Fast Filter Spreading and its Applications

Todd Jerome Kosloff
Justin Hensley
Brian A. Barsky

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-54
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-54.html

April 30, 2009

Fast Filter Spreading and its Applications

Todd J. Kosloff*
University of California, Berkeley, EECS

Justin Hensley†
Advanced Micro Devices, Inc. – Graphics Product Group

Brian A. Barsky‡
University of California, Berkeley, EECS

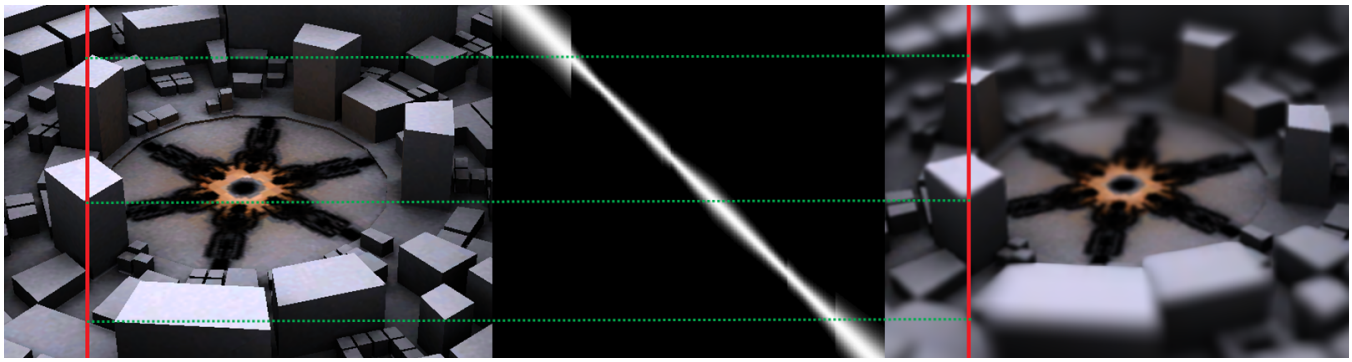


Figure 1: *Illustration of filter spreading. Left: an all-in-focus image. Right: The result of applying depth of field postprocessing via fast filter spreading. Center: A 2D (matrix) slice of the 4D (tensor) representation of the filter spreading operator. This particular slice corresponds to the column of the image indicated in red. The top and bottom green dotted lines show the correspondence between depth discontinuities in the image and discontinuities in the matrix. The center green dotted line shows how a discontinuity in the derivative of the depth map impacts the matrix. Observe that each column of the matrix is very simple. Our method achieves constant-time-per-pixel filtering by exploiting this column structure.*

Abstract

In this paper, we introduce a technique called *filter spreading*, which provides a novel mechanism for filtering signals such as images. By using the repeated-integration technique of Heckbert, and the fast summed-area table construction technique of Hensley, we can implement fast filter spreading in real-time using current graphics processors. Our fast implementation of filter spreading is achieved by running the operations of the standard summed-area technique in *reverse* - e.g. instead of computing a summed-area table and then sampling from a table to generate the output, data is first placed in the table, and then an image is computed by taking the summed-area table of the generated table. While filter spreading with a spatially invariant kernel results in the same image as one produced using a traditional filter, by using a spatially varying filter kernel, our technique enables numerous interesting possibilities. (For example, filter spreading more naturally mimics the effects of real lenses, such as a limited depth of field.)

CR Categories: I.4.3 [IMAGE PROCESSING AND COMPUTER VISION]: Enhancement—Filtering I.3.8 [COMPUTER GRAPHICS]: Applications;

Keywords: constant-time filtering, summed-area tables

1 Introduction

Image filtering is a pervasive operation in computer graphics. In this paper, we describe a technique which we refer to as *filter spreading*. Filter spreading can be thought of as a dual of standard image filtering. In standard image filtering, the output value of a pixel is determined by all the pixels that are spatially nearby. Conceptually, this is a *gathering* operation. This work introduces a method where each pixel *spreads* its value to nearby pixels. In the special situation where the filter kernel is constant over the entire image, the gather and spreading formulations are identical.

