $\mathbf{q} = (q_x, q_y, q_z, q_w)$.

Depth-of-field is a shear in clip-space [RKLC*11], parameterized on the lens coordinates, (u,v), and can be seen as applying the matrix:

$$\mathbf{S}(u,v) = \begin{bmatrix} 1 & 0 & Au & Bu \\ 0 & 1 & Cv & Dv \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{6}$$

to the clip space vertex positions. A, B, C, and D are constants depending on the focus plane, aperture size, and the near and far plane.

A specific sample, (x,y,u,v), is culled by H_j if (x,y) is outside the edge $E_{\mathbf{pq}}$ sheared by $\mathbf{S}(u,v)$. If (x,y) is outside the bounding box of the triangle as seen from (u,v), it will be culled by the half-space lines h_i. Hence, it is only points within the defocused triangle bounding box that can be culled by H_j. Within this bounding box, the screen space area outside $E_{\mathbf{pq}}$ is:

$$A_{cull}(u,v) = \frac{|X_{\mathbf{q}}(u) - X_{\mathbf{p}}(u)||Y_{\mathbf{q}}(v) - Y_{\mathbf{p}}(v)|}{2}, \tag{7}$$

where $(X_{\mathbf{p}}, Y_{\mathbf{p}})$ are the projected pixel coordinates of vertex $\mathbf{p}$.

$$\begin{aligned} X_{\mathbf{p}}(u) &= \frac{p_x + (Ap_z + Bp_w)\,u}{p_w}\frac{R_X}{2} + \frac{R_X}{2} = X^0_{\mathbf{p}} + k_{\mathbf{p}}u, \\ Y_{\mathbf{p}}(v) &= \frac{p_y + (Cp_z + Dp_w)\,v}{p_w}\frac{R_Y}{2} + \frac{R_Y}{2} = Y^0_{\mathbf{p}} + m_{\mathbf{p}}v, \end{aligned} \tag{8}$$

where $R_X \times R_Y$ is the image resolution. The number of samples culled by $E_{\mathbf{pq}}$, assuming the samples are evenly spread over $(u,v) \in \Omega = [-1,1] \times [-1,1]$, and with sampling rate R_S, is thus:

$$N_{samples} = \frac{R_S \int_\Omega A_{cull}(u,v)\partial u \partial v}{\int_\Omega \partial u \partial v} = \frac{R_S}{4}\int_\Omega A_{cull}(u,v)\partial u \partial v. \tag{9}$$

To solve this integral, we take a closer look at A_{cull} in Equation 7, where the differences can be computed using Equation 8:

$$\begin{aligned} X_{\mathbf{q}}(u) - X_{\mathbf{p}}(u) &= X^0_{\mathbf{q}} - X^0_{\mathbf{p}} + (k_{\mathbf{q}} - k_{\mathbf{p}})u = a + bu, \\ Y_{\mathbf{q}}(v) - Y_{\mathbf{p}}(v) &= Y^0_{\mathbf{q}} - Y^0_{\mathbf{p}} + (m_{\mathbf{q}} - m_{\mathbf{p}})v = c + dv, \\ A_{cull}(u,v) &= \frac{|a+bu||c+dv|}{2}. \end{aligned} \tag{10}$$

The integral of this expression is:

$$\int_\Omega A_{cull}(u,v)\partial u \partial v = \frac{1}{2}F(a,b)F(c,d), \tag{11}$$

where:

$$F(a,b) = \begin{cases} 2|a| & \text{if } |a| \geq |b|, \\ \left|\frac{a^2}{b} + b\right| & \text{otherwise.} \end{cases} \tag{12}$$

With Equations 11–12, $N_{samples}$ can be computed using Equation 9.

The cost incurred by culling with $E_{\mathbf{pq}}$ is proportional to the number of tiles, N_{tiles}, covered by the bounding box of the defocused triangle. The oracle thus disables the tile edge test, H_j, for a particular triangle edge, $E_{\mathbf{pq}}$, if:

$$N_{samples}C_{sample} < N_{tiles}C_{tile}, \tag{13}$$

where C_{sample} is the sample test cost, and C_{tile} is the cost of testing one H_j line against a tile. The computation of A_{cull} can be simplified by assuming that the edge's bounding box does not become zero (i.e., degenerate) within Ω. This simplifies the expression for A_{cull} to:

$$A_{cull} = \frac{1}{2}|X^0_{\mathbf{q}} - X^0_{\mathbf{p}}||Y^0_{\mathbf{q}} - Y^0_{\mathbf{p}}|, \tag{14}$$

which means that $N_{samples}$ can be expressed as:

$$N_{samples} = \frac{R_S}{2}|X^0_{\mathbf{q}} - X^0_{\mathbf{p}}||Y^0_{\mathbf{q}} - Y^0_{\mathbf{p}}|. \tag{15}$$

This simplification results in a slightly underestimated culling potential in areas where the potential is already low, and works well in practice. The measurements in the paper are gathered using this simplification.

Note that the $N_{samples}$ computation presented in this section does not consider the discretization of the screen into tiles, nor does it consider the discretization of the lens into grid cells. These approximations may cause the algorithm to overestimate the real culling potential, and more refined oracle functions may give better results.