


Fast Polygonal Splatting using Directional Kernel Difference

Y. Moroto¹ and T. Hachisuka² and N. Umetani¹ 

¹ The University of Tokyo, Japan

² University of Waterloo, Canada

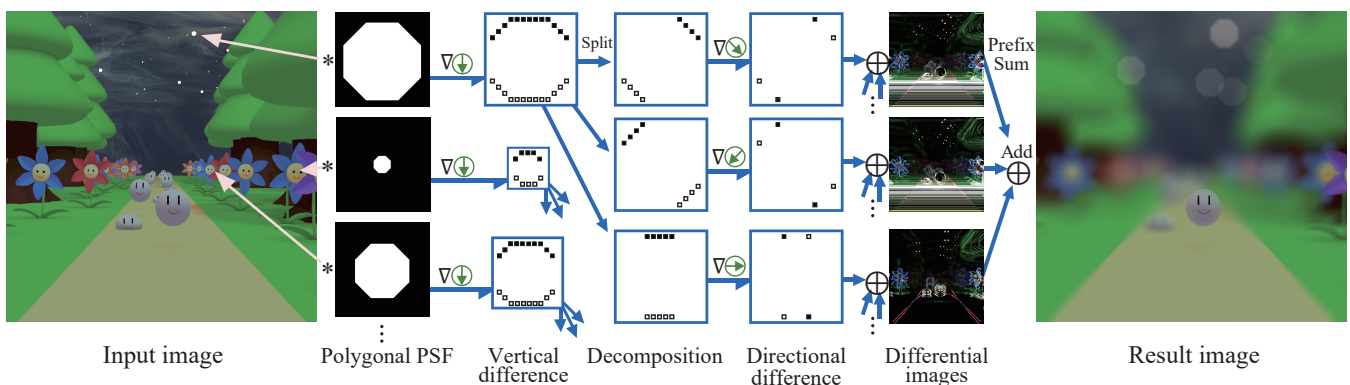


Figure 1: It is extremely expensive to compute depth-of-field image filtering where the Point Spread Function (PSF), i.e., kernel, is similar to that of actual camera lens, which typically takes polygonal shapes. For filtering with a flat polygonal kernel, we present a method that achieves a constant computation time regardless of the size of the kernel. The edges of the vertically differenced kernels are further differenced along the edge. The differenced kernel is highly sparse, which enables an efficient splatting. The blurred image is reconstructed by computing the prefix sum (i.e., cumulative sum) in the edge direction followed by the vertical direction.

Abstract

Depth-of-field (DoF) filtering is an important image-processing task for producing blurred images similar to those obtained with a large aperture camera lens. DoF filtering applies an image convolution with a spatially varying kernel and is thus computationally expensive, even on modern computational hardware. In this paper, we introduce an approach for fast and accurate DoF filtering for polygonal kernels, where the value is constant inside the kernel. Our approach is an extension of the existing approach based on discrete differenced kernels. The performance gain here hinges upon the fact that kernels typically become sparse (i.e., mostly zero) when taking the difference. We extended the existing approach to conventional axis-aligned differences to non-axis-aligned differences. The key insight is that taking such differences along the directions of the edges makes polygonal kernels significantly sparser than just taking the difference along the axis-aligned directions, as in existing studies. Compared to a naive image convolution, we achieve an order of magnitude speedup, allowing a real-time application of polygonal kernels even on high-resolution images.

CCS Concepts

• **Computing methodologies** → **Image processing**; **Rasterization**; **Image-based rendering**; **Massively parallel algorithms**;

1. Introduction

Lens blur is often used to create a sense of depth and is leveraged by photographers to guide the viewer’s attention toward the area in focus. Lens blur has gained increasing attention in recent years

owing to its esthetic appeal. Because lens blur typically requires a large-aperture lens, which is not readily available in mobile device cameras, the effect of lens blur is often achieved as a post process using *depth of field* (DoF) image filtering. DoF filtering is generally costly, particularly when the kernel size is large. Thus, it is

often difficult to apply a real-time computational animation, such as video games.

The DoF effect is typically achieved by filtering a pinhole image using a spatially varying kernel, that is, a *Point Spread Function* (PSF). The PSF specifies how the energy arriving at each pixel should be diffused over the image [LRT08]. The computational cost is proportional to the product of the number of pixels in the PSF (i.e., the size of the blur) and the number of pixels in the image. Thus, the computational cost increases rapidly when high-resolution images require extensive blurring. Acceleration is possible by hierarchically applying blur using an image pyramid to avoid direct convolution with a large kernel [KS07]. However, this approach is typically limited to extremely simple kernels, such as a Gaussian or box kernel, which is often quite different from the PSF of an actual camera lens. Because of the aperture blades, PSFs usually take the shape of polygons, and thus a polygonal kernel is often preferred to approximate the PSF [AMG*18] of the lens blur.

In this paper, we present an efficient DoF filtering method for polygonal kernels where the kernel is constant within a polygon and zero outside the polygon. Although, strictly speaking, the PSF of an actual camera lens is not entirely flat, many expensive lenses aim to provide flat bokeh, and many image editing software packages (e.g., Adobe Photoshop) employ flat PSF shapes for achieving a lens blur. Our basic approach involves filtering within the difference domain to increase the sparsity of the kernel. The final image is reconstructed from the filtered image by summing the differenced image by exploiting the associative and distributive properties of the filtering operation. Our approach achieves a better sparsity compared to the previous method by computing the differences *along the edges* of the polygon. Unlike other approaches, the number of non-zero elements in our method remains the same regardless of the size of the kernel.

Theoretically, our filtering operation does not involve any approximation, producing exactly the same outputs as a brute-force filtering method. This makes the filtering results coherent over the image frames and is thus suitable for film and video game applications. However, our method only supports the kernel of a polygon whose edges have slopes represented by simple integer ratios between their rise and run. Therefore, we also developed a technique for rectifying the input polygon of the user into that supported by our algorithm. Our rectification method is based on the dynamic programming approach and is extremely efficient, requiring only a few seconds of pre-computation to find the best approximation to the input polygon.

We evaluated the performance of our method through various comparisons with existing techniques and achieved orders-of-magnitude faster image filtering, particularly when the kernel size was large. We also demonstrated the results of image filtering using complex polygonal kernel shapes.

2. Related Work

When rendering a three-dimensional scene, lens blur can be computed by directly modeling the optical lens response through ray tracing [CPC84, HA90, LES09, SSD*09, LES10, JKL*16, PJH16].

However, although accurate, this method is generally computationally expensive for high-resolution images.

Rather than simulating light going through a lens, image-based approaches apply filters to pinhole images [PC81, Rok93]. We refer to this post-processing of the original image as “lens blur filtering.” Such an image-based approach performs better than other methods and should therefore be preferred for interactive applications such as video games [GK07, AMG*18]. Our method simulates lens blur through image filtering. Various approaches have been developed for the efficient computation of lens blur filtering. Hensley et al. [HSC*05] used a summed-area table [Cro84] to accelerate filtering through the kernels, as defined by Heckbert [Hec86]. Pyramidal image processing [KS07] can also be used to accelerate the filtering; however, the filtering kernel (PSF) is limited to simple functions, such as Gaussian functions. Lee et al. [LKC08] efficiently computed the lens blur with occlusions using layered images. Subsequently, they described an anisotropically filtered MIP mapping technique that reduced the artifacts associated with lens blur filtering [LKC09]. Imajo [Ima12] approximated a Gaussian function with a spline curve to accelerate the computation. The methods described above do not provide accurate results when the PSF is polygonal. Hensley et al. [HSC*05] reported precise results for certain types of kernels, including box kernels, but not for polygonal kernels. In addition, apart from the methods by Kosloff et al. [KTB09] and Imajo [Ima12], the approaches mentioned above compute the convolution of a pinhole image according to the PSF and can thus be computationally expensive when the kernel size is large.