The paper is organized as follows: Section 2 introduces the concept of filter spreading using a linear algebraic approach. Next, in Section 3, we provide background information and discuss previous work. Then, in Section 4 we present the mathematical derivation of fast filter spreading via repeated integration. In Section 5 we describe how to translate the mathematics into an implementation of our algorithm. Section 6 presents the details of an example GPU implementation that provides real time performance. Section 7 describes separable spreading, a variation on our basic method that

*e-mail: koslofto@cs.berkeley.edu
†e-mail: Justin.Hensley@amd.com
‡e-mail: barsky@cs.berkeley.edu

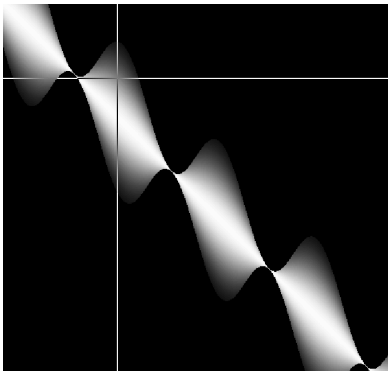


Figure 2: *A spreading matrix for blurring a one dimensional image with a sinusoidally varying Gaussian kernel. Observe that the matrix is not symmetric, i.e. the columns are simply Gaussians, whereas the rows are more complex, containing bits and pieces of various sized Gaussians.*

provides increased speed at the expense of a slight loss in visual quality. Section 8 presents several example applications of filter spreading. Finally, future work and conclusions are presented Section 9.

2 Spreading vs. Gathering

In this section, we use linear algebra to illustrate the relation between traditional image filtering (gathering) and filter spreading. In the texture mapping literature, gathering is known as reverse-mapping, and spreading is known as forward-mapping. Spreading is sometimes known as scattering, although we avoid this term because it implies randomness.

If an image is considered to be a column vector of pixels I, then a linear image filter can be considered to be a linear operator A acting on that vector to produce a filtered image O via matrix vector multiply: $O = A * I$. Usually we think of image filters as being defined by the filter kernel, but an image filter can instead be defined by a matrix that transforms an input image vector into a filtered image vector. This linear algebra approach is complicated by the fact that vectors are one dimensional arrays of numbers, whereas images are two dimensional arrays of pixels. Two dimensional images can be represented as vectors by unraveling the image in either row major or column major order. Alternatively, the image can be kept as a two dimensional entity, in which case the linear operator corresponding to the filter becomes an order-4 tensor. Our purpose in discussing the matrix viewpoint is conceptual, rather than algorithmic, so we simplify the discussion by restricting ourselves to one dimensional images, which lend themselves naturally to vector representation.

Consider a filter with a Gaussian kernel whose standard deviation varies from pixel to pixel. Such a filter might be used, for example, for-depth-of-field postprocessing. The variation in standard deviation depends on the particular scene being blurred, but for purposes of this example we use a simple sinusoidal variation. The corresponding matrix clearly contains one Gaussian of appropriate standard deviation, for each pixel. Conventionally filters are designed to compute the output by taking weighted averages of the input, so the Gaussians should be placed in the rows of the matrix.

Consider what would happen if we had constructed the matrix by arranging the Gaussians down the columns, rather than the rows, as illustrated in Figures 1 and 2. Our filter now has the effect of spreading input pixels into Gaussians, rather than taking weighted averages using Gaussian weights. If the matrix happened to be symmetric, then the rows would be the same as the columns, and the distinction between spreading and gathering would vanish. However, in many cases, such as the ones illustrated in Figures 1 and 2, the matrix is not symmetric, and so spreading and gathering produce different filters. The spreading approach to filtering is the topic of the current paper. For some applications, such as depth-of-field postprocessing, the use of gathering leads to artifacts, because spreading is a more appropriate model to the underlying optical process.

The highlighted row in Figure 2 shows that rows in a spreading matrix are not simply the filter kernel, but instead can contain arbitrary slices of different sized filter kernels. Even for a simple kernel such as a Gaussian, