Multi-Layer Depth of Field Rendering with Tiled Splatting

Linus Franke linus.franke@fau.de
Nikolai Hofmann nikolai.hofmann@fau.de
Marc Stamminger marc.stamminger@fau.de
Kai Selgrad kai.selgrad@fau.de
Computer Graphics Group, University of Erlangen-Nuremberg

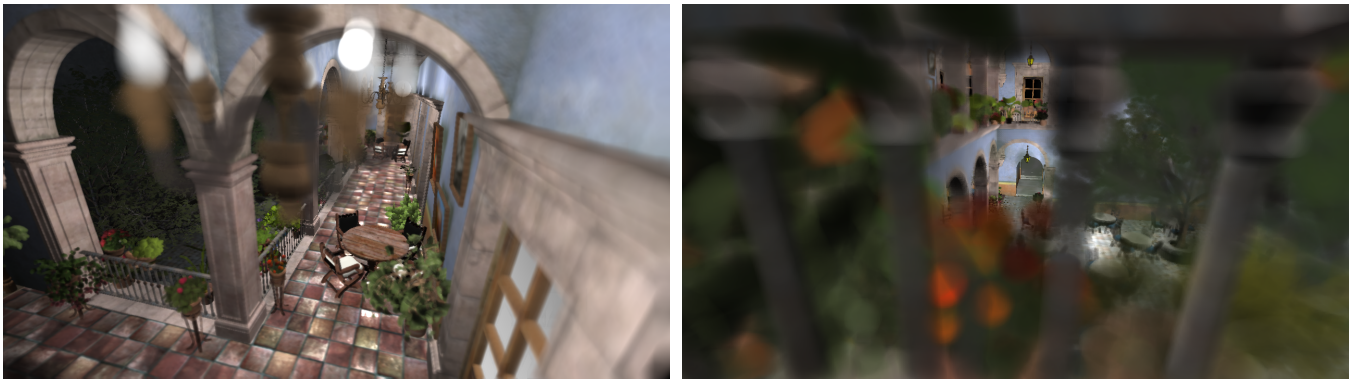

Figure 1: Our novel method to computing depth of field supports partial occlusion and produces consistent, noiseless bokeh (left, 41fps@720p) by working on screen-space tiles composed of partially multi-layered data. It also manages complicated and dense partial occlusion settings (right, 26fps@720p, both on GTX 1070).

ABSTRACT

In this paper we present a scattering-based method to compute high quality depth of field in real time. Relying on multiple layers of scene data, our method naturally supports settings with partial occlusion, an important effect that is often disregarded by real time approaches. Using well-founded layer-reduction techniques and efficient mapping to the GPU, our approach out-performs established approaches with a similar high-quality feature set.

Our proposed algorithm works by collecting a multi-layer image, which is then directly reduced to only keep hidden fragments close to discontinuities. Fragments are then further reduced by merging and then splatted to screen-space tiles. The per-tile information is then sorted and accumulated in order, yielding an overall approach that supports partial occlusion as well as properly ordered blending of the out-of-focus fragments.

Multi-Layer Depth of Field Rendering with Tiled Splatting, Author's Version
Final version to be published in PACM CGIT
https://doi.org/10.1145/3203200

1 INTRODUCTION

Defocus blur, originating from lenses with finite extend (such as camera lenses, or the human eye) is a common effect seen in video games and movies. Very fast algorithms for computing approximations of it are well established [Riguer et al. 2003; Sousa 2013; White and Barré-Brisebois 2011; Zhou et al. 2007]. Even though some common approximations exhibit systematic error [Demers 2004; Hammon 2007], the forgiving nature of this effect and decades of research make for plausible results. Still, we believe that efficient, more robust methods are welcomed by implementors, especially in the domain of high-quality imaging. Targeting this domain, we contribute a novel, image-based approach to rendering defocus blur that shows excellent quality, even for very hard geometric cases, and that outperforms established methods supporting a similarly wide feature set.

Our method relies on multiple layers of input data in areas where those layers will add further information to faithfully represent geometry hidden by out-of-focus near-field objects. To this end we compute a partial multi-layer image, the fragments of which we then further thin-out and merge (based on their distance to surrounding geometry and the size of the circle of confusion) to yield a manageable data set. Based on this we sort the collected fragments into screen-space tiles and apply fast, coherent accumulation.

Our work draws on recent advances in graphics research, namely tiled shading [Harada et al. 2012; Olsson and Assarsson 2011] applied to compute depth of field [Selgrad et al. 2016] with correct fragment-ordering, efficient, temporal depth peeling [Mara et al. 2016], discontinuity-based filtering [Segovia et al. 2006; Simmons and Séquin 2000; Widmer et al. 2016] with detection of areas of disocclusion [Widmer et al. 2016] and efficient, domain-specific, reduction of multi-layer data [Lee et al. 2010].

The specific contribution of our work is the integration of all those aspects to form a novel, high-quality approach to compute defocus blur, as well as the optimizations originating from our depth-of-field use-case such as heavily reducing layering during rendering and by post-processing of the multi-layer data.

Following a short survey of related work given in Section 2, we give a high-level overview of our algorithm in Section 3. We then describe how we generate our partial multi-layer image in Section 4 and how it is processed and traversed to compute images with depth of field in Section 5. These sections describe our algorithm's basic workings, from which we detail how to further reduce the amount of data that has to be processed in Section 6. In Section 7 we evaluate our method in terms of image quality as well as render time. We conclude our presentation with Section 8.

2 RELATED WORK

Depth of field is an effect originating from physical properties of lenses such as in cameras or the eye. There is a huge body of work in simulating it, both in offline and real-time contexts. In offline rendering, it can easily be incorporated into path-tracing based systems as an additional integration over the lens [Cook et al. 1984]. An alternative is to simply average many separate renderings with slightly shifted camera positions [Haeberli and Akeley 1990].

Initial work on rendering depth of field proposed the thin-lens model [Potmesil and Chakravarty 1981]. A thin lens with a given aperture causes points of a certain distance from the lens to be in focus and points further away from this focus distance get progressively more blurry. The amount of blur is given by the size of the circle of confusion (Coc), see Figure 2 for a visualization of the thin lens and the extent of the Coc.

For real-time rendering there are two principal post-processing techniques to simulating defocus blur: *Scattering* depth of field, first presented by Potmesil and Chakravarty [1981] uses a pinhole-based image and renders pixels as semi-transparent splats, creating the blur. For correct accumulation, the splats need to be sorted by depth and accumulated, which can be challenging with regard to performance [Demers 2004; Křivánek et al. 2003]. This method is the basis for rendering defocus blur in point based rendering systems [Křivánek et al. 2003], on per-pixel layers [Lee et al. 2008] and in tiled pipelines [Selgrad et al. 2016], each reducing workload to achieve better performance by either reducing the complexity of the global sorting problem to discrete layers or introducing small errors in the image to simplify computations.

Gathering approaches also start with a pinhole image, but then gather information about neighboring pixels in the area of the current pixel's Coc. Such approaches are better suited for modern graphics hardware [Demers 2004; Riguer et al. 2003; Scheuermann and Tatarchuk 2004; Sousa 2013; White and Barré-Brisebois 2011]. Naively filtering this area, however, exhibits strong artifacts, such as intensity leakage (color of in-focus pixels bleeding into the far-field) and depth discontinuities at edges (near-field blur having sharp edges at discontinuities to in-focus areas) [Demers 2004; Hammon 2007] . With more sophisticated filters, e.g. considering the depths of the pixels contributing to the blur, these artifacts can be reduced [Zhou et al. 2007], but more nuanced artifacts in the form of incorrect color accumulation (originating from the lack of correct depth ordering) can still be observed [Selgrad et al. 2016].

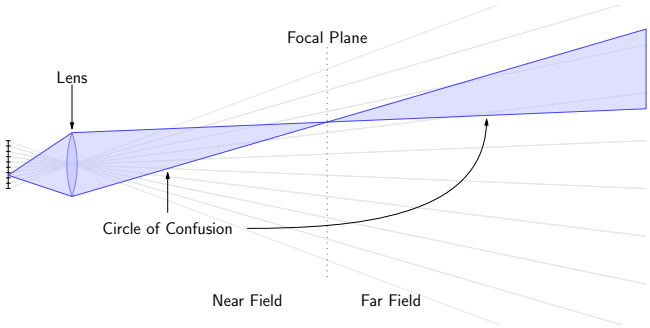

Figure 2: The size of the circle of confusion (Coc) goes to zero at the focal plane and rises with distance from it. As can be seen, many pixel footprints can overlap the Coc of a single pixel (especially in the near field).

Recently, methods that combine the two principal techniques, have emerged [Harada et al. 2012; Jimenez 2014; Olsson and Assarsson 2011]. By gathering in a large enough area and then splatting the gathered pixels on this reduced scale, the workload of global sorting from scattering techniques is heavily reduced and amenable to parallelization. Further, reducing sorting complexity can be achieved [Jimenez 2014] (based on depth-partitioning [Lee et al. 2008]) by separating near and far field in this technique. Note that recent, fast gather and scatter-as-gather techniques mostly rely on statistical sampling to reduce workload, which usually needs additional work to reduce artifacts based on sampling variance [Jimenez 2014; Sousa 2013; White and Barré-Brisebois 2011].

Most of the post processing methods mentioned so far suffer from not being able to properly simulate partial occlusion. This effect is caused by fragments that are not visible from a pinhole camera image but can become visible due to objects in front of them becoming blurry and transparent. Schedl and Wimmer [2012] present one way to simulate this effect by subdividing the scene into depth layers and then filtering them individually. Mei et al. [2005] use a camera model with distorted rays that also samples parts of the scene that are not visible from a pinhole view. Approaches that base their input on multi-layer images instead of single-layer pinhole images are also able to circumvent this artifact. Lee et al.'s algorithms [2009; 2010] use depth-peeling to collect the missing information and represent layers as height fields to efficiently trace rays, which, assuming a sufficient amount of samples are taken, yields excellent results, also for strong near-field blur. Selgrad et al. [2015] used per-pixel arrays for collection, progressive filtering and alpha blending to simulate large and near-field blurs.

Multi-layer images can be acquired in a number of different ways. Depth-peeling [Everitt 2001] uses multiple render passes to generated successive layers that can become visible by partial occlusion. The information is implicitly sorted by depth, but using multiple render calls can be costly. Methods to speed up the approach by exploiting state of the art GPU features [Liu et al. 2009] or temporal reprojection [Mara et al. 2016] achieve higher performance results. Another technique collects all fragments of the scene in one pass in per-pixel linked lists [Yang et al. 2010] or arrays [Selgrad et al. 2015]. This technique can be very fast compared to depth-peeling [Hofmann et al. 2017], but the a-priori unbounded memory requirements and unsorted results pose challenges.

3 ALGORITHM OVERVIEW

Our algorithm executes a number of different stages that we group by the following phases:

Generation The first phase is comprised of two steps, the first of which is the rendering of a multi-layer image to support partial occlusion. In the second step we limit the multi-layer data to only keep hidden fragments along depth discontinuities, as this is where partial occlusion can be observed.

Reduction The second phase of our method further decimates the multi-layer data by combining fragments from neighboring pixels if they are out of focus and at similar depth values and shaded color. The result of this step is a list of fragments (of various sizes) per 4×4 pixel region.

Tiling The next phase then bins these fragments into larger screen-space tiles (usually 16×16). Due to the highly heterogeneous lengths of the input lists we apply load-balancing to fully utilize the GPU.

Accumulation In the final phase we compute blurred output-image based on our tiled structure. To this end we first sort the fragments stored in each tile. This allows us to take the actual order of the fragments into consideration (and not indiscriminately apply uniform weights) when traversing the per-tile lists for each pixel to determine the final output color.

The following sections describe these phases in more detail, starting with data structure construction (Section 4) and use (Section 5). These two sections cover our basic pipeline in detail, but without the reduction phase. We then describe how to further reduce the number of fragments to obtain manageable per-tile lists (Section 6) before evaluating our results and comparing them to competing approaches (Section 7).

4 PARTIAL MULTI-LAYER GENERATION

A key aspect of our method is proper handling of partial occlusion (see Section 2). Partial occlusion describes that fragments that are not visible from a pinhole camera can become visible due to objects in front of them becoming blurry. To support this, we keep multiple layers of the scene in order to provide the information becoming relevant due to such disocclusions.

4.1 Depth Peeling with Depth of Field

To generate our partial multi-layer structure, we follow Mara et al. [2016]. Their method works by multiplying the scene's primitives in the geometry shader and emitting the same primitive to a (fixed) number of layers. For each layer the primitive's z-values are compared to the reprojected depth of the earlier layer in the previous frame. Thus, the entire depth peeling step is computed with a single render-pass by re-using previous frames. As demonstrated in earlier work [Hofmann et al. 2017; Mara et al. 2016; Widmer et al. 2015], this temporal construction works well in practice.

As observed by Mara et al. [2016], collecting exactly successive scene layers (as with traditional depth peeling [Everitt 2001]) requires many layers to add significant information in dense regions. Therefore they propose to only keep fragments during depth peeling that have a certain distance to the previous layer. This distance parameter, even though quite robust, is scene dependent. In the case of depth-peeling for depth of field we can rely on Lee et al.'s [2010] insights. They showed that using the lens and a fragment at a given distance, there is a region behind that fragment which cannot be reached by lens-rays. This can be understood more intuitively when considering the lens as an area light source and the fragment as an occluder. In this setup, fragments in the umbra-region are mostly shadowed and can thus be disregarded without introducing noticeable error [Lee et al. 2010]. We use this observation as a heuristic to optimize the per-fragment minimum separation for temporal depth peeling [Mara et al. 2016]. In the remainder of this paper we will refer to this threshold as *umbra threshold*. Note that this threshold depends on a fragment's z-value, i.e. it is not a constant. Figure 3 illustrates this per-fragment threshold.

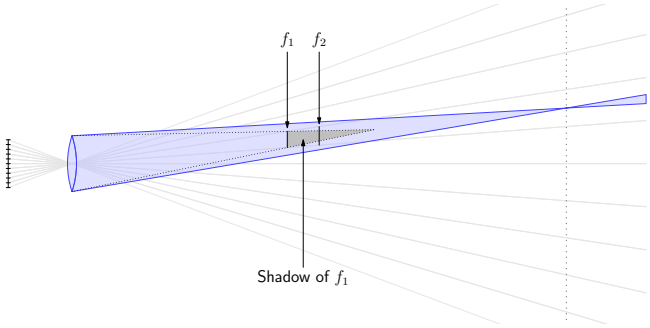

Figure 3: Umbra thresholding: Fragment f_1 blocks most of the lens rays on f_2.

Thus, the combination of Lee et al.'s [2010] observations with Mara et al.'s [2016] temporal depth peeling can be considered the state of the art in depth peeling for our specific use-case. It is both efficient in terms of run time and effective in producing small sets of fragments per pixel. Its effect on image quality will be evaluated in Section 6 where we also propose a more aggressive extension.

4.2 Where to keep Multiple Layers

To support partial occlusion for near-field out-of-focus objects in the general case, a number of layers has to be kept. Only keeping two layers will work in synthetic cases, but not in general. However, we note that disocclusion only affects object-silhouettes [Mei et al. 2005] (extended by the blur factor) and incorporate this observation into our pipeline to reduce the number fragments, see Figure 4. In the umbra-terminology introduced above, areas of disocclusion are the penumbra-region of fragments closer to the camera.

To determine for which pixel-locations multi-layer data should be stored we run a full-screen pass on the first layer computed in the previous step, resulting in a disocclusion buffer (in the spirit of Widmer et al. [2016]). To this end we start at all pixels that have discontinuous edges, i.e. pixels that have neighbors with significantly differing depths. We then tag all neighboring pixels in the respective circle of confusion (Coc) as requiring multi-layer data. This naturally grows a border into near-field objects which will become partially transparent by this. Pixels further from a depth-discontinuity than the radius of the edge's Coc cannot be part of disocclusion, the defocus blur only affects the front-most part then.

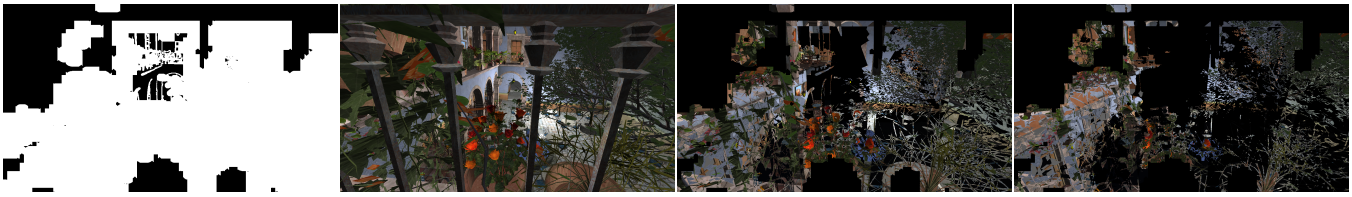

Figure 5: Disocclusion buffer (left) and the first three partial multi-layer images used to compute the image shown in Figure 1, right. A breakdown of the render-times for the different steps of our algorithm for this view is displayed in Figure 13 (Section 7).

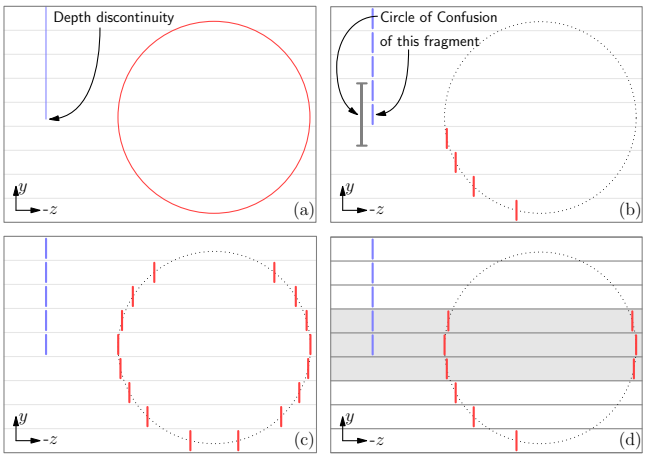

Figure 4: (a) Side-view of a blue plane (slightly out of focus) in front of a red sphere (in focus). (b) Circle of confusion (Coc) for a pixel on the plane, all fragments that would be rasterized for a standard post processing effect are displayed. (c) All fragments of the scene, keeping them allows to compute partial occlusion. (d) Only keeping multiple layers of fragments in the Coc of pixels at discontinuities greatly reduced the number of fragments collected. Note how the discontinuity (bottom-end of the plane) is dilated by the Coc.

Illustrating this scheme, Figure 4 (a) shows a side-view of a large blue plane in front of a red sphere. The blue plane is slightly out of focus and thus a small Coc arises. Figure 4 (b) shows the fragments generated by standard rasterization, i.e. only the front-most layer. It also indicates the size of the Coc for one pixel on the blue plane. To properly manage partial occlusion, earlier work relies on a complete multi-layer image (c). For the discontinuity detected at the pixel referenced in (d), however, only fragments in the footprint of its Coc have to be collected. Figure 4 (d) highlights the pixels for which this is the case. Note how the area requiring multi-layer data extends upwards into the blue plane (and also below it). Thus, fragments from the red sphere are only kept when they are the closest ones to the viewer (standard z-test, lower three rows) or when they are in the area tagged for collecting multi-layer data (shaded gray).

Figure 5 shows the disocclusion buffer of the scene shown in Figure 1 (right). As can be seen, this is a challenging case where multi-layer data is required for most of the pixels. The figure further shows the first three (out of five) layers generated to support partial occlusion.

4.3 Dataflow

We incorporate this process into our temporal pipeline such that we use the thusly computed multi-layer mask (disocclusion buffer) in the next frame, i.e. each frame uses the re-projected mask of the previous one. However, our algorithm can also be run with two render passes, one to compute the first layer, and one to collect the other layers after the mask has been computed. We have opted for the former choice for performance reasons, but choosing the latter would work as well.

The layered render target used to compute the partial multi-layer image is then converted to per-pixel arrays holding the layers, in order. Note that these arrays are allocated to hold the maximal number of fragments that can be stored, and thus there will be many instances for which the array is not completely filled. Using a more compact representation is not beneficial here as the allocated storage will be re-used by later sorting and merging stages. Note, however, that the number of layers is fixed and known beforehand, i.e. we usually render four or five layers and thus can preallocate without fear of unbounded storage requirements (such as with per-pixel linked lists).

5 TILING AND ACCUMULATION

Based on the partial multi-layer data computed in the first phase, proper defocus blur can be computed by splatting the contribution of all collected fragments (in-order) to all neighboring pixel locations within their circle of confusion. However, naively doing so will yield poor performance and is problematic regarding the correct ordering of fragments during splatting. Therefore, we align our splatting with a depth of field variant [Selgrad et al. 2016] of tiled splatting [Harada et al. 2012; Olsson and Assarsson 2011], which is very similar to tiled particle accumulation [Thomas 2014]. In contrast to tiled depth of field splatting [Selgrad et al. 2016], our multi-layer setting shows longer lists with highly incoherent lengths, especially when also allowing for larger blur circles (splatting into 2nd-degree neighbors).

In the following we will describe how the tile lists are constructed from the depth peeled image (Section 5.1), sorted (Section 5.2) and efficiently traversed with proper blending (Section 5.3).

5.1 Tiling Inhomogeneous Lists

To facilitate efficient computation of the per-pixel blur, we partition the pixels of the framebuffer into tiles (usually 16×16 pixels). These tiles are populated by all the fragments within their screen-space footprint. Furthermore, for each pixel the eight (default choice) surrounding tiles are checked for overlap with the circle of confusion

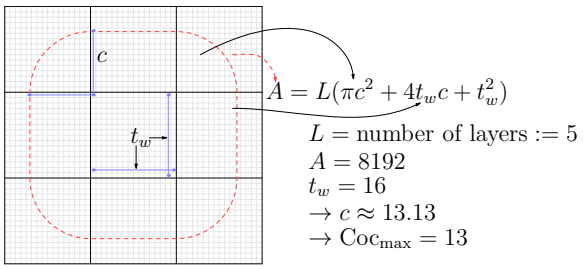

Figure 6: Limitations on the circle of confusion arising from shared-memory bitonic sorting (on current hardware tile-lists can not exceed 8192 fragments). Note that this limit can be relaxed by sorting tiles exceeding it in global memory.

of this pixel's fragments. All of those tiles that the fragment's Coc overlaps will receive a copy of that fragment. The result of this is a multiplication of fragments of up to a factor of 9 (see Section 7 for more realistic list sizes) such that each tile has a copy of all the fragments that can affect the color of one of its encompassed pixels.

The input of this process is the set of per-pixel arrays holding the partial multi-layer data generated in the first phase. The output is a set of per-tile arrays containing all the relevant fragments for each tile. This is easily implemented by conservatively estimating the number of fragments that will be inserted into each tile by the sum of fragments stored in the tile's direct neighbors (or 2nd-degree neighbors to support lager blur). Filling the arrays is then simply done by maintaining an atomic counter.

As we keep a partial multi-layer image, it can be expected that the lengths of the lists that are processed by the same warp in our GPU implementation vary dramatically. To distribute the workload more evenly over the individual threads we establish a work-queue similar to dynamic fetching in ray traversal [Aila and Laine 2009]. To efficiently facilitate this, we compute a indexing structure over all the pixels of each tile, prior to splatting. This structure simply compensates for the fact that not all per-pixel arrays are filled to the maximum allocated limit (see Section 4.2) and provides contiguous indexing over the tile's pixels. It can be easily computed by a per-tile scan of the list lengths and adding one index entry for each fragment. During splatting this data structure is then used such that each thread allocates a set of fragments (usually four) at a time and splats them to the surrounding tiles.

Note that adding pixels only to adjacent tiles effectively limits the maximum Coc that can be represented with this structure. For 16×16 tile lists its radius can not exceed 16 pixels (conservative from the tile-borders, but see also below). Larger tile sizes are possible, but our evaluation showed that best performance is achieved using 16×16 tiles. However, the restriction of splatting to the directly neighboring tiles, only, can be relaxed and thus larger blur is possible, see Section 7.

5.2 Sorting Tile Lists

The thusly constructed per-tile lists are then sorted by depth. This allows us to use alpha blending when computing the blurred output image in the next stage, which results in a higher-quality simulation of depth of field than when disregarding ordering [Selgrad et al. 2016] (even in the absence of multi-layer data). As sorting can be very costly for large tile-lists (see Section 7), we implemented bitonic sorting [Batcher 1968] that works entirely in shared memory. In the following we describe the implications of using this highly efficient alternative.

Fast Sorting. Our shared-memory bitonic sort is very fast, but requiring all data to be present in shared memory entails a limit on the number of fragments that can be processed. On current-generation GPUs there is only 48 KiB of shared memory available per block [Nvidia 2017]. We can fully exploit this memory when sorting 32-bit depth-values together with 16-bit indices. As bitonic sorting requires power of two sized working sets, we can sort up to 8192 fragments. Using more aggressive compression, e.g. 19 bits for depth and 13 bits for the index, increases performance by allowing 3 instead of 2 blocks to run on the same shared multiprocessor [Nvidia 2017] (reducing execution time for this step by around 35%). However, temporal stability can suffer from this as z-fighting can occur as a result of the reduced depth-precision.

Going a less aggressive route, we noticed that in most cases many tile lists are not even close to be filled up to this maximum. Therefore, we group the lists by upper bounds on their size, in our implementation the bins are the powers of two from 256 up to 8192. This way many more smaller tiles can be sorted in parallel by reducing the shared memory restrictions. This reduces sorting time by around 10% in our test scenes. Allowing a larger blur radius for those would also be possible, but yield inconsistent results depending on the tiles' depth complexities.

Figure 6 shows the splatting setup for a tile with its neighborhood. To ensure consistent splat sizes the circle of confusion of all fragments in a tile is capped to the maximum one valid across all tile-pixels. For example, for one of the center pixels of the tile the Coc could go up to 23 without exceeding the one-ring neighborhood. However, the same distance will exceed the neighborhood when closer towards the tile-border. Therefore, the maximum radius of the Coc is set to the tile size. Depending on the number of layers collected during depth peeling, the radius might further get reduced by the requirement to sort entirely in shared memory. The red border in Figure 6 shows the geometry of the tile dilated by its maximum Coc. In the right part it shows how to compute the number of pixels encompassed by this region. It can be seen that with enough shared memory for 8192 elements to sort, and 5 layers (worst case: fully populated) of fragments per pixel, the maximum Coc-radius is only 13. Following the same argument for 4 depth layers results in a maximum radius of 15, which is almost as large as possible under the neighborhood constraints of tiled splatting as described above.

Tile Sizes and Coc Radii. The restrictions described so far can be relaxed by not strictly sticking to worst-case scenarios. In practice, many tile lists contain only a fraction of the allowed entries and thus sorting with differently configured shared memory partitions (to allow sorting more lists in parallel) is possible. For tile lists that exceed the limit, we extend this grouping scheme to also collect tiles that have overly long lists and sort these in global memory. As can be expected, this slows down rendering but prevents errors from not sorting the full lists. Our tests showed that sorting 2% of the tile lists in global memory (i.e. for the longest 72 tile lists in Figure 1, right) sorting time increases by 10%. As shown in Section 7, even for

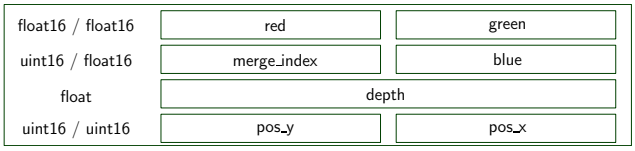

Figure 7: Illustration of the data kept for each fragment. Beside HDR color and depth information, we keep the screen-space position to allow fractional positions after merging and merge-index to ensure consistent merging.

very dense cases of partial occlusion (such as displayed in Figure 1, right), our method does not have to rely on global sorting at all (also because of fragment reduction, see Section 6). We can even extend our scheme to produce large blurs by splatting into 2nd-degree neighbors, yielding a Coc-radius of up to 32. Section 7 shows that render-times for even larger blur sizes increase moderately.

5.3 List Traversal

In the final stage of our algorithm we accumulate the fragments stored in a tile (now including the relevant fragments from neighboring tiles) straightforwardly using alpha blending [Selgrad et al. 2016; Thomas 2014]. Therefore, we traverse each tile with a block of 16×16 threads. The fragments of each warp-sized sub-tile (usually 16×2) are processed completely coherently, new data is loaded from the per-tile arrays and stored to shared memory in chunks of 32 elements. Fragment contributions are accumulated for a pixel if the current fragment's circle of confusion overlaps this particular pixel. If so, the contributions are weighted with front-to-back alpha blending, and upon saturation the respective threads are only active for loading. Once all pixels of a sub-tile are saturated traversal itself is terminated.

6 FRAGMENT REDUCTION

From the description of our algorithm in Sections 4 and 5 it is clear that the number of fragments collected in the first phase has a dramatic effect on performance. Even more so, when it can be guaranteed that no more than a certain number of fragments will be collected per pixel, this can even allow larger blur (see Section 5.2). Therefore we only accept new fragments during depth peeling that have a minimum distance to earlier fragments [Mara et al. 2016]. This distance is driven by Lee et al.'s [2010] umbra threshold, which allows us to retain a blur radius of 13 or 15 pixels, while still rendering images of high quality with partial occlusion.

The remainder of this section provides details on the second phase of our algorithm, reduction (Section 6.1), as well as on umbra merging used in the first two phases (Section 6.2).

6.1 Fragment Merging

The previous sections described our overall algorithm (without merging of similar fragments in the multi-layer image) to give a simplified overview of our pipeline. To improve run time, fragment merging is applied after the partial multi-layer image has been generated, and prior to tiling. Therefore, similar to the tiling phase, the input of this phase is a set of per-pixel arrays resulting from depth peeling. Each array consists of up to a maximum of L layers, usually four or five, where the array's allocated size is the maximum length allowed (follows from depth peeling).

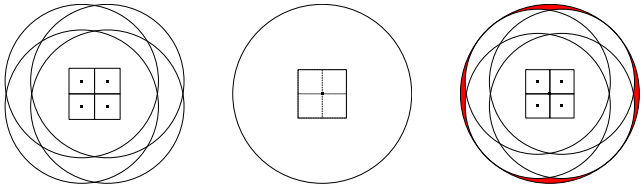

Figure 8: Four fragments that will be merged (left). Resulting fragment with new position and Coc-radius (center). Note that there is some overestimation where the individual fragments' Cocs do not overlap (right, shown in red).

We hierarchically merge fragments in a two step process, each step collapsing a 2×2 region of pixels, so that after merging the image size is reduced to one fourth in each direction. During this step, we always keep the front-most fragments from each pixel of this 2×2 region and compare their depths. If the depth values are similar we also check for similarity in shaded color. When the colors are also similar and the fragments are not very close to the focal plane, then they are combined and emitted as a single, larger fragment. Figure 7 illustrates the data-layout of a fragment. It shows that we also keep a (screen-space) position, which is with respect to the full-resolution image. This is required for determining if a fragment (with its circle of confusion) overlaps with a given pixel in the final accumulation step (see Section 5.3). The position is changed when fragments are merged to move the center of the new fragment to the center of the merge-region. Note that we only merge when *all* four fragments for the 2×2 pixel range match (i.e. we do not generate transparent fragments). We then set the resulting fragment's Coc to the union of all the individual fragments' Cocs relative to this new position. Fragments that are not merged are still stored in the same output-lists, but will not be considered for merging in the next step and their position and Coc are kept as is. To ensure that they are not merged by the next merging step, their merge-id is also encoded.

Figure 8 illustrates the input and output of a single fragment-merging operation. The four fragments to the left have similar depth values, and consequently similar Coc-radii. The merged fragment (center) is assigned a Coc-radius (r) that encompasses all the fragments' Cocs (with radius r_i) and its color is the average color of the fragments. This step incurs overestimation (right) but it is usually minor due to the similar Coc-radii. A closer approximation could be attained when taking an average that takes the circular nature into account, e.g. $r = \sqrt{r_1{}^2 + r_2{}^2 + r_3{}^2 + r_4{}^2}$, but since we only store depth values (and reconstruct a fragment's Coc on demand), merging to the largest circle proved to be more robust in practice.

Note that, in contrast to multi-layer filtering [Selgrad et al. 2015], simply averaging the colors is generally valid as we never merge fragments originating from within the same pixel. Similar depths inside a single pixel are even ruled out by umbra thresholding (on the depth peeled level, see also Section 6.2). Regarding multi-layer filtering, note that we do not compute as many levels and, more importantly, do not suffer from approximation errors as a result of overly aggressive down-filtering.

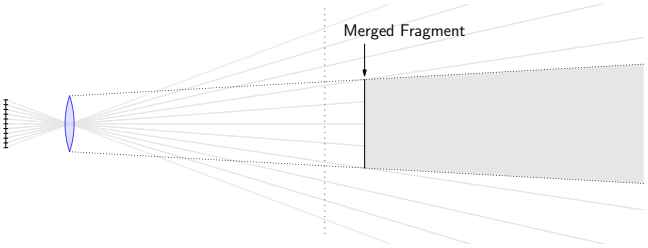

Figure 9: Umbra aware merging: When a merged fragment's size exceeds the lens aperture the shadow will diverge, all fragments behind it will be shadowed.

6.2 Umbra Aware Merging

Umbra thresholding as applied during depth-peeling to skip over fragments that can only be reached by a very small fraction of the lens rays can also be applied during merging. To this end, we always keep the last-merged fragment, compute its umbra-length and skip successive fragments for the input lists as long as they are still considered to be in shadow. This is especially effective in reducing list lengths as the larger fragments, naturally, have larger umbra-extent. This umbra aware merging is thus applied when merging fragments that will have a footprint of 2×2 as well as when generating fragments of 4×4. Especially in the latter case, there is a chance that far-field fragments will become larger than the lens' aperture and thus cast an infinite shadow, effectively terminating the merging-process and thus trimming lists even more strongly (Figure 9 shows an accentuated example).

The reason for this is easily seen by an example. Consider a camera of 35mm focal length at $f/11$. Then the aperture is 3.18mm (similar to the human eye's [Winn et al. 1994]) and the field of view is 54.4 degrees [Claff 2016]. With this, the size of a fragment at 720p matches the aperture at 15.8 meters, a fragment that was merged from 4×4 fragments reaches the aperture's size already at 3.9 meters. For Figure 1 (left) the aperture is 0.4 units, for Figure 1 (right) it is 1.6, at 720p and fovy of 70 degrees the fragment sizes catch up at 206 and 822 units, respectively (for a scene extent of 5000 units).

7 EVALUATION

In this section we show how our method performs on real-world use cases, describe the impact of certain parameter choices and compare our results with competing approaches. We will first compare our render times to classical, post-processing based approaches [Riguer et al. 2003; Selgrad et al. 2016] and an established approach that uses screen-space ray traversal and that supports partial occlusion [Lee et al. 2009, 2010]. We then describe the results of parameter choices and optimizations of our method and finally evaluate the quality of the images computed the our approach and the aforementioned methods.

Performance. Unless otherwise mentioned, our method is configured to compute five layers using temporal depth peeling, uses 16×16 screen-space tiles and splats at most into 2nd degree neighboring tiles (i.e. the Coc is limited to radius 32 at most). Sorting is allowed to fall back to global memory for tiles with more than 8192 elements. All screenshots presented are rendered with this configuration as well (at 720p on a GTX 1070).

Method	Render	Merge	Splat	Sort	Apply	Overall
Figure 1, left						
Gathering	7.09				2.09	9.18
Tiled Splatting	7.09		0.88	1.80	5.22	14.99
RT, 16sppx	13.76				13.65	27.41
RT, 32sppx	13.76				28.82	42.58
RT, 64sppx	13.76				58.90	72.66
Ours	13.21	1.19	1.42	3.79	4.52	24.13
Figure 10						
Gathering	0.80				3.80	4.60
Tiled Splatting	0.80		1.16	2.08	8.56	12.60
RT, 16sppx	2.95				12.20	15.15
RT, 32sppx	2.95				25.27	28.22
RT, 64sppx	2.95				51.37	54.32
Ours	2.71	1.09	1.07	2.66	3.34	10.87
Figure 11						
Gathering	0.14				2.60	2.74
Tiled Splatting	0.14		0.91	2.16	5.89	9.10
RT, 16sppx	0.28				2.00	2.28
RT, 32sppx	0.28				3.95	4.23
RT, 64sppx	0.28				7.91	8.19
Ours	0.39	0.53	0.62	0.87	1.01	3.42

Table 1: Times (in ms) for two single-layer post-processing methods, gathering [Riguer et al. 2003] and tiled splatting [Selgrad et al. 2016], and three settings of Lee et al.'s [2010] lens ray tracing compared to our method (for three scenes as indicated in the table). The apply column refers to the tracing times for the ray tracing implementations.

Figure 10: A view into Sponza that shows partial occlusion and high depth complexity (along the columns). See Table 1 (middle) for detailed render times and Figure 12 (bottom) for information on tile list lengths.

Table 1 lists detailed render times for a high-performance gathering approach [Riguer et al. 2003], single layer tiled splatting [Selgrad et al. 2016] and for high-quality depth-of-field ray tracing [Lee et al. 2010] that also supports partial occlusion. As can be expected, the single-layer methods are significantly faster than multi-layer bases approaches, however, they are not able to produce plausible blur in partial occlusion cases, as will be illustrated below. To ensure a fair comparison we use our temporal depth peeling with Lee et

Figure 11: A very simple setup consisting of a white plane in the far field, a red sphere in focus and a small blue quad in the near field. (Right) The Disocclusion-buffer illustrating the simplicity: Only the white areas have more than one layer. See Table 1 (bottom) for detailed render times.

al.'s [2010] method, too (using five layers, N-buffers with $1/4^{th}$ resolution and jittered tracing). Our rendering times (first column in Table 1) are still a little lower in the non-trivial scenes, as our method also uses the disocclusion-buffer and consequently shades fewer fragments. Disocclusion-buffer generation is also contained in the 'render' column. Overall, our method performs as well as ray tracing 16 samples per pixel. Larger sample counts (as required for noise-free images, see below) take much more time than using our method. With a very simple setup, as seen in Figure 11, our method's performance is much closer to ray tracing with 32 samples. This is because Lee et al.'s [2010] method benefits from a scene's low depth range, as it uses a N-buffer for reducing ray tracing times. In extreme cases, e.g. of a scene containing large depth differences in the near field and in close screen-space proximity, such as with the railing and the courtyard in Figure 1 (right), the N-buffer does not prove to be very effective, and consequently its utility diminishes in such cases.

Parameters. One of the most influential parameters regarding both performance and possible blur-size is the maximum radius of the Coc that is allowed. In Section 5.2 we showed that, under worst-case assumptions, the radius is limited to 13 pixels. However, the limit on the tile-list lengths is usually not reached, even for setups with very large areas of partial occlusion and high depth-complexity. Figure 12 shows two histograms, one for the image shown in Figure 1 (right), a challenging setup, and one for the simpler setup in Sponza, displayed in Figure 10. This suggests that larger Coc-values are indeed possible, and even if a few individual tiles exceed the limit, they can still be sorted in global memory without significant impact on run time (see Section 5.2 and Figure 12). Figure 13 exemplifies this by showing render times for the image displayed in Figure 1, right. It shows that much larger blur sizes are possible (global sorting was not necessary in any case) and how this impacts performance. The two noticeable steps from radius 15 to 17 and from 31 to 33 are where we have to start splatting into the 2-ring and 3-ring, respectively, with our default 16×16 tiles. Note that the time to generate the multi-layer image also increases with the maximum Coc. This is because the larger blur is reflected in the disocclusion buffer and thus causes more pixels to collect multi-layer data in the first place.

The shared-memory sorting in the first column of Figure 13 is 4.3 ms. When we instead sort in global memory the sorting time increases to 18.9 ms, making the whole algorithm impractical. Thus we determine that sorting in shared memory is essential. As the evaluation of different blur sizes and tile-list lengths (see Figures 13

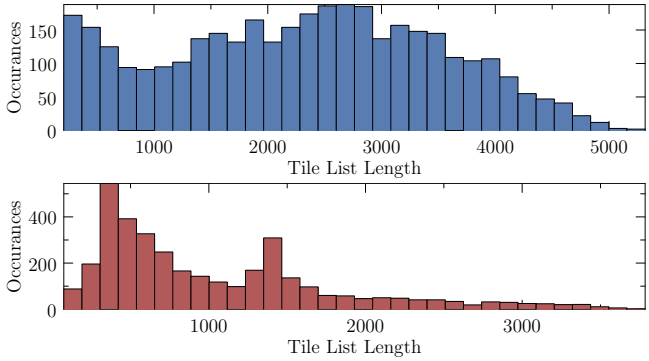

Figure 12: Histogram of the tile-list lengths for the scenes shown in Figure 1, right (top), and Figure 10 (bottom). Note that even the former, a very hard case with very strong partial occlusion and layering (see also Figure 5), does not exceed the shared-memory sorting limit.

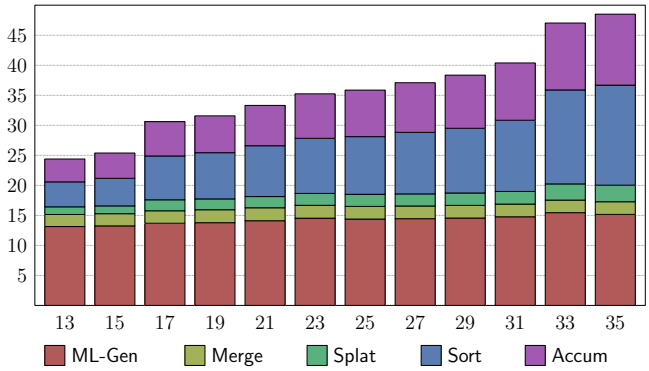

Figure 13: Different upper bounds on the circle of confusion result in longer tile lists and consequently decreasing performance. Render-times (in ms) for the scenes shown in Figure 1, right (there with $r = 29$): a very challenging setup with considerable partial occlusion. Figure 5 shows the disocclusion buffer (left) and three of the five layers used to compute the final image. Computation is done with our algorithm's standard configuration.

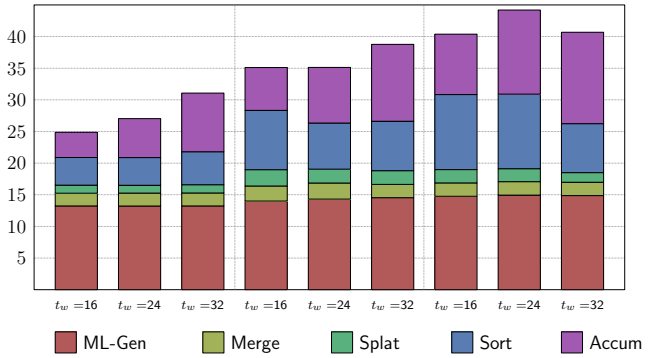

Figure 14: Run time (in ms) for using different tile-sizes ($t_w \times t_w$) on the scene shown in Figure 1, right. The left part shows times for Coc radius $r = 15$, the middle for $r = 23$, the right part for $r = 31$. Note that $t_w = 16$ can also be found in Figure 13 and that, for large blur sizes, increased tile sizes can be beneficial.

Gathering [Riguer et al. 2003] Tiled Splatting [Selgrad et al. 2016] Ours

Lee et al. [2010] with 16 samples Lee et al. [2010] with 32 samples Lee et al. [2010] with 64 samples

Figure 15: Quality comparison of different DOF-algorithms. Gathering shows washed-out areas and far-field leaking in blurry near field regions. Tiled splatting keeps better track of the ordering, but overly darkens interior areas. Our method produces proper partial occlusion. In contrast to the ray tracing methods (bottom row) it can be seen that splatting and alpha accumulation result in enlarged borders due to saturation. Note how with the ray tracing methods, even at 64 samples there is still considerable noise.

and 12, respectively) shows, this is not a severe limitation as large Coc-values can be managed without falling back to global sorting at all. Figure 14 shows run times for three different tile sizes, namely 16×16, 24×24 and 32×32. It shows that our default of 16 performs well for large and medium blur sizes.

Image Quality. Figure 15 displays renderings for the viewpoint shown in Figure 1 (left) for the different methods evaluated with regard to performance. It exemplifies that, in scenarios with partial occlusion, the single-layer methods (gathering [Riguer et al. 2003] and tiled splatting [Selgrad et al. 2016]) clearly show their limitations. They are very fast and provide high quality when blur is mostly visible in the far-field, but are not adept for this use-case. Ray tracing methods [Lee et al. 2010] generate excellent, high-quality results, but, as shown above, take more computation time. Even with 64 samples per pixel, there is still noticeable noise with strongly defocused areas and for smaller highlights. Our approach resides in-between: performance is closer to traditional, post-processing based methods, while image quality is much closer to ray tracing based images.

We note, however, that objects' outwards-blur is overestimated, as can be seen in Figure 15. This is a common artifact due to splatting based approaches [Jimenez 2014]. Especially with multiple layers of data the alpha accumulation saturates earlier. With ray tracing methods this is not the case as different rays can simply be averaged. When traversing lists and alpha blending in order this is not a simple as it is not clearly determined if any given fragment could be hit directly by a lens ray. However, even with slightly overestimated borders our results look much better than earlier, fast methods and at only a fraction of the cost of using ray tracing [Lee et al. 2010].

8 CONCLUSION

With this paper we have presented a novel depth of field rendering approach that runs in real-time on current generation graphics hardware, even for large scenes in challenging setups. While it runs at more than twice the render time required by the fastest gathering-based algorithms it can handle cases that such approaches systematically fail at. Competing approaches that solve the same issues are often ray tracing based and require a multiple of our render times to converge to images with low-enough noise. With our method we trade rendering noise for some overestimation on the border of out-of-focus objects, making them appear less thin, while still providing proper, order-correctly blended partial occlusion.

We believe that our approach strikes a promising compromise between noise-free results and real-time post-processing rendering performance.

REFERENCES

Timo Aila and Samuli Laine. 2009. Understanding the Efficiency of Ray Traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics 2009 (HPG '09)*. ACM, New York, NY, USA, 145–149.

Kenneth E. Batcher. 1968. Sorting Networks and their Applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. ACM, 307–314.

Bill Claff. 2016. FOV Tables: Field-of-view of lenses by focal length. https://www.nikonians.org/reviews/fov-tables. (2016). Accessed on 15 Dec, 2017.

Robert L. Cook, Thomas Porter, and Loren Carpenter. 1984. Distributed Ray Tracing. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '84)*. ACM, New York, NY, USA, 137–145.

Joe Demers. 2004. Depth of Field: A Survey of Techniques. In *GPU Gems*, Randima Fernando (Ed.). Pearson Higher Education.

Cass Everitt. 2001. *Interactive Order-Independent Transparency*. Technical Report. NVIDIA Corporation.

Paul Haeberli and Kurt Akeley. 1990. The Accumulation Buffer: Hardware Support for High-quality Rendering. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '90)*. ACM, New York, NY, USA, 309–318.

Earl Hammon. 2007. Practical Post-Process Depth of Field. In *GPU Gems III*, Hubert Nguyen (Ed.). Addison-Wesley.

Takahiro Harada, Jay McKee, and Jason C. Yang. 2012. Forward+: Bringing Deferred Lighting to the Next Level. In *Eurographics 2012 - Short Papers Proceedings, Cagliari, Italy, May 13-18, 2012*. 5–8.

Nikolai Hofmann, Phillip Bogendörfer, Marc Stamminger, and Kai Selgrad. 2017. Hierarchical Multi-layer Screen-space Ray Tracing. In *Proceedings of High Performance Graphics (HPG '17)*. ACM, New York, NY, USA, Article 18, 10 pages.

Jorge Jimenez. 2014. Next Generation Post Processing in Call of Duty Advanced Warfare. (July 2014). Siggraph 2014.

Jaroslav Křivánek, Jiří Žára, and Kadi Bouatouch. 2003. Fast Depth of Field Rendering with Surface Splatting. In *Computer Graphics International, 2003. Proceedings*. IEEE, 196–201.

Sungkil Lee, Elmar Eisemann, and Hans-Peter Seidel. 2009. Depth-of-field Rendering with Multiview Synthesis. *ACM Trans. Graph. (Proc. of SIGGRAPH Asia)* 28, 5 (2009).

Sungkil Lee, Elmar Eisemann, and Hans-Peter Seidel. 2010. Real-time Lens Blur Effects and Focus Control. *ACM Trans. Graph.* 29, 4, Article 65 (July 2010), 7 pages.

Sungkil Lee, Gerard Jounghyun Kim, and Seungmoon Choi. 2008. Real-Time Depth-of-Field Rendering Using Point Splatting on Per-Pixel Layers. *Computer Graphics Forum* (2008).

Fang Liu, Meng-Cheng Huang, Xue-Hui Liu, and En-Hua Wu. 2009. Efficient Depth Peeling via Bucket Sort. In *Proceedings of the Conference on High Performance Graphics 2009 (HPG '09)*. ACM, New York, NY, USA, 51–57.

Michael Mara, Morgan McGuire, Derek Nowrouzezahrai, and David Luebke. 2016. Deep G-Buffers for Stable Global Illumination Approximation. In *HPG*. 11.

Chunhui Mei, Voicu Popescu, and Elisha Sacks. 2005. The Occlusion Camera. *Computer Graphics Forum* (2005).

Nvidia. 2017. Tuning CUDA Applications for Pascal. `http://docs.nvidia.com/cuda/pascal-tuning-guide/index.html#shared-memory`. (2017).

Ola Olsson and Ulf Assarsson. 2011. Tiled Shading. *Journal of Graphics, GPU, and Game Tools* 15, 4 (2011), 235–251.

Michael Potmesil and Indranil Chakravarty. 1981. A Lens and Aperture Camera Model for Synthetic Image Generation. In *Proceedings of the 8th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '81)*. ACM, New York, NY, USA, 297–305.

Guennadi Riguer, Natalya Tatarchuk, and John R. Isidoro. 2003. Real-Time Depth of Field Simulation. In *ShaderX2: Shader Programming Tips and Tricks with DirectX 9.0*, Wolfgang Engel (Ed.). Wordware, Plano, Texas.

D Schedl and M Wimmer. 2012. A Layered Depth-of-Field Method for Solving Partial Occlusion. *Journal of WSCG* 20, 3 (2012), 239–246.

Thorsten Scheuermann and Natalya Tatarchuk. 2004. Improved Depth of Field Rendering. In *ShaderX3: Advanced Rendering with DirectX and OpenGL (Shaderx Series)*, Wolfgang Engel (Ed.). Charles River Media, Inc., Rockland, MA, USA, Chapter Advanced Depth-of-Field Rendering.

Benjamin Segovia, Jean Claude Iehl, Richard Mitanchey, and Bernard Péroche. 2006. Non-interleaved Deferred Shading of Interleaved Sample Patterns. In *Graphics Hardware*, Marc Olano and Philipp Slusallek (Eds.). The Eurographics Association.

Kai Selgrad, Linus Franke, and Marc Stamminger. 2016. Tiled Depth of Field Splatting. In *EG 2016 - Posters*, Luis Gonzaga Magalhaes and Rafal Mantiuk (Eds.). The Eurographics Association.

Kai Selgrad, Christian Reintges, Dominik Penk, Pascal Wagner, and Marc Stamminger. 2015. Real-time Depth of Field Using Multi-layer Filtering. In *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games (i3D '15)*. ACM, New York, NY, USA, 121–127.

Maryann Simmons and Carlo H. Séquin. 2000. *Tapestry: A Dynamic Mesh-based Display Representation for Interactive Rendering*. Springer Vienna, Vienna, 329–340.

Tiago Sousa. 2013. Graphic Gems - Cry Engine 3. (Aug. 2013). Siggraph 2013.

Gareth Thomas. 2014. Compute-Based GPU Particle Systems. (2014). GDC'14.

John White and Colin Barré-Brisebois. 2011. More Performance! Five Rendering Ideas from Battlefield 3 and Need For Speed: The Run. (Aug. 2011). Siggraph 2011.

Sven Widmer, Dawid Pajak, A. Schulz, Kari Pulli, Jan Kautz, Michael Goesele, and David Luebke. 2015. An Adaptive Acceleration Structure for Screen-space Ray Tracing. In *Proceedings of the 7th Conference on High-Performance Graphics (HPG '15)*. ACM, New York, NY, USA, 67–76.

Sven Widmer, Dominik Wodniok, Daniel Thul, Stefan Guthe, and Michael Goesele. 2016. Decoupled Space and Time Sampling of Motion and Defocus Blur for Unified Rendering of Transparent and Opaque Objects. *Computer Graphics Forum* (2016).

Barry Winn, David Whitaker, David B Elliott, and Nicholas J Phillips. 1994. Factors affecting light-adapted pupil size in normal human subjects. *Investigative Ophthalmology & Visual Science* 35, 3 (1994), 1132.

Jason C Yang, Justin Hensley, Holger Grün, and Nicolas Thibieroz. 2010. Real-Time Concurrent Linked List Construction on the GPU. *Computer Graphics Forum (Proc. EG Symposium on Rendering)* 29, 4 (2010), 1297–1304.

Tianshu Zhou, Jim X. Chen, and Mark Pullen. 2007. Accurate Depth of Field Simulation in Real Time. *Computer Graphics Forum* (2007).