When all pixels are uniformly blurred through a simple convolution against the fixed kernel, we can leverage the *fast Fourier transform* (FFT) for a fast filtering computation. Both the input image and kernel are transformed into the frequency domain, and the product is then computed. The filtered image is also then computed by applying an inverse Fourier transform. The FFT-based method is efficient because the kernel complexity does not affect the computational time.

Kernel splatting is typically a bottleneck in DoF filtering, and the more non-zero elements present in the kernel, the longer the computation time required to split the kernel. Therefore, significant effort has been made to reduce the number of non-zero elements to be splatted, particularly when the kernel radius is large. Our method leverages the property in which a flat PSF, i.e., a kernel with constant values, becomes sparse when the difference is taken.

Leimkühler et al. [LSR18] recently accelerated image filtering by exploiting the sparsity of the kernel when Laplacian is computed. Although this method closely approximates kernels whose values smoothly change, it is difficult to obtain a significant sparsification without an approximation error for flat polygonal kernels where the value discontinuously changes around the edge. The Laplacians of such polygonal kernels create large positive and negative values along the edge of the kernel, whereas our approach produces non-zero elements *only at the corners* of the polygon. In addition, for a reconstruction of differentiated images, our method requires a prefix sum (i.e., a *cumulative sum*), which is fast and accurate. By contrast, the method of [LSR18] requires solving the

Poisson’s equation, which is extremely expensive unless approximated using the convolution pyramid method [FFL11].

White et al. [WBB11] computed the lens blur effect using a hexagonal kernel. However, the application of this technique to other polygons is challenging, and the computational time is proportional to the kernel radius. The computational cost of our method is independent of the kernel size, and large blurs can be achieved. In addition, Kosloff et al. [KTB09] developed a DoF filter for box-shaped kernels with a constant per-pixel computational time. This represents a special case of the proposed method. This method is similar in that it computes the differencing; however, our method is different in that we take the differences in skewed directions other than the horizontal and vertical directions.

Even more similar to our approach, Harrington [Har03] applied summed-area tables to directions not aligned with the axis to achieve hexagonal and octagonal blur. Because this approach uses summed-area tables, it applies a summation (computing the summed-area table) and the difference (fetching the table). Theoretically, our approach is similar to this approach. However, the major difference is the order of summation and difference, i.e., we first filter using different kernels and then integrate the resulting image. This allows us to achieve a scatter blur, which is more similar to the actual blur mechanism, whereas their method is a gather blur. In the case of DoF filtering where the kernel size varies, the gathering is not physically correct and produces strong artifacts around the silhouettes where the kernel size discontinuously changes. Although DoF with gathering is easy to compute in parallel, it typically requires masks or costly post-filtering, such as bilateral filters around silhouettes. By contrast, scattering can compute the DoF filtering to accurately without the special treatment around such silhouettes.

Piponi [Pip12] introduced a fast and exact convolution method using a Z-transform, which is also independent of the kernel radius. However, this method only supports a gather-type DoF and convex polygons, and our approach can also handle concave polygons. Furthermore, in the case of hexagons, their method requires 28 samples per pixel, whereas our method requires only eight samples per pixel. For more complex polygons with many edges, the cost of their method is polynomial, whereas that of our method is linear.

A number of studies employ deep learning or generative adversarial networks to achieve the DoF effect by training the network with numerous photos taken with a camera [NAM*17, IPT20, QQL*21, CKCL20]. Some studies achieved filtering at an interactive speed for still images when the kernel was small. However, they are still too slow for real-time filtering required for games and video editing when the kernel is large. Alternatively, other methods filter input images based on estimated disparity maps or masks [BASH15, WGJ*18, LXJ*18, PSKA19, LLL*20, LPX*20]. These approaches are orthogonal to our technique because our method can be used to accelerate their filtering.

3. Overview

Kernel rectification. Our method computes the filtering of the input image I_{in} with a kernel \mathbf{K} , which is a rectangle-shaped binary array. The kernel is polygonal, that is, given a polygon, the kernel

takes a value of 1 inside the polygon and a zero otherwise. Our method requires each edge of the polygon to have a slope with the rise and run of small integers (e.g., 1:1, 2:1, or 1:0). For fast image filtering, it is also desirable for the polygon to have a minimum number of edges. Hence, in Section 4, we present a technique for efficiently rectifying the user-specified shape $\bar{\mathcal{P}}$ into a polygon \mathcal{P} that satisfies our requirements.

The rectified polygon \mathcal{P} is scaled to the specified pixel radius and then rasterized into a two-dimensional array \mathbf{K} for the following splatting computation. In the case of DoF filtering, the radius of the kernel is different for each pixel depending on its depth.

Kernel splatting. First, we compute the difference in the vertical direction. The differenced image takes non-zero values only above and below the non-vertical edge of the polygon. Subsequently, our method groups the pixels that belong to the edge of the same direction and again computes the difference along the direction of the edge. As a result, our method keeps the number of non-zero elements to a constant number, even if the size of the kernel increases. Finally, we compute the final filtered image by splatting to each direction buffer using this difference kernel and computing the prefix sum in the edge directions and finally in the vertical direction (see Section 5).

Note that in our implementation, we first take the vertical differences; however, this can be in other directions (e.g., horizontal differences). If the polygon of the kernel has a specific dominant direction, we can reduce the number of non-zero elements by differencing in that direction. However, for simplicity, our method can be described as taking the vertical differences first.

4. Kernel Rectification

Our method requires the kernel to be represented by a polygon \mathcal{P} whose edges have slopes with simple integer ratios between their rise and run (e.g., 2:1 or 1:0). This requirement is naturally satisfied when the kernel is simple regular polygons such as octagon. To handle more complicated polygons, we present a technique to rectify input polygon to satisfy this requirement. As the computational

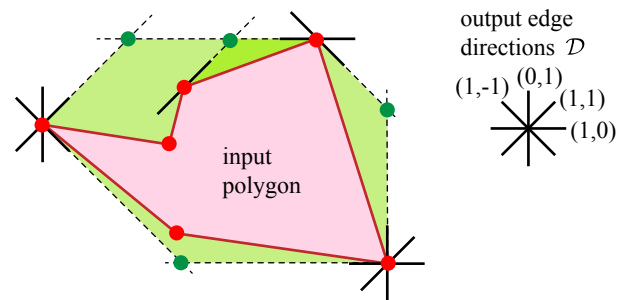


Figure 2: For each vertex of the input polygon (red circle), we place the candidate edges touching the vertex from outside (bold black line) chosen from the specified set of edge directions. We then list pairs of candidate edges (green circles) that cross outside the input polygon. Each enclosed area between the pair of crossing edges and the input polygon is associated with the pair (green region).

cost increases in proportion to the number of edges, it is necessary to express polygons with as few edges as possible.

This section presents a method for finding the polygon to satisfy the requirements using *Dynamic Programming* (DP) approach which has two steps: building a weighted graph and traversing it. We denote \mathcal{D} as the set of possible directions of the output edge where its members are the tuples of two small integers for the rise and run of the slope, e.g., $\mathcal{D} = \{(-1, +1), (+1, 0), \dots\}$. Aside from the input polygon $\bar{\mathcal{P}}$, our method takes \mathcal{D} and the maximum number of the output edge N as inputs.

Error metric. There are various metrics for measuring the difference between two polygons (e.g., Hausdorff distance or Jaccard distance). Our method finds the polygon with the smallest area that encloses the input polygon $\bar{\mathcal{P}}$. This metric allows us to find the polygon with desired edge property in an efficient and deterministic way approximating the input polygon with the minimum error. See Figure 6 for the examples of the output for various polygonal inputs.

Graph construction. All the edges of the output polygon \mathcal{P} must touch the vertices of the input polygon $\bar{\mathcal{P}}$, otherwise, the area difference can be further reduced by moving the edges inward. Hence, for every vertex of the input polygon, we first list all the potential output edges that touch the vertex from outside and their directions are included in \mathcal{D} (see Figure 2). We then check whether two candidate output edges that belong to the different vertices of $\bar{\mathcal{P}}$ cross outside the $\bar{\mathcal{P}}$. We record all such pairs of edges together with the area enclosed by the edges and the $\bar{\mathcal{P}}$. These pairs of edges can be seen as the edges of a weighted graph whose nodes are the vertices of the input polygon $\bar{\mathcal{P}}$.

Graph traversal. With the graph built, we search for the output polygon of minimum cost by traversing the graph. Starting from a vertex of the input polygon (i.e., node of the graph), we traverse the graph to come back to that vertex with one counter-clockwise rotation around the input polygon. Given the counter-clockwise traversing orientation, the graph can be seen as a *Directed Acyclic Graph*,

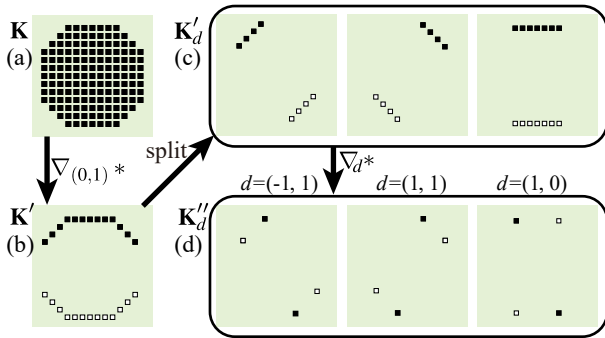


Figure 3: Kernel differencing. First, we compute the difference in the input kernel (a) in the vertical direction, which produces non-zero elements around non-vertical edges (b). These edges are grouped according to their direction (c). Then, by differencing along the edges, we obtain sparse differenced kernels (d).

therefore we topologically sort the graph and traversing nodes in sorted order to find the minimum cost. We find the output polygon with minimum cost by repeating this traversing procedure setting all the vertices of the input polygon as the starting vertex.

5. Splatting Kernel Using Directional Differencing

Section 4 describes a method for rectifying an input polygon into a polygon that can be handled by our method. To filter an image, we first rasterize the kernel into pixels \mathbf{K} and then compute the splatting against the input image $\mathbf{K} * \mathbf{I}_in$. This section describes how this convolution can be accelerated by computing the differenced kernel (Section 5.1) in the pre-computation and computing the splatting against the differenced kernel and summing the differenced filtered image (Section 5.2).

5.1. Directionally Differenced Kernel

Given the rasterized kernel \mathbf{K} of radius $r \in \mathbb{N}$, the vertical difference of the kernel is a convolution of the kernel with the differencing kernel in the vertical direction.

$$\mathbf{K}' = \nabla_{(0,1)} * \mathbf{K}, \quad (1)$$

where ∇_d denotes the differencing kernel in the direction of d . This vertical differencing of the kernel is equivalent to an element-wise computation as follows:

$$\mathbf{K}'(i, j) = \mathbf{K}(i, j) - \mathbf{K}(i, j - 1), \quad \text{where } -r \leq i, j \leq +r + 1, \quad (2)$$

where its component takes either -1 or $+1$ on the non-vertical edges (see Figure 3(b)).

Next, we decompose the vertically differenced kernel \mathbf{K}' into a set of kernels \mathbf{K}'_d in the same direction

$$\mathbf{K}' = \sum_{d \in \mathcal{D}} \mathbf{K}'_d, \quad (3)$$

where \mathcal{D} is the set of directions in the decomposed vertically differenced kernel (see Figure 3(c)).

Finally, we compute the difference of the decomposed kernel \mathbf{K}'_d in the direction of the edge. Let us denote the direction $d = (x_d, y_d) \in \mathcal{D}$; the directional difference of the decomposed kernel can then be written as

$$\mathbf{K}''_d = \nabla_d * \mathbf{K}'_d, \quad (4)$$

whose element-wise computation can be expressed as

$$\mathbf{K}''_d(i, j) = \mathbf{K}'_d(i, j) - \mathbf{K}'_d(i - x_d, j - y_d). \quad (5)$$

Because the component of the decomposed kernel \mathbf{K}'_d is constant along the edge, the difference in the edge direction leaves only a few non-zero elements per edge around its endpoints (see Figure 3(d)). Note that the number of non-zero elements in \mathbf{K}''_d is constant regardless of the blur radius r . This provides a significant advantage over the naïve implementation of kernel splatting, which has a number of non-zero elements in $\mathcal{O}(r^2)$.

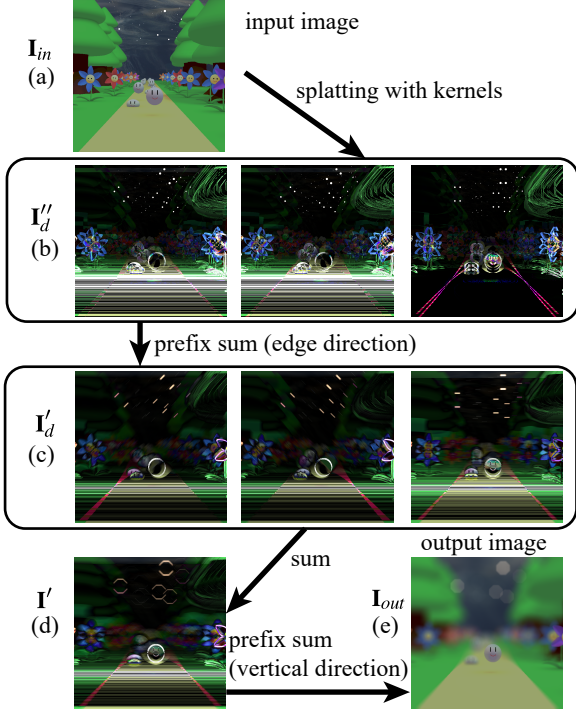


Figure 4: Runtime image filtering. The input image (a) is splatted with sparse differencing kernels (b). For each splatted image, we computed the prefix sum in the direction of the edge (c). These images sum to an image (d). Finally, by computing the prefix sum in the vertical direction, we obtain the output image (e).

5.2. Runtime Image Filtering

This section describes the runtime image filtering procedure. Please refer to Figure 4 for the overall procedure for an octagonal flat kernel. An input image I_{in} is filtered by applying a kernel to produce the output image I_{out} .

The prefix sum is an operation that adds up the value of the array in a specific direction. We denote the prefix sum in the direction of d as $Sum_d(\cdot)$ (see Appendix A for more detail). A differenced image can be reconstructed using this prefix sum as

$$\mathbf{K} = Sum_d(\nabla_d * \mathbf{K}) \quad (6)$$

In Figure 5, we illustrate the relationship between the directional difference ∇_d and prefix sum Sum_d . Using this notation, our filtering approach can be derived as follows:

$$\mathbf{I}_{out} = \mathbf{K} * \mathbf{I}_{in} = Sum_{(0,1)}(\nabla_{(0,1)} * \mathbf{K} * \mathbf{I}_{in}) \quad (7)$$

$$= Sum_{(0,1)}\left(\sum_{d \in \mathcal{D}} \mathbf{K}'_d * \mathbf{I}_{in}\right) \quad (8)$$

$$= Sum_{(0,1)}\left(\sum_{d \in \mathcal{D}} Sum_d(\mathbf{K}'_d * \mathbf{I}_{in})\right). \quad (9)$$

Note that we use the associative and distributive properties of the convolution operation.

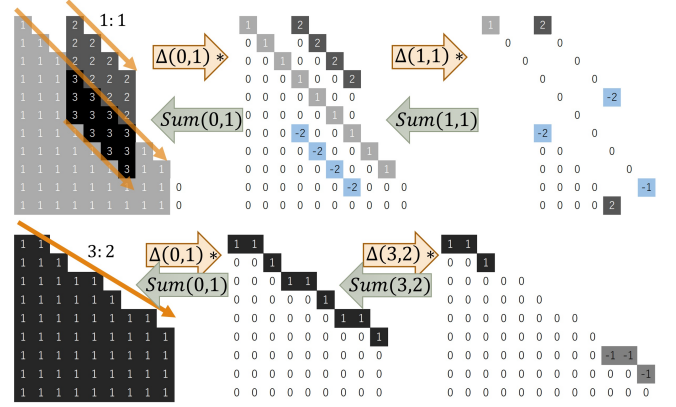


Figure 5: Relationship between directional differencing and its reconstruction based on a prefix sum. The top row shows edges with slopes having a ratio of run to rise of (1 : 1), and the bottom row shows an edge with a ratio of (3 : 2). In the case of a (1 : 1) ratio, there are two non-zero elements per edge, and in the case of (3 : 2), there are six non-zero elements per edge. The image can be reconstructed using the directional prefix sum. A simple slope ratio is preferable because it results in fewer non-zero elements.

We describe our filtering operation in detail (9). First, the image is split with the directionally differenced decomposed kernel $\mathbf{I}'_d = \mathbf{K}'_d * \mathbf{I}_{in}$. Next, for each direction, we computed the prefix sum of the image

$$\mathbf{I}'_d(m,n) = \mathbf{I}'_d(m-x_d, n-y_d) + \mathbf{I}'_d(m,n). \quad (10)$$

Note that this is the inverse operation corresponding to the operation of the directional difference in (5). We then compute the sum of the directionally summed image $\mathbf{I}' = \sum_{d \in \mathcal{D}} \mathbf{I}'_d$, which corresponds to the decomposition of the kernel in (3). Finally, we compute the prefix sum of the summed image to obtain the final filtered output image.

$$\mathbf{I}_{out}(m,n) = \mathbf{I}_{out}(m,n-1) + \mathbf{I}'(m,n), \quad (11)$$

which corresponds to the vertical difference in (2).

Computing the prefix sums in parallel is straightforward. The prefix sum of a two-dimensional array can be computed in parallel in each column, because each column is independent. In the case of CPU parallelization, the number of columns is generally much larger than the number of threads, and thus we can simply split the column tasks among all threads. In the case of a GPU, the number of threads may be larger than the number of columns, and thus a single column may need to be processed by dozens of threads in parallel. There is a popular technique for computing the prefix sum on a GPU [HSO07]. In our implementation on Nvidia CUDA, a sufficient speedup was achieved using one warp per column, because 32 threads in the same warp can exchange values without synchronization by applying a *shuffle* instruction.

6. Results

In this section, we first evaluate our kernel rectification method (Section 6.1) and then compare the differenced kernels (Section 6.2), as well as the computational efficiency and accuracy of our methods against those of the existing approaches. Note that all measured filtering times include time for both the splatting and reconstruction using prefix sum.

In this study, we evaluated the performance using an Intel®Core i9-9900X 3.5 GHz CPU, an Nvidia GeForce RTX 2080 Ti GPU, 64 GB of memory (DDR4-2400 Quad Channel), and Windows Server 2019 standard (version 1809) operating system.

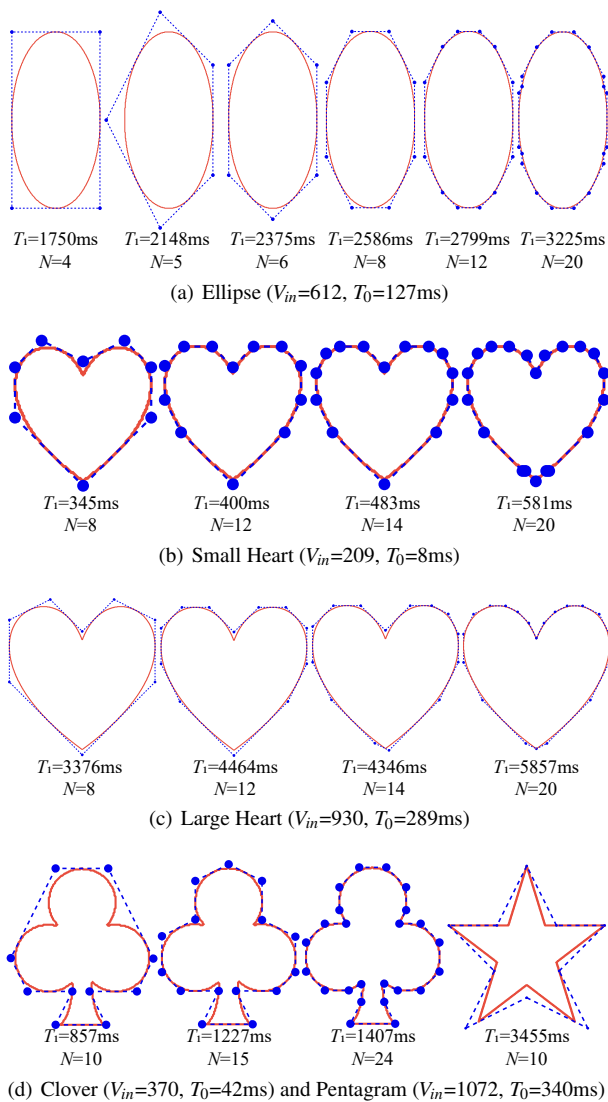


Figure 6: Examples of a rectified polygon (blue) for various input polygons (red) computed for various maximum numbers of output polygon edges N and the set of edge directions $\mathcal{D} = \{(1, 0), (0, 1), (1, 1), (1, -1), (1, 2), (1, -2), (2, 1), (2, -1)\}$. We denote the number of vertices in the input polygon as V_{in} , graph construction time as T_0 , and graph traversal time as T_1 .

6.1. Rectification of Polygonal Kernels

In Section 4, we present a method for modifying the input polygon such that it satisfies the constraint on the edge direction. Figure 6 shows examples of the rectified output polygons for various input polygons, including concave polygons for the various maximum numbers of the output polygon edges N . For all input polygons, we observed that the approximation of the input kernel steadily improves as the number of output edges N increases.

On rare occasions, our rectification algorithm breaks the symmetry of the input polygon. For example, the mirror symmetry is broken in the large heart example in Figure 6(c) with $N = 14$, and the rotational symmetry is broken in the Pentagram example in Figure 6(d). This is due to the error metric based solely on the area difference between the input and output polygons. Developing error metrics that handle other criteria (e.g., symmetry) is left for future study.

We measure the computational time for the two stages of our method: building a graph T_0 and its traversal T_1 (see Figure 6) for various polygonal inputs. In constructing the graph, most of the time is consumed in checking the intersection between the edge candidate and the input polygon, which takes longer when the number of vertices in the input polygon becomes larger. In the case of small and simple inputs, the graph is typically constructed within a few tens of milliseconds (see the small heart example in Figure 6(b)).

Our graph traversal method is based on the DP algorithm and has complexity in $\mathcal{O}(V_{in}^2 |\mathcal{D}| N)$. Figure 6 lists the traversal times for various inputs as T_1 . Even if the input polygon consists of more than 1000 vertices, our traversal algorithm finds the output polygon with a minimum error of a few seconds. Note that our DP-based algorithm finds the solution by listing all possible polygons with edges smaller than N . Hence, the polygon of the minimum error with edges smaller than N can be found without additional cost when finding a polygon of N edges.

6.2. Kernel Differencing for Commonly Used Polygons

Table 1 compares how kernels of commonly used polygons (hexagons, octagons, dodecagons, and hexadecagons) are differenced in three different approaches: our approach, differencing

Table 1: Number of non-zero elements in different kernel shapes and radiuses of kernels for three different approaches: a Laplace operator on the kernel [LSR18], horizontal and then vertical differencing (i.e., $\nabla_x \nabla_y$) based on [KTB09], and our proposed approach.

	Laplacian		$\nabla_x \nabla_y$		Ours	
	25	100	25	100	25	100
radius						
Hexagon	408	1608	204	804	8	
Octagon	292	1140	116	468	12	
Dodecagon	300	1196	108	412	28	
Hexadecagon	292	1140	116	468	36	

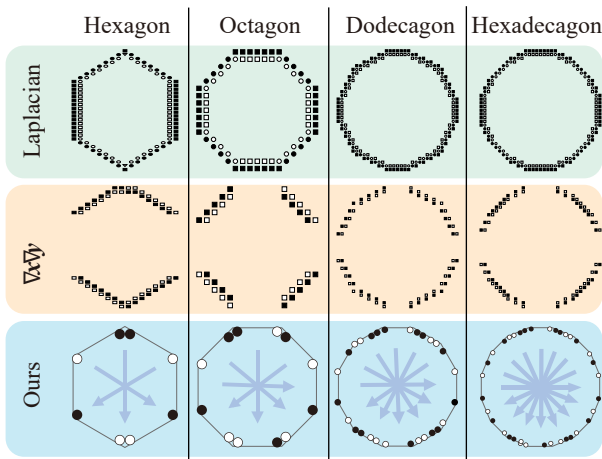


Figure 7: Comparison of the layout of non-zero elements among the three different approaches: the Laplacian approach (top row), the approach taking the horizontal difference followed by the vertical difference (middle row), and our proposed approach (bottom row). The arrows in our approach indicate the direction of the edge group. Our method generates non-zero elements only at the corners of the polygon, making the number of non-zero elements constant regardless of the pixel radius. Other approaches result in many non-zero elements on the edges, where the number of non-zero elements increases in proportion to the radius of the kernel.

with a Laplace operator [LSR18], and the approach used to compute the difference horizontally and then vertically based on the methods of Kosloff et al. [KTB09]. Note that a flat regular hexagonal kernel is different from the PSF of an actual camera; however, it is often employed in video editing software (e.g., Adobe After Effects) and video games [WBB11]. Although our method cannot directly handle circular kernel shapes, dodecagonal and hexadecagonal kernels can be used in practice as a good approximation of circles for video production and games. When approximating a circle with a polygon, the corners of the approximated polygon become more noticeable as the radius increases. This is visible in the output image when there are a few bright pixels in a dark input image (see Figure 10). However, in most cases, the corners of the approximated polygon do not stand out very much owing to the presence of blur.

As Figure 7 shows, the Laplacian approach [LSR18] and the approach developed by Kosloff et al. [KTB09] result in many non-zero elements on the edges of the polygon. However, our method produces non-zero elements only around the corners of the polygon. As seen in table 1, the Laplacian approach [LSR18] and the approach by Kosloff et al. [KTB09], which takes a horizontal and then a vertical difference, leave hundreds to thousands of non-zero elements for the large kernel, and the number of non-zero elements then increases proportionally to the radius. By contrast, our method requires only several tens of non-zero elements, which is an order of magnitude less than the other approaches, and the number of non-zero elements remains constant when the radius increases.

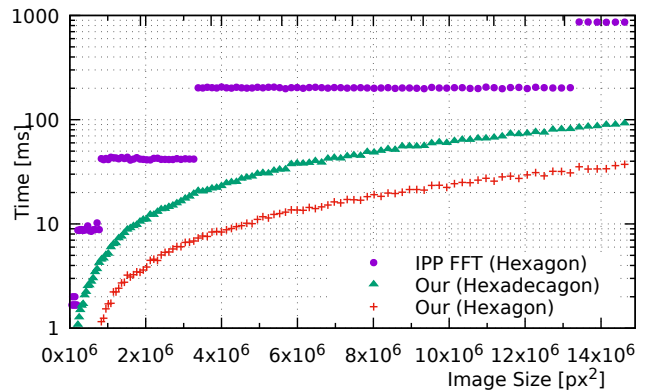


Figure 8: Comparison of our method against the method based on the fast Fourier transform for the computation of uniform blur on a CPU. The computation time is measured for different sizes of square input images, where the radius of the blur kernel is fixed (60 px). We measured the average running time of the defocus blur computation for ten runs, excluding the minimum and maximum cases for each image size.

6.3. Comparison against Fast Fourier Transform on CPU

Figure 8 compares the performance of our algorithm against the Fast Fourier Transform (FFT) method for filtering images of various sizes, where all pixels of the input image accept the same splatting kernels (e.g., uniform blur across the whole image). The experiment was performed on a 32-bit floating-point gray-scale square image and the radius of the kernel is fixed to 60 px. For the filtering method using an FFT, we compute the FFT for the kernel as a pre-computation, and this computation is not included in the time measurement. We used the FFT implementation in Intel’s Integrated Performance Primitives library, which has a highly optimized vectorized FFT implementation for our hardware setup. Both implementations use the SIMD instructions, and we execute both on a single thread on the CPU. Note that the graph for FFT does not continuously change because we padded the input image to a power of 2, allowing it to be handled by the FFT. In general, our

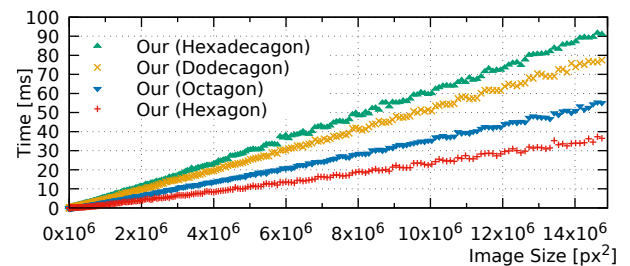


Figure 9: Computation time of our method for kernel shapes of hexagon, octagon, dodecagon, and hexadecagon with respect to the different sizes of a square input. The radius of the kernel was fixed at 60 px.

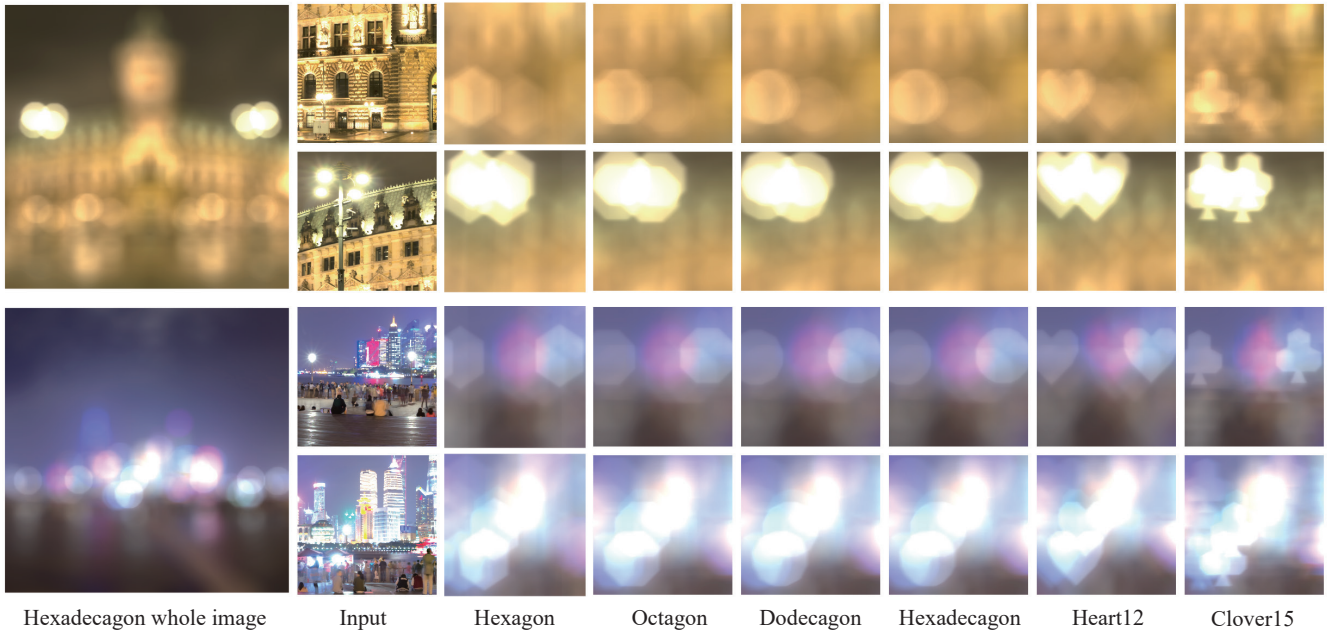


Figure 10: Image generated through our blurring method. The radius of the kernels was constant across the input image. The bright spot in the input image reveals the shape of the kernel in the output image. The blurred images with dodecagonal and hexadecagonal kernels show that they can be used as approximations to the circular kernel. Heart12 is a heart shape with 12 vertices, and Clover15 is a clover shape with 15 vertices. Even with such rough approximations, the synthesized images look natural as the splatting blurs the shapes of the kernel.

method achieves a running speed of 37- to 7.2-times faster than an FFT for images for a hexagonal kernel, and 10-to 2.5-times faster for a hexadecagonal kernel. Both FFT and our approach have computational complexities independent of the kernel size. Our method also runs faster for large images and bokeh because the complexity of FFT is approximately $O(n \log n)$, whereas that of our method is $O(n)$, where n is the number of pixels in the image.

6.4. Filtering with Various Polygonal Kernels on CPU

Figure 9 shows a plot of the running times of a uniform blur computation with different kernel shapes (hexagon, octagon, dodecagon, and hexadecagon). As the plot suggests, the running time of our method is proportional to the total number of pixels in the input image regardless of the shape of the kernel. Moreover, the running time is roughly proportional to the number of non-zero elements in the differenced kernel (e.g., the image filtering with a hexagonal kernel takes approximately 2.5-times longer than that of a hexadecagonal kernel, where the hexagon has 8 non-zero elements and the hexadecagon has 36 non-zero elements).

Figure 10 shows the blurring results for kernels computed from various polygons including the concave polygons (heart and clover). Notice that the hexagon in the image slightly deviates from the regular hexagon because our method is inherently unable to handle the edges of the slope with an irrational rise and run ratio. A perfectly regular hexagon can be easily achieved by splatting with a vertically scaled input image to adjust the aspect ratio.

Heart12 is an approximation of a heart with 12 vertices, and Clover15 is an approximation of a clover with 15 vertices, which are polygon kernels obtained using our method of Section 4, and the actual shapes can be seen in Figure 6. The computation time of Heart12 was almost the same as that of a dodecagon, and that of Clover15 was almost the same as that of a hexadecagon; therefore, the graphs in Figure 9 are omitted.

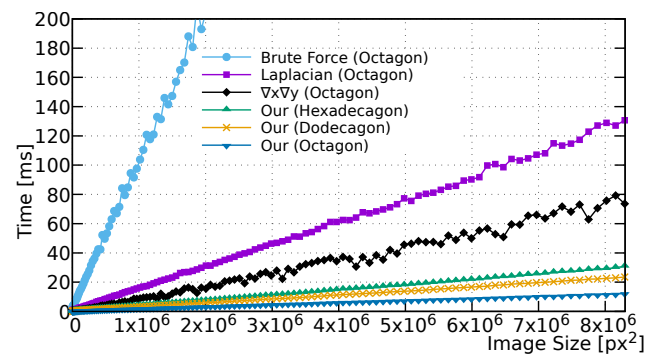


Figure 11: Comparison of GPU DoF filtering time among four different methods: our approach, horizontal and then vertical differencing based on the method developed by Kosloff et al. [KTB09] (i.e., $\nabla_x \nabla_y$), a Laplacian-based approach [LSR18], and the brute force method. The maximum radius of the DoF is fixed at 60 px, and the image size is increased to 3840×2160 .

6.5. Computational Speed of DoF Filtering on a GPU

Our filtering method is architecture-independent. All computations described in Sections 6.3 and 6.4 are processed on a CPU; however, our method can be computed on massively parallel processors such as a GPU. Figure 11 compares the total computation time for DoF filtering completed on the GPU with four different methods: the brute force method, Laplacian kernel splatting method [LSR18] (i.e., Laplacian method), the method taking a horizontal and then vertical difference based on the approach by Kosloff et al. [KTB09] (i.e., horizontal/vertical differencing method), and our approach. Here, the radius of the octagonal kernel is different across the input image depending on the depth (maximum kernel radius of 60 px). Similar to the result of constant blur computation on the CPU in Section 6.4, our method achieved the fastest computation speed among the four methods because of the small number of non-zero elements compared to other approaches (see Section 6.2). The results of DoF filtering are shown in Figure 12.

The differences in speed were clearer when the computation time was plotted for the kernel radius. Figure 13 compares the computation time when the maximum kernel radius is increased to 200 px for DoF filtering against a constant image size (1920 × 1080). This plot supports our estimation of the computational complexity for the kernel radius r , i.e., whereas the brute force method is $\mathcal{O}(r^2)$, and the Laplacian method, horizontal/vertical differencing method is $\mathcal{O}(r)$, our method is $\mathcal{O}(1)$, which is almost constant with respect to the kernel radius.

6.6. Accuracy of DoF Filtering on GPU

Finally, in Figure 14, we compare the computational accuracy of our method against the Laplacian kernel splatting method [LSR18].

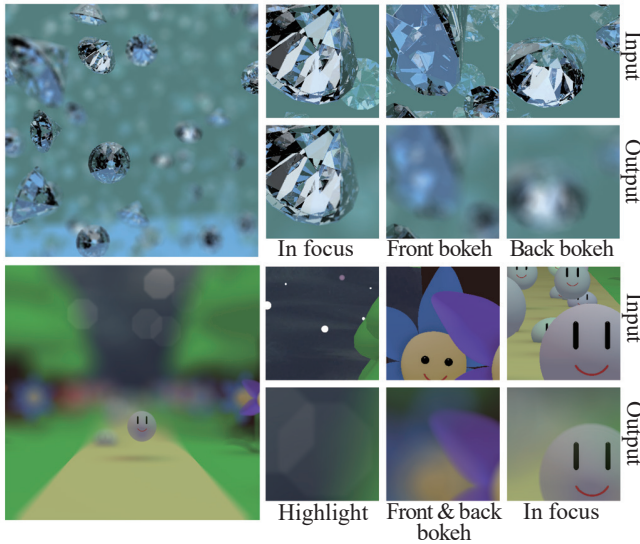


Figure 12: Image with DoF filter using our method. Because there is no approximation error, the output is exactly the same as that of the brute force method.

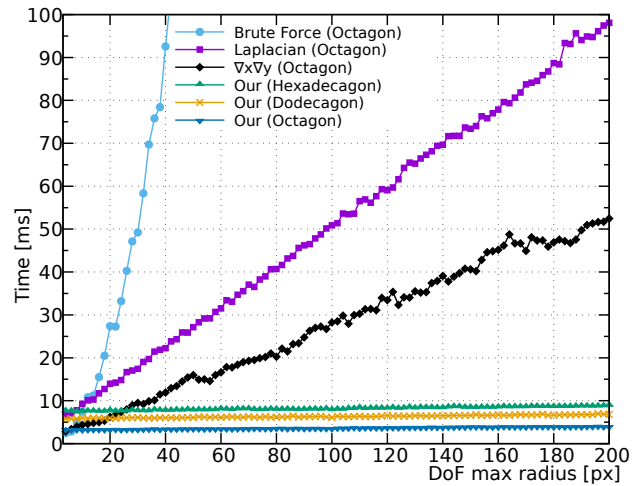


Figure 13: Computation time when the size of the image is fixed to 1920 × 1080 and the maximum DoF radius is increased to 200 px. The brute force has a computation time of $\mathcal{O}(nr^2)$; the Laplacian, $\mathcal{O}(nr)$; and our method, $\mathcal{O}(n)$, where n is the number of pixels and r is the kernel radius.

We applied a DoF filter with a maximum radius of 60 px on a float square image with a side length of 4 to 4096 px and compared the results with the brute-force approach. The error of our method is only approximately 10^{-7} , which is equivalent to the accuracy of a 32-bit float. By contrast, the Laplacian method has an average error of 3% and a maximum error of approximately 10%, which originates from the approximated reconstruction technique using the convolutional pyramid method [FFL11], which is essential for a fast computation using the Laplacian approach.

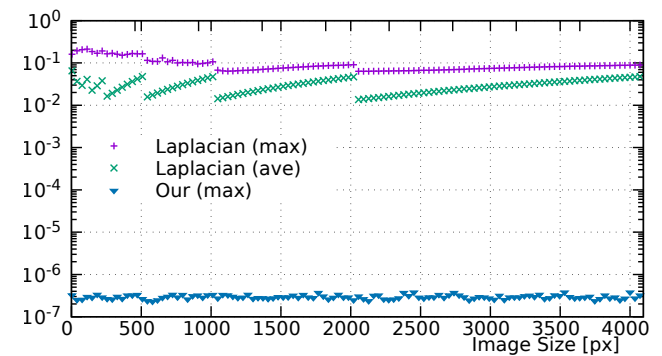


Figure 14: Error of DoF filtering on a floating-point image within the range of [0, 1] with a maximum blur radius of 60 px. The error is measured against the ground truth, which is computed using the brute force method at a high pixel depth.

7. Limitations and Future Work

A major limitation of our method is that it can only handle polygons with edges whose slopes are represented by simple integer ratios. Furthermore, the values of the kernel must be constant within the polygons. As the PSFs of real cameras are not constant kernels, our method cannot be used to simulate a photorealistic DoF. However, in practice, the use of constant polygonal kernels as PSFs is very popular among video games and film production; our method can be used in such applications.

Our rectification algorithm efficiently converts kernel shapes that our method does not directly support (e.g., curved shapes) into the one supported. However, the optimization metric and the solver of the rectification can be improved to facilitate real-time conversion. Supporting non-constant kernels remains important future work. This can be potentially achieved by approximating non-constant kernels as the combination of constant polygonal kernels.

Similar to other difference-based methods, our method also does not directly support the DoF filtering with occlusion, this is difficult because the depth changes discontinuously at the occlusion boundary. A technique that supports occlusion with differenced kernels is left as another target of future research.

8. Conclusion

We present a technique to compute the DoF of images using flat polygonal kernels. Unlike other methods, the speed of our method is not affected by the size of the kernel. Therefore, our method enjoys a much faster speed than the existing ones when kernel size is large. Furthermore, our filtering method is accurate and does not involve any approximation once the kernel is rectified. This was achieved by splatting in the gradient domain, where the gradient is computed by directional kernel differencing.

We improved the order of computational complexity for filtering. For example, a DoF filter with a maximum kernel radius of 200 px processing 1080 px square images was 13 times faster than the concurrent method and was orders of magnitude faster than older methods. We also developed an efficient rectification algorithm to convert kernels that cannot be handled directly by our method (e.g., those containing curved segments), into ones that can be dealt with minimum error. Our method and algorithm can efficiently produce high-definition filtered images in real-time and can be very useful in the production of videos and games.

Acknowledgements

I would like to thank Dr. Daisuke Takahashi and Dr. Aranha Claus de Castro for their helpful discussions. The 3D models used in the teaser and some of the tests were provided by Optie, who is an active video artist. I also would like to thank him very much. We also thank the anonymous reviewers for valuable feedback that has improved our manuscript.

References

[AMG*18] ABADIE G., MCAULEY S., GOLUBEV E., HILL S., LAGARDE S.: Advances in real-time rendering in games. In *ACM*

SIGGRAPH 2018 Courses (New York, NY, USA, 2018), SIGGRAPH '18, Association for Computing Machinery. URL: <https://doi.org/10.1145/3214834.3264541>, doi:10.1145/3214834.3264541. 2

[BASH15] BARRON J. T., ADAMS A., SHIH Y., HERNANDEZ C.: Fast bilateral-space stereo for synthetic defocus. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2015). 3

[CKCL20] CHOI M.-S., KIM J.-H., CHOI J.-H., LEE J.-S.: Efficient bokeh effect rendering using generative adversarial network. In *2020 IEEE International Conference on Consumer Electronics - Asia (ICCE-Asia)* (2020), pp. 1–5. doi:10.1109/ICCE-Asia49877.2020.9276807. 3

[CPC84] COOK R. L., PORTER T., CARPENTER L.: Distributed ray tracing. *SIGGRAPH Comput. Graph.* 18, 3 (Jan. 1984), 137–145. URL: <https://doi.org/10.1145/964965.808590>, doi:10.1145/964965.808590. 2

[Cro84] CROW F. C.: Summed-area tables for texture mapping. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques* (1984), pp. 207–212. 2

[FFL11] FARBMAN Z., FATTAL R., LISCHINSKI D.: Convolution pyramids. *ACM Trans. Graph.* 30, 6 (Dec. 2011), 1–8. URL: <https://doi.org/10.1145/2070781.2024209>, doi:10.1145/2070781.2024209. 3, 9

[GK07] GÖRANSSON J., KARLSSON A.: Practical post-process depth of field. *GPU Gems 3*, 583–606 (2007), 2. 2

[HA90] HAEBERLI P., AKELEY K.: The accumulation buffer: Hardware support for high-quality rendering. *SIGGRAPH Comput. Graph.* 24, 4 (Sept. 1990), 309–318. URL: <https://doi.org/10.1145/97880.97913>, doi:10.1145/97880.97913. 2

[Har03] HARRINGTON S. J.: Hexagonal and octagonal regions from summed-area tables, Jan. 14 2003. US Patent 6,507,676. 3

[Hec86] HECKBERT P. S.: Filtering by repeated integration. *ACM SIGGRAPH Computer Graphics* 20, 4 (1986), 315–321. 2

[HSC*05] HENSLEY J., SCHEUERMANN T., COOMBE G., SINGH M., LASTRA A.: Fast summed-area table generation and its applications. Citeseer. 2

[HSO07] HARRIS M., SENGUPTA S., OWENS J. D.: Parallel prefix sum (scan) with cuda. *GPU gems 3*, 39 (2007), 851–876. 5

[Ima12] IMAJO K.: Fast gaussian filtering algorithm using splines. In *Proceedings of the 21st International Conference on Pattern Recognition (ICPR2012)* (2012), pp. 489–492. 2

[IPT20] IGNATOV A., PATEL J., TIMOFTE R.: Rendering natural camera bokeh effect with deep learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops* (June 2020). 3

[JKL*16] JOO H., KWON S., LEE S., EISEMANN E., LEE S.: Efficient ray tracing through aspheric lenses and imperfect bokeh synthesis. *Computer Graphics Forum* 35, 4 (2016), 99–105. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.12953>, arXiv:<https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.12953>, doi:10.1111/cgf.12953. 2

[KS07] KRAUS M., STRENGERT M.: Depth-of-field rendering by pyramidal image processing. *Computer Graphics Forum* 26, 3 (2007), 645–654. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2007.01088.x>, arXiv:<https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2007.01088.x>, doi:10.1111/j.1467-8659.2007.01088.x. 2

