# Depth of Field Postprocessing For Layered Scenes Using Constant-Time Rectangle Spreading

Todd J. Kosloff*
University of California, Berkeley
Computer Science Division
Berkeley, CA 94720-1776
USA

Michael W. Tao†
University of California, Berkeley
Computer Science Division
Berkeley, CA 94720-1776
USA

Brian A. Barsky‡
University of California, Berkeley
Computer Science Division and School of Optometry
Berkeley, CA 94720-1776
USA

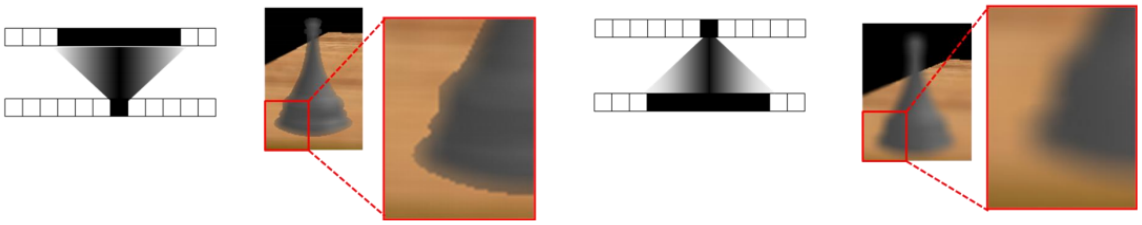

Figure 1: Gathering vs. Spreading. Image filtering is often implemented as gathering, in which output pixels are weighted averages of input pixels. Spreading involves distributing intensity from each input pixel to the surrounding region. For depth of field post-processing, spreading is a better approach. Left: Gathering leads to sharp silhouettes on blurred objects. Right: Spreading correctly blurs silhouettes. This simple scene uses two layers: one for the background, one for the foreground.

**ABSTRACT**

Control over what is in focus and what is not in focus in an image is an important artistic tool. The range of depth in a 3D scene that is imaged in sufficient focus through an optics system, such as a camera lens, is called depth of field. Without depth of field, the entire scene appears completely in sharp focus, leading to an unnatural, overly crisp appearance. Current techniques for rendering depth of field in computer graphics are either slow or suffer from artifacts, or restrict the choice of point spread function (PSF). In this paper, we present a new image filter based on rectangle spreading which is constant time per pixel. When used in a layered depth of field framework, our filter eliminates the intensity leakage and depth discontinuity artifacts that occur in previous methods. We also present several extensions to our rectangle spreading method to allow flexibility in the appearance of the blur through control over the PSF.

**Index Terms:** I.3.3 [Computer Graphics]: Picture/Image Generation-display algorithms, bitmap and frame buffer operations, viewing algorithms—

## 1 INTRODUCTION

### 1.1 Background

Control over what is in focus and what is not in focus in an image is an important artistic tool. The range of depth in a 3D scene that is imaged in sufficient focus through an optics system, such as a camera lens, is called *depth of field*. This forms a swath through a 3D scene that is bounded by two planes that for most cameras are both parallel to the film/image plane of the camera.

Professional photographers or cinematographers often achieve desired depth of field effects in the image by controlling the focus distance, aperture size, and focal length. For example, by restricting only part of a scene to be in focus, the viewer or the audience automatically attends primarily to that portion of the scene. Analogously, *pulling focus* in a movie directs the viewer to look at different places in the scene over time, following the point of focus as it moves continuously within the scene.

Rendering algorithms in computer graphics that lack depth of field are in fact modeling a pinhole camera. Without depth of field, the entire scene appears in completely sharp focus, leading to an unnatural, overly crisp appearance. Previous techniques for rendering depth of field in computer graphics are are either slow or suffer from artifacts or limitations.

Distributed ray tracing [8] can render scenes by directly simulating geometric optics, resulting in high quality depth of field effects. This requires many rays per pixel, leading to slow render times. Distributed ray tracing succesfully simulates *partial occlusion*, which is the fact that the edges of blurred objects are semi-transparent.

Post-processing, which adds blur to an image that was rendered with everything in perfect focus, is the alternative to distributed ray tracing for efficiently simulating depth of field. Previous fast post-process methods often fail to simulate partial occlusion, leading to unnaturally sharp edges on blurred objects, which we refer to as *depth discontinuity artifacts*. These previous methods also often lead to a blurred background leaking on top of an in-focus foreground, which we refer to as *intensity leakage artifacts*.

### 1.2 Goals

Ideally, our depth of field postprocess method should satisfy the following criteria:

1. Allows the amount of blur to vary arbitrarily for each pixel,

*e-mail: koslofto@cs.berkeley.edu
†e-mail: mtao@berkeley.edu
‡e-mail: barsky@cs.berkeley.edu

Graphics Interface Conference 2009
25-27 May, Kelowna, British Columbia, Canada

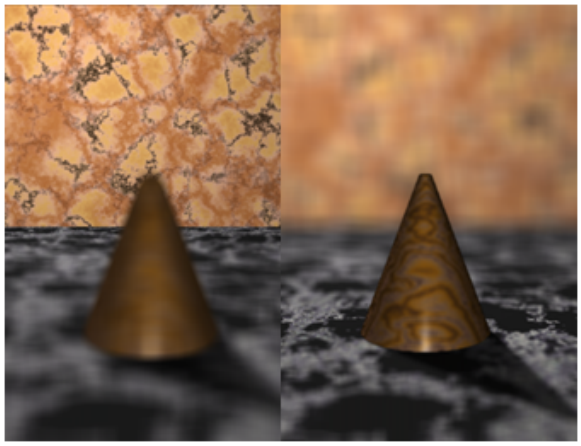

Figure 2: A scene blurred using our rectangle spreading method. Left: focused on wall. Right: focused on cone.

since the depth corresponding to each pixel can differ.

2. Achieves high performance, even for large amounts of blur.
3. Allows control over the nature of the blur, by allowing flexibility in choice of the PSF (Point Spread Function).
4. Simulates partial occlusion.
5. Avoids depth discontinuity artifacts.
6. Avoids intensity leakage artifacts.

In this paper, we describe new image filters that simultaneously meet all of these criteria when used on layered scenes.

We find it useful to classify image filters as using either *gathering* or *spreading* approaches, as will be explained in Section 3. Gathering filters generate output pixels by taking weighted averages of input pixels. Spreading involves distributing intensity from each input pixel to the surrounding region. Existing fast blur filters generally use gathering, which is physically incorrect, leading to noticeable flaws in the resulting image.

In this paper, we show that spreading removes these flaws, and can be very efficient. We describe a novel fast spreading blur filter, for use in depth of field, for layered scenes. Our method takes inspiration from the summed area table (SAT) [9], although our method differs substantially. An SAT is fundamentally a gather method whereas our method uses spreading. We can characterize the appearance of the blur in an image by the PSF, which describes how a point of light in the scene would appear in the blurred image.

Our first method uses rectangles of constant intensity as the PSF. To allow for a more flexible choice of PSFs, we describe two alternative extensions to our rectangle method: one that lifts the restriction to rectangular shape, allowing constant-intensity PSFs of arbitrary shape, and another that forms an algorithm based on a hybrid of any fast blur method with a slower, direct method that allows PSFs of arbitrary shape and intensity distribution. In this way, we achieve a controllable tradeoff between quality and speed.

### 1.3 Depth of Field Postprocessing of Layered Scenes

When blurring an image of a scene using a linear filter, blurred background objects will incorrectly leak onto sharp foreground objects, and sharp foreground objects that overlap blurred background objects will incorrectly exhibit sharp silhouettes. Layers are a simple, well-known method for solving this problem [1][4][5][21][29]. Each object is placed into a layer, along with an alpha matte. The layers and alpha mattes are blurred using FFT convolution or pyramids, and the blurred objects are composited using alpha blending with the blurred alpha mattes.

We describe a novel set of image filters suitable for efficiently extending the layered approach to nonplanar layers. As a prerequisite to using a layered method, the scene must be decomposed into layers. We assume that the decomposition has already been performed, either manually or automatically, and describe how to perform depth of field postprocessing using these layers.

## 2 Previous Work

### 2.1 Classic Depth of Field Methods

The first depth of field rendering algorithm was developed by Potmesil and Chakravarty [26]. They used a postprocess approach that employed complex PSFs derived from physical optics. Their direct, spatial domain filter is slow for large blurs.

Extremely high quality depth of field rendering can be achieved by distributed ray tracing, introduced by Cook [8]. Kolb [20] later simulated particular systems of camera lenses, including aberrations and distortions, using distributed ray tracing. Methods based on distributed ray tracing faithfully simulate geometric optics, but due to the number of rays required, are very slow. The accumulation buffer [15] uses rasterization hardware instead of tracing rays, but also becomes very slow for large blurs, especially in complex scenes.

### 2.2 Realtime Depth of Field Methods

A realtime postprocess method was developed by Scheuermann and Tatarchuk [28], suitable for interactive applications such as video games. However, this approach suffers from depth discontinuity artifacts, due to the use of gathering. This method selectively ignores certain pixels during the gathering in order to reduce intensity leakage artifacts; however, due to the use of a reduced resolution image that aggregates pixels from many different depths, this method does not eliminate them completely. Their method does not allow for a choice of point spread function; it produces an effective PSF that is a convolution of a bilinearly resampled image with random noise. Our methods enable a choice of PSF, and eliminate the depth discontinuity artifact.

A more recent realtime depth of field postprocess method was developed by Kraus and Strengert [21], who used pyramids to perform fast uniform blurring. By running the pyramid algorithm multiple times at different pyramid levels, they approximate a continuously varying blur. Unlike our method, their method does not provide a choice of PSFs, but rather produces a PSF that is roughly Gaussian.

Bertalmio et al. [6] showed that depth of field can be simulated as heat diffusion. Later, Kass et al. [19] used a GPU to solve the diffusion equation for depth of field in real time using an implicit method. Diffusion is notable for being a blurring process that is neither spreading nor gathering. Diffusion, much like pyramids, inherently leads to Gaussian PSFs.

Mulder and van Lier [23] used a fast pyramid method at the periphery, and a slower method with better PSF at the center of the image. This is somewhat similar to our hybrid method, but we use an image-dependent heuristic to adaptively decide where the high quality PSF is required.

Other methods [11][22][27][31] exist that quickly render depth of field, although we omit detailed discussion of them due to space constraints. For a comprehensive survey of depth of field methods, please consult Barsky et al.'s [2][3] and Demers' [10] surveys.

### 2.3 Layers

Barsky et al. used layers as one element in vision realistic rendering [1]. To our knowledge, the first published use of layers and alpha blending [25] for depth of field was Scofield [29]. Barsky et

al. showed in [4] and [5] how to use object identification to allow objects to span layers without engendering artifacts at the seams. The present paper also uses layers, although our image filters offer significant advantages over the FFT method previously used.

There are other methods that do not use layers, such as Catmull's method for independent pixel processing [7], which is efficient for scenes composed of a few large, untextured polygons, and Shinya's ray distribution buffer [30], which resolves intensity leakage in a very direct way, but at great additional cost. We choose to use layers due to their simplicity and efficiency.

### 2.4 Fast Image Filters

The method presented in this paper uses ideas similar to Crow's summed area table [9] originally intended for texture map anti-aliasing. Other constant-time image filters intended for texture map anti-aliasing methods include Fournier and Fiume [12] and Gotsman [14]. We build our method along the lines of Crow's, since that is the simplest.

Although Huang's method [18] for computing box filters is fast, Perrault et al. [24] extended it to be even faster, improving the computational cost from linear to constant with respect to kernel size. As the window of the kernel shifts, the new pixel values of the window are added to the summation while the old values are subtracted from the summation. By storing the summation of the columns, the process of calculating the blurred value becomes a constant-time computation. Unfortunately, these methods are restricted to rectangular kernels of constant size.

Finally, the Fast Fourier Transform (FFT) is a traditional fast method for blurring pictures via convolution, but can only be used when the amount of blur is constant throughout an image. Therefore, many FFTs are required if we want the appearance of continuous blur gradations. However, when many FFTs are required, we no longer have a fast algorithm, even if the FFTs are performed by a highly optimized implementation such as FFTW [13]. For this reason, we do not use FFTs in our method.

## 3 Gathering vs. Spreading

The process of convolving an image with a filter can be described equivalently as gathering or spreading. Gathering means that each pixel in the filtered image is a weighted average of pixels from the input image, where the weights are determined by centering the filter kernel at the appropriate pixel. Spreading means each pixel in the image is expanded into a copy of the filter kernel, and all these copies are summed together. When we allow the filter kernel to vary from pixel to pixel, gathering and spreading are no longer equivalent.

In any physically plausible postprocess depth of field method, each pixel must spread out according to that pixel's PSF. Since each pixel can have a PSF of different size and shape, we designed our filter to vary from pixel to pixel. Fast image filters typically only support gathering; thus, fast depth of field postprocess methods generally use gathering, despite the fact that spreading would produce more accurate results.

We initially considered using Crow's summed area table (SAT) as the image filter, in the manner of Hensley et al. [17]. SATs are appealing because they run in constant time per pixel and allow each pixel to have an independently chosen filter size. In any given layer, some of the pixels will be opaque, and others will be completely transparent. To simplify the presentation, we ignore semi-transparent pixels, although these are allowed as well. To generate the blurred color for an opaque pixel, we look up the depth value of the pixel, and determine a filter size using a lens model. We then perform an SAT lookup to determine the average color of the filter region. Unfortunately, it is not straightforward to determine a blurred color for a transparent pixel, because that corresponds to a region outside any object, and thus does not have a depth value. Depth values must be extrapolated from the opaque pixels to the transparent pixels, a process that generally only approximates the correct result, and requires additional computation. Barring such extrapolation, transparent pixels must be left completely transparent. Consequently, blurred pixels that are outside of, but adjacent to, an object remain completely transparent. Visually, this results in depth discontinuity artifacts (Figure 1, left).

A better option is to use a spreading filter. When an object is blurred by spreading, opaque pixels near silhouettes will spill out into the adjacent region, yielding a soft, blurred silhouette (Figure 1, right). This is a highly accurate approximation to the partial occlusion effect seen in real depth of field. This need for spreading filters motivates the constant-time spreading filters presented in this paper.

## 4 Constant Time Rectangle Spreading

We now present our depth of field postprocess approach for layered scenes. For example, consider a scene with a foreground layer and a background layer. Both layers are blurred, and are then composited with alpha-blending. Blurring and compositing layers for depth of field postprocessing is well-known [1][4][5][21][29]. However, previous methods generally use filters that limit each layer to a constant amount of blur; this is accurate only for 2D objects oriented parallel to the image plane. Our method alleviates this limitation by using a spatially varying spreading filter.

### 4.1 Motivation For Examining the Summed Area Table

We draw inspiration from Crow's summed area table (SAT). We base our method on the SAT because of the following speed and quality reasons.

Table-based gathering methods can be used in a setting where the table is created offline. This means that it is acceptable for table creation to be slow. Consequently, some of these methods do indeed have a lengthy table creation phase, such as Fournier and Fiume [12] and Gotsman [14]. However, the summed area table requires very little computation to create, and thus can be used online [17].

With an SAT, we can compute the average color of any rectangular region. Both the size and location of the region can be specified with pixel-level precision. Other methods, such as [12][14], give less precise control.

Although summed area tables have beneficial qualities, they can only be used for gathering, necessitating the development of a new method that can be used for spreading. Furthermore, SATs have precision requirements that grow with increased image size. Our rectangle spreading method does not inherit these precision problems because the signal being integrated includes alternating positive and negative values.

### 4.2 Our Rectangle-Spreading Method

First, we must create an array of the same dimensions as the image, using a floating point data type. We will refer to this as *table*. After initializing the array to zero, we enter Phase I of our method (Figure 4). Phase I involves iterating over each pixel in the input image, consulting a depth map and camera model to determine the size of the circle of confusion, and accumulating signed intensity markers in the array, at the corners of each rectangle that we wish to spread. Thus, we tabulate a collection of rectangles that are to be summed together to create the blurred image.

At this point, the array contains enough information to construct the blurred image, but the array itself is not the blurred image, since only the corners of each rectangle have been touched.

To create the blurred image from the array, we need Phase II of our method (Figure 5). Phase II is similar to creating a summed area table from an image. However, the input to Phase II is a table of accumulated signed intensities located at rectangle corners,

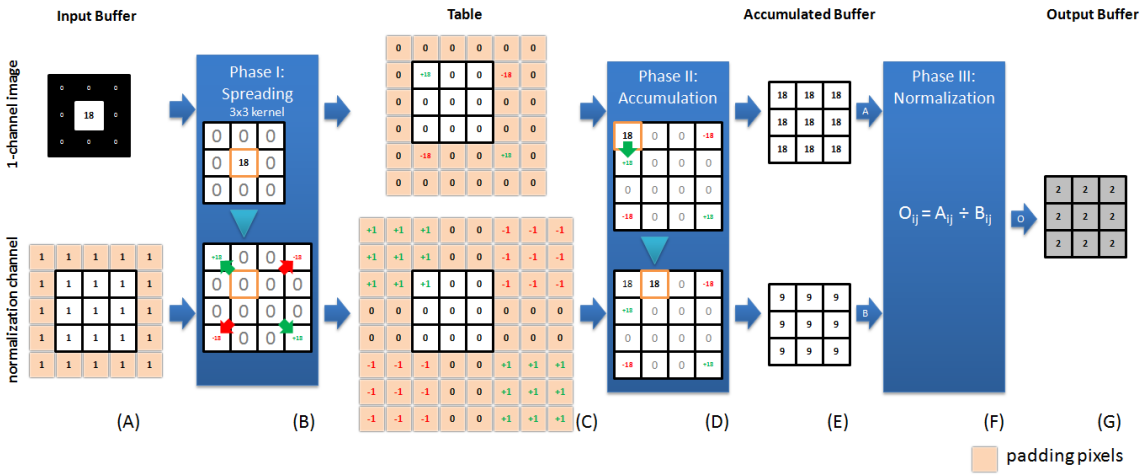


Figure 3: For every image input, there are two input buffers (A)- the image channel as an input and the normalization channel, which is the size of the image and contains pixels of intensity 1.0. In Phase I (B), for each pixel, we read the value of the pixel and spread the positive and negative markers accordingly on to the table with paddings added around the buffers (c). In Phase II (D), there is an accumulator (shown in orange) that scans through the entries of the tables from left to right, top to bottom. The accumulated buffer (E) stores the current state of the accumulator with the paddings removed. Phase II is equivalent to building a summed area table of the table (c). In phase III (F), for each entry at i and j, we divide the images accumulated buffer by the normalization channels accumulated buffer to obtain the output of the image (G).

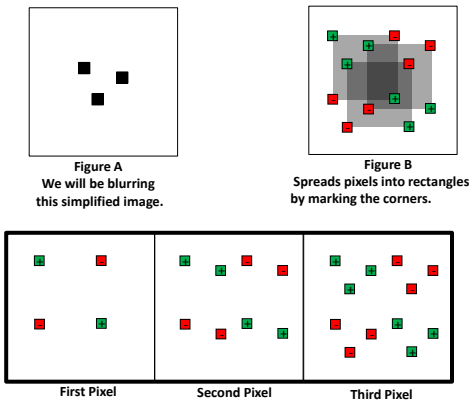


Figure 4: Phase I: Accumulating

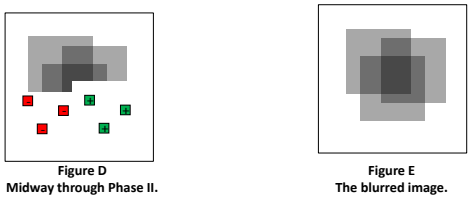


Figure 5: Phase II: Filling In

whereas the input to a summed area table creation routine is an image. Furthermore, the output of Phase II is a blurred image, whereas the output of a summed area table creation routine is a table meant to be queried later.

A summary of the entire blur process is illustrated in Figure 3.

```
//This code is for a single color channel,
//for simplicity.
//In a real implementation, R,G,B and A channels
//must all be blurred by this same process.
//Fast rectangle spreading: Phase I.
float table[width][height];
//zero out table (code omitted)
//S is the input image (S stands for sharp)
for(int i=0; i<width; i++)
for(int j=0; j<height;j++)
{
    int radius = get_blur_radius(i,j);
    float area = (radius*2+1)*(radius*2+1);

    table[i-radius][j-radius]+= S[i][j] / area;
    table[i+radius][j-radius]-= S[i][j] / area;
    table[i-radius][j+radius]-= S[i][j] / area;
    table[i+radius][j+radius]+= S[i][j] / area;
}

//Fast rectangle spreading: Phase II.
float accum;
float I[width][height]; //I is the blurred image
for(int y=1; y<height;y++)
{
    accum = 0;
    for(int x=0; x<width; x++)
    {
    accum += table[x][y];
    I[x][y] = accum + I[x][y-1];
    }
}
```

### 4.3 Normalization

Each output pixel is the sum of a variable number of rectangles. The result of executing the above code listing is that the pixels receiving more rectangles will appear too bright, whereas the pixels receiving fewer rectangles will appear too dim. We fix this by adding a fourth channel, which we will use for normalization. We spread rectangles of unit intensity into the normalization channel, and divide through by the normalization channel at the end. Note that overlapping PSFs of disparate sizes will have appropriate relative contributions to the final image, since intensities were divided through by area during the initial spreading phase. The alpha channel is not normalized, to preserve semi-transparent silhouettes (partial occlusion). In place of normalization, the alpha channel is biased slightly to prevent dimming, and clamped at 1.0 to avoid brightening.

### 4.4 Non-Integer Blur Sizes

In scenes with continuously varying depth levels, there will often be pixels whose blur values are not integers. If we were to simply use rectangles whose radius is the closest integer to the desired blur value, visible discontinuities would appear in the blurred image. Fortunately, we can easily approximate fractional blur sizes by spreading two integer sized rectangles, one slightly larger than the other, with weights dependent on the size of the desired rectangle.

### 4.5 Borders

As is typical in blur algorithms, our methods require special consideration near the borders of the image. When a pixel near the border is spread into a PSF, the PSF may extend beyond the borders of the image. We handle this case by padding the image by a large enough amount to ensure that the PSFs never extend beyond the borders of the padded image. The padding is cropped away before the image is displayed.

### 4.6 GPU Implementation

We have developed a DirectX 10 implementation of fast rectangle spreading to achieve real-time performance. The implementation is straightforward.

1. Phase I

   To accumulate corners, each corner is rendered as a point primitive. To avoid transferring large amounts of geometry, the points are generated without the use of vertex buffers, via the vertex ID feature of DirectX 10. A vertex shader maps the vertex id to pixel coordinates and appropriate signed intensities. To cause the signed intensities to accumulate rather than overwrite one another, alpha blending hardware is used, configured to act as additive blending.

2. Phase II

   Any GPU implementation of SAT generation could be used for the integration step. Currently we are using the recursive doubling approach [17].

Our GPU implementation achieves 45 frames per second with two layers at 800x600 on an ATI HD4870. Numerous low-level optimizations should raise performance even higher. Currently, our method is about twice as expensive as an SAT. We suspect this is due to the serialization that inherently must occur inside the alpha blending units when multiple pixels spread corners to the same point. As future work, we plan to mitigate the serialization problem by reordering the spreading to reduce contention.

Figure 7 demonstrates our GPU implementation. This example also shows that spreading works relatively well even in the absence of layers, whereas gathering leads to particularly severe artifacts, as can be seen in Figure 6. Our method is therefore applicable, if imperfect, for complex scenes that cannot be decomposed into layers.

Figure 6: A complex scene, *without* layers, blurred using a summed area table. Depth discontinuity artifacts are severe. (This 3D model is provided by the Microsoft DirectX 10 SDK.)

Figure 7: A complex scene, *without* layers, blurred using rectangle spreading. Depth discontinuity artifacts are reduced. (This 3D model is provided by the Microsoft DirectX 10 SDK.)

## 5 Arbitrarily Shaped Constant Intensity PSFs

Although our rectangle spreading method is fast and simple, we require a different method for applications that mandate circular or hexagonal PSFs, which are necessary to mimic the PSFs commonly found in real photographs.

There is a simple modification to our rectangle method that lifts the rectangular limitation, enabling arbitrarily shaped PSFs with constant intensity. We modify Phase I such that, for each scanline, we write markers wherever the PSF boundary intersects the scanline. In Phase II, each scanline is integrated in a one dimensional fashion. These changes are very simple to implement, but the enhanced generality comes at a cost: blurring now has a cost proportional to the perimeter of the PSF, whereas our rectangle method had a fixed cost for any size PSF. Note that direct blurring has a cost related to the area of the PSF, so a cost related to the perimeter is a significant improvement. Also note that a cost proportional to the perimeter is optimal for PSFs that have arbitrary, per-pixel shapes.

## 6 Hybridizing With Arbitrary PSFs

### 6.1 Contrast-Controlled Hybrid Method Concept

In the previous section, we lifted the restriction of rectangular shape to allow arbitrarily shaped PSFs of constant intensity. In this section, we further lift the restriction on contant intensity, to allow PSFs with arbitrary intensity distribution. PSFs that are derived from physical rather than geometric optics can have complex diffraction fringes, and are not well approximated as having constant intensity. Highly complex PSFs are best dealt with directly.

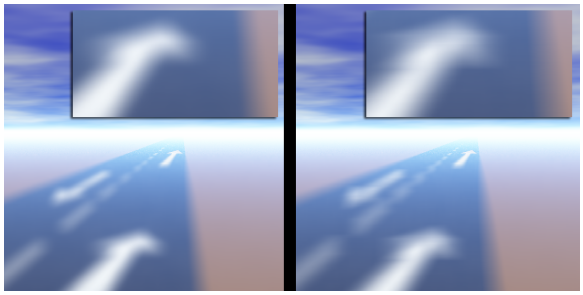

Figure 8: This example demonstrates continuously varying blur, and the hybrid method. Left: Hybrid method. Right: Rectangle spreading. Observe that the hybrid method significantly improves image quality, especially at high contrast edges.

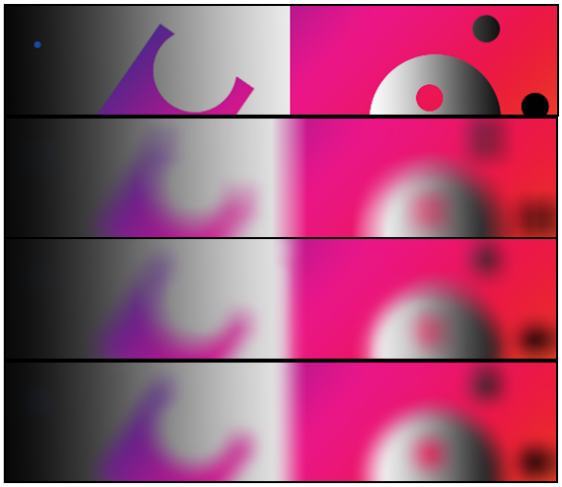

Figure 9: Top: image with high and low contrast regions. 2nd from top: image blurred via rectangle spreading. 2nd from bottom: image blurred with a smooth PSF. Bottom: The image blurred with our hybrid method. Notice that the hybrid image looks similar to the high quality PSF image, but took much less time to blur.

Although this is extremely expensive, accurate PSFs are not necessary at every pixel, but only in the high contrast regions of the image. Thus, we can develop a hybrid method that blurs the low contrast portions of the image with our fast rectangle spreading method while using accurate PSFs in the high contrast regions; this will produce an image without significant artifacts. See Figures 8 and 9 for examples.

### 6.2 Details of the Contrast-Controlled Hybrid Method

The first step is to determine the contrast of each pixel. Our contrast measure is the absolute difference between a pixel's intensity and the average intensity of that pixel's neighborhood. The size of the neighborhood is the size of the PSF for that pixel. We use a summed area table to efficiently compute the average intensity over the neighborhood. This contrast measure is simple and works reasonably well; however, we emphasize that our hybrid approach could be used with other contrast measures, as well as frequency measures, if this were found to produce better results. Having determined the contrast, we apply a threshold to determine whether to use a full-quality PSF, or whether to use a fast rectangle instead. The threshold is a user-definable parameter that trades off quality for speed.

A key contribution of this hybrid method is that the speed/quality

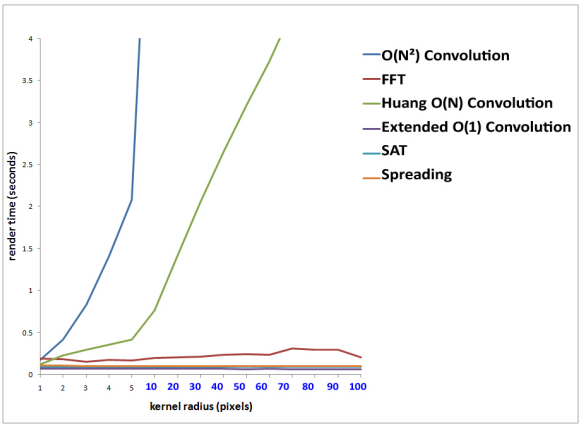


Figure 10: Our constant-time rectangle spreading filter compares competitively with other fast box filters while improving the quality for depth of field. Tests are done on a 1680 x 1050 image, Intel Core 2 Quad Q9450 processor, 4 GB RAM.

tradeoff can be finely tuned. When the contrast threshold is very high, this method degenerates into our fast rectangle approach. When the contrast threshold is very low, this method becomes a direct filter that uses precisely the true PSF. The threshold can be set anywhere along this continuum, making this method useful for a range of applications. We find that we can raise the threshold until our method takes only 30 percent the time of the direct method, before significant quality loss occurs.

## 7 Performance Comparison

Our rectangle spreading method has $O(1)$ cost per pixel, and the arbitrary shape spreading variant has $O(p)$ cost per pixel, where $p$ is the perimeter of the PSF. However, it is still important to consider experimental timings to determine if the proportionality constants are reasonable in comparison to other methods.

Our fast rectangle method uses constant-intensity rectangular PSFs, so we compare against other methods that use constant-intensity rectangular PSFs, including $O(N^2)$ spatial-domain convolution, SATs, and Huang's methods, both basic [18] and extended [24]. See Figure 10. For completeness, we also compare against fast convolution via FFT. These comparisons are for software implementations, as we do not yet have complete GPU implementations for some of these methods. Our method performs the same as SATs, much faster than direct convolution, much faster than Huang's linear-time method, significantly faster than the FFT method, and about 20 percent slower than Huang's constant time filter. Note that Huang's method as well as FFT convolution require the kernel to be the same at all pixels, but ours allows the kernel to vary.

Spatially uniform blurring methods can be used in the nonuniform case by repeated application with each different filter required. This increases cost, but it is conceivable that the simplicity of uniform blurring outweighs this added cost. We show this not to be the case in Figure 11.

Our arbitrary-outline method has a cost proportional to the perimeter of the PSF. We compare the performance of this method with other methods in Figure 12.

Finally, we compare the performance of our hybrid method with the fast rectangle blur, and with the slow detailed PSF method in Figure 13.

## 8 Limitations and Future Work

Some scenes cannot be decomposed into layers since objects can span many depth levels. Our simple treatment of layers only ap-

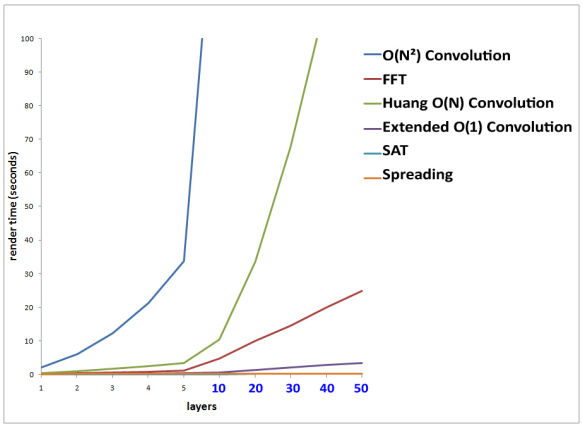


Figure 11: Our constant-time rectangle spreading filter compares competitively with the summed area table algorithm. To perform a spatially varying blur with a spatially uniform method, on the other hand, requires multiple executions, once for each kernel size. This leads to performance hits proportional to the number of different kernel sizes. Our spreading algorithm remains constant time as the number of kernel sizes increases because the algorithm requires only one execution. The first kernel radius starts with radius 5 and each kernel after the first the radius increments by 2. Tests are done on a 1680 x 1050 image, Intel Core 2 Quad Q9450 processor, 4 GB RAM.

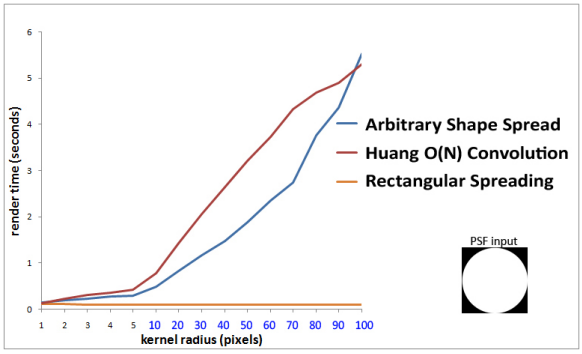


Figure 12: In this example, a constant intensity circle shape was used. The arbitrary shape spreading is linear to the circumference of the circle. Although the timings are significantly different at higher kernel radii, the method competes competitively with the Huang's linear convolution. Tests are done on a 1680 x 1050 image, Intel Core 2 Quad Q9450 processor, 4 GB RAM.

plies to scenes where objects do not straddle layers. This precludes self-occluding objects. The method of Kraus and Strengert [21] allows objects to straddle layers by applying a careful weighting to the hide the seam at layer boundaries. We believe that our fast spreading filter could be used with Kraus and Strengert's layering framework, to allow our method to work with arbitrarily complex scenes. This method would often need fewer layers than Kraus and Strengert's method, because we need additional layers only when there is additional depth complexity, whereas they need additional layers whenever there are additional amounts of blur, whether or not there is any depth complexity.

Our GPU implementation is real time at 45 frames per second with two layers, but it would be useful to have more layers. Further low-level optimizations of our GPU implementations remain, such as carefully ordering the spreading to avoid contention in the alpha blending hardware.

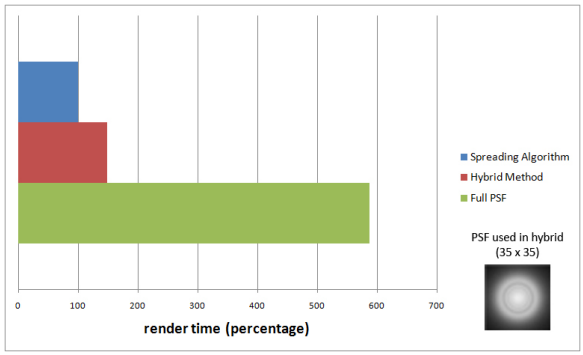


Figure 13: In this example, the hybrid used 1,761,271 fast rectangular PSFs and 127,747 slow detailed PSFs. As we can see, the hybrid method takes only 49 percent more time than the spreading algorithm while the using all PSF on the same image takes nearly 487 percent more time than the spreading algorithm. Through the image comparisons in figure 9, we can see that the image quality does not have a noticeable degradation. Tests are done on a 1680 x 1050 image, Intel Core 2 Quad Q9450 processor, 4 GB RAM.

Just as Crow's SATs can be extended to use arbitrary-order polynomial kernels via Heckbert's repeated integration scheme [16], our rectangle spreading method can also be extended in a similar way. In Phase I, the weights developed for Heckbert's scheme are accumulated, much as rectangle corners are accumulated in our rectangle spreading method. In Phase II, $N$ integration steps are performed, for polynomials of order $N$. We will describe the extension to polynomial kernels in future work.

Finally, the contrast measure in our hybrid method is a heuristic that seems to work well. Further work is required to determine more precisely when simple PSFs are sufficient, and when more complex PSFs are required.

## 9 Conclusion

This paper provides a method for artifact-free depth of field rendering in real time. We achieve this through the following specific contributions:

1. We show that spreading filters, unlike gathering filters, eliminate depth discontinuity artifacts by simulating partial occlusion.

2. We show that Crow's summed area table, fundamentally a gathering method, can be transformed into a rectangle spreading method.

3. We describe a GPU implementation of our rectangle spreading method that runs at 45 frames per second on current GPUs.

4. We extend the SAT-like method to handle constant-intensity PSFs of arbitrary shape, at modest additional cost.

5. We show how to forms an algorithm based on a hybrid of any fast spreading filter with arbitrary PSFs according to a controllable cost/quality tradeoff.

In this paper, we have shown that the well-known method of convolving and compositing 2D image-plane parallel layers (e.g. [1][4][5][21][29]) can be extended to non-planar layers with great efficiency, using constant-time spreading filters. Unlike previous depth of field methods of comparable speed, our methods provide choice of PSF, accurately simulate partial occlusion, and are free of depth discontinuity and intensity leakage artifacts.

We have presented three distinct algorithms, each of which is useful for a different application domain. Our rectangle spreading method runs at 45 frames per second on GPUs, enabling artifact-free depth of field rendering for interactive applications such as video games. Our arbitrarily shaped constant-intensity PSF method enables higher quality blur, although performance is not interactive. Our contrast-controlled hybrid method combines our fast rectangle spreading method with a direct spreading method that enables PSFs of arbitrary complexity, both in shape and intensity distribution. By varying the contrast threshold parameter, the user can configure the hybrid method to behave precisely like the rectangle spreading method, or precisely like the high quality direct method, as well as *anywhere in between*. This parameter can be configured, for example, to provide maximum quality for a given computational budget. If this budget is later increased due to using a faster computer, the contrast threshold parameter can be adjusted to exploit this computational power to achieve accordingly higher quality.

## REFERENCES


[1] B. A. Barsky. Vision-realistic rendering: simulation of the scanned foveal image from wavefront data of human subjects. In *APGV '04: Proceedings of the 1st Symposium on Applied perception in graphics and visualization*, pages 73–81, New York, NY, USA, August 2004. ACM.

[2] B. A. Barsky, D. R. Horn, S. A. Klein, J. A. Pang, and M. Yu. Camera models and optical systems used in computer graphics: Part i, object-based techniques. In *Proceedings of the 2003 International Conference on Computational Science and its Applications (ICCSA'03)*, pages 246–255, May 2003.

[3] B. A. Barsky, D. R. Horn, S. A. Klein, J. A. Pang, and M. Yu. Camera models and optical systems used in computer graphics: Part ii, image-based techniques. In *Proceedings of the 2003 International Conference on Computational Science and its Applications (ICCSA'03)*, pages 256–265, May 2003.

[4] B. A. Barsky, M. J. Tobias, D. Chu, and D. R. Horn. Elimination of artifacts due to occlusion and discretization problems in image space blurring techniques. *Graphical Models 67(6)*, pages 584–599, November 2005.

[5] B. A. Barsky, M. J. Tobias, D. R. Horn, and D. Chu. Investigating occlusion and discretization problems in image space blurring techniques. In *First International Conference on Vision, Video, and Graphics*, pages 97–102, July 2003.

[6] M. Bertalmio, P. Fort, and D. Sanchez-Crespo. Real-time, accurate depth of field using anisotropic diffusion and programmable graphics cards. In *Proceedings of the 2nd International Symposium on 3D Data Processing, Visualization and Transmission 3DPVT 2004*, pages 767–773, 6–9 Sept. 2004.

[7] E. Catmull. An analytic visible surface algorithm for independent pixel processing. In *SIGGRAPH 1984 Conference Proceedings*, pages 109–115. ACM Press, 1984.

[8] R. Cook, T. Porter, and L. Carpenter. Distributed ray tracing. In *ACM SIGGRAPH 1984 Conference Proceedings*, pages 137–145, 1984.

[9] F. C. Crow. Summed-area tables for texture mapping. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 207–212, New York, NY, USA, 1984. ACM Press.

[10] J. Demers. Depth of field: A survey of techniques. *GPU Gems*, pages 375–390, 2004.

[11] P. Fearing. Importance ordering for real-time depth of field. In *Proceedings of the Third International Computer Science Conference on Image Analysis Applications and Computer Graphics*, volume 1024, pages 372–380. Springer-Verlag Lecture Notes in Computer Science, 1995.

[12] A. Fournier and E. Fiume. Constant-time filtering with space-variant kernels. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 229–238, New York, NY, USA, 1988. ACM.

[13] M. Frigo and S. G. Johnson. The design and implementation of fftw3. *Proceedings of the IEEE 93 (2) Special Issue on Program Generation, Optimization, and Platform Adaptation*, 93(2):216–231, Feb. 2005.

[14] C. Gotsman. Constant-time filtering by singular value decomposition. *Computer Graphics Forum*, pages 153–163, 1994.

[15] P. Haeberli and K. Akeley. The accumulation buffer: hardware support for high-quality rendering. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 309–318, New York, NY, USA, 1990. ACM.

[16] P. S. Heckbert. Filtering by repeated integration. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 315–321, New York, NY, USA, 1986. ACM Press.

[17] J. Hensley, T. Scheuermann, G. Coombe, M. Singh, and A. Lastra. Fast summed-area table generation and its applications. In *Eurographics 2005*, 2005.

[18] T. Huang, G. Yang, and G. Yang. A fast two-dimensional median filtering algorithm. *IEEE Transactions on Acoustics, Speech, and Signal Processing v. 27*, pages 13–18, 1979.

[19] M. Kass, A. Lefohn, and J. Owens. Interactive depth of field. *Pixar Technical Memo 06-01*, 2006.

[20] C. Kolb, D. Mitchell, and P. Hanrahan. A realistic camera model for computer graphics. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 317–324, New York, NY, USA, 1995. ACM.

[21] M. Kraus and M. Strengert. Depth of field rendering by pyramidal image processing. *Computer Graphics Forum 26(3)*, 2007.

[22] J. Krivanek, J. Zara, and K. Bouatouch. Fast depth of field rendering with surface splatting. In *Computer Graphics International 2003*, 2003.

[23] J. Mulder and R. van Lier. Fast perception-based depth of field rendering. In *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 129–133, 2000.

[24] S. Perreault and P. Hebert. Median filtering in constant time. *IEEE Transactions on Image Processing 16(9)*, pages 2389–2394, 2007.

[25] T. Porter and T. Duff. Compositing digital images. In *ACM SIGGRAPH 1984 Conference Proceedings*, pages 253–259, New York, NY, USA, 1984. ACM.

[26] M. Potmesil and I. Chakravarty. Synthetic image generation with a lens and aperture camera model. *ACM Transactions on Graphics 1(2)*, pages 85–108, 1982.

[27] P. Rokita. Generating depth-of-field effects in virtual reality applications. *IEEE Computer Graphics and Applications 16(2)*, pages 18–21, 1996.

[28] T. Scheuermann and N. Tatarchuk. Advanced depth of field rendering. In *ShaderX3: Advanced Rendering with DirectX and OpenGL*, 2004.

[29] C. Scofield. 2 1/2-d depth of field simulation for computer animation. In *Graphics Gems III*. Morgan Kaufmann, 1994.

[30] M. Shinya. Post-filtering for depth of field simulation with ray distribution buffer. In *Proceedings of Graphics Interface '94*, pages 59–66. Canadian Information Processing Society, 1994.

[31] T. Zhou, J. X. Chen, and M. Pullen. Accurate depth of field simulation in real time. *Computer Graphics Forum 26(1)*, pages 15–23, 2007.