[KTB09] KOSLOFF T. J., TAO M. W., BARSKY B. A.: Depth of field postprocessing for layered scenes using constant-time rectangle spreading. In *Proceedings of Graphics Interface 2009 (CAN, 2009)*, GI '09, Canadian Information Processing Society, p. 39–46. 2, 3, 6, 7, 8, 9

- [LES09] LEE S., EISEMANN E., SEIDEL H.-P.: Depth-of-field rendering with multiview synthesis. *ACM Trans. Graph.* 28, 5 (Dec. 2009), 1–6. URL: <https://doi.org/10.1145/1618452.1618480>, doi:10.1145/1618452.1618480. 2
- [LES10] LEE S., EISEMANN E., SEIDEL H.-P.: Real-time lens blur effects and focus control. *ACM Trans. Graph.* 29, 4 (July 2010). URL: <https://doi.org/10.1145/1778765.1778802>, doi:10.1145/1778765.1778802. 2
- [LKC08] LEE S., KIM G. J., CHOI S.: Real-time depth-of-field rendering using point splatting on per-pixel layers. *Computer Graphics Forum* 27, 7 (2008), 1955–1962. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2008.01344.x>, arXiv:<https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2008.01344.x>, doi:10.1111/j.1467-8659.2008.01344.x. 2
- [LKC09] LEE S., KIM G. J., CHOI S.: Real-time depth-of-field rendering using anisotropically filtered mipmap interpolation. *IEEE Transactions on Visualization and Computer Graphics* 15, 3 (2009), 453–464. 2
- [LLL*20] LUO C., LI Y., LIN K., CHEN G., LEE S.-J., CHOI J., YOO Y. F., POLLEY M. O.: Wavelet synthesis net for disparity estimation to synthesize dslr calibre bokeh effect on smartphones. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2020). 3
- [LPX*20] LUO X., PENG J., XIAN K., WU Z., CAO Z.: Bokeh rendering from defocus estimation. In *Computer Vision – ECCV 2020 Workshops* (Cham, 2020), Bartoli A., Fusiello A., (Eds.), Springer International Publishing, pp. 245–261. 3
- [LRT08] LANMAN D., RASKAR R., TAUBIN G.: Modeling and synthesis of aperture effects in cameras. In *Proceedings of the Fourth Eurographics Conference on Computational Aesthetics in Graphics, Visualization and Imaging* (Goslar, DEU, 2008), Computational Aesthetics’08, Eurographics Association, p. 81–88. 2
- [LSR18] LEIMKÜHLER T., SEIDEL H.-P., RITSCHER T.: Laplacian kernel splatting for efficient depth-of-field and motion blur synthesis or reconstruction. *ACM Trans. Graph.* 37, 4 (July 2018). URL: <https://doi.org/10.1145/3197517.3201379>, doi:10.1145/3197517.3201379. 2, 6, 7, 8, 9
- [LXJ*18] LIJUN W., XIAOHUI S., JIANMING Z., OLIVER W., ZHE L., CHIH-YAO H., SARAH K., HUCHUAN L.: DeepLens: Shallow depth of field from a single image. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 37, 6 (2018), 6:1–6:11. 3
- [NAM*17] NALBACH O., ARABADZHIYSKA E., MEHTA D., SEIDEL H.-P., RITSCHER T.: Deep shading: Convolutional neural networks for screen space shading. *Comput. Graph. Forum* 36, 4 (July 2017), 65–78. URL: <https://doi.org/10.1111/cgf.13225>, doi:10.1111/cgf.13225. 3
- [PC81] POTMESIL M., CHAKRAVARTY I.: A lens and aperture camera model for synthetic image generation. *SIGGRAPH Comput. Graph.* 15, 3 (Aug. 1981), 297–305. URL: <https://doi.org/10.1145/965161.806818>, doi:10.1145/965161.806818. 2
- [Pip12] PIPONI D.: Fast and exact convolution with polygonal filters. 3
- [PJH16] PHARR M., JAKOB W., HUMPHREYS G.: *Physically Based Rendering: From Theory to Implementation*. Elsevier Science, 2016. URL: <https://books.google.co.jp/books?id=inMVBQAAQBAJ>. 2
- [PSKA19] PUROHIT K., SUIN M., KANDULA P., AMBASAMUDRAM R.: Depth-guided dense dynamic filtering network for bokeh effect rendering. In *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)* (2019), pp. 3417–3426. doi:10.1109/ICCVW.2019.00424. 3
- [QQL*21] QIAN M., QIAO C., LIN J., ZHENYU G., LI C., LENG C., CHENG J.: *BGGAN: Bokeh-Glass Generative Adversarial Network for Rendering Realistic Bokeh*. 01 2021, pp. 229–244. doi:10.1007/978-3-030-67070-2_14. 3
- [Rok93] ROKITA P.: Fast generation of depth of field effects in computer graphics. *Computers & Graphics* 17, 5 (1993), 593 – 595. URL: <http://www.sciencedirect.com/science/article/pii/0097849393900107>, doi:[https://doi.org/10.1016/0097-8493\(93\)90010-7](https://doi.org/10.1016/0097-8493(93)90010-7). 2
- [SSD*09] SOLER C., SUBR K., DURAND F., HOLZSCHUCH N., SIL-LION F.: Fourier depth of field. *ACM Trans. Graph.* 28, 2 (May 2009). URL: <https://doi.org/10.1145/1516522.1516529>, doi:10.1145/1516522.1516529. 2
- [WBB11] WHITE J., BARRÉ-BRISEBOIS C.: More performance! five rendering ideas from battlefield 3 and need for speed: The run. *ACM SIGGRAPH 2011: Advances in the realtime rendering course* (2011). 3, 7
- [WGJ*18] WADHWA N., GARG R., JACOBS D. E., FELDMAN B. E., KANAZAWA N., CARROLL R., MOVSHOVITZ-ATTIAS Y., BARRON J. T., PRITCH Y., LEVOY M.: Synthetic depth-of-field with a single-camera mobile phone. *ACM Trans. Graph.* 37, 4 (July 2018). URL: <https://doi.org/10.1145/3197517.3201329>, doi:10.1145/3197517.3201329. 3

Appendix A: Directional Prefix Sum

This section explains directional prefix sum operator Sum_d and (6) in Section 5.2. When computing prefix sum in the direction of $(d_x, d_y) \in \mathbb{Z}^2$ on an input two-dimensional array K' , the output two-dimensional array $K = Sum_{(d_x, d_y)}(K')$ at the location $(i, j) \in \mathbb{Z}^2$ is computed by summing up the value of K' in the direction of (d_x, d_y) skipping pixels

$$K(i, j) = \sum_{n=0}^{\infty} K'(i - nd_x, j - nd_y). \quad (12)$$

Note that we define the value of the input array K' as zero outside its boundary. This prefix sum can be simply computed by *cumulatively* adding the value from the boundary of the array

$$K(i, j) = K(i - d_x, j - d_y) + K'(i, j). \quad (13)$$

In the meanwhile, when the array K is directionally differenced as $K' = \nabla_{(d_x, d_y)} * K$, its element is computed as

$$K'(i, j) = K(i, j) - K(i - d_x, j - d_y). \quad (14)$$

Using this relationship repetitively

$$K(i, j) = K'(i, j) + K(i - d_x, j - d_y) \quad (15)$$

$$= K'(i, j) + K'(i - d_x, j - d_y) + K(i - 2d_x, j - 2d_y) \quad (16)$$

$$= \sum_{n=0}^{\infty} K'(i - nd_x, j - nd_y), \quad (17)$$

we have the the identity in (6) as

$$K = Sum_{(d_x, d_y)}(\nabla_{(d_x, d_y)} * K). \quad (18)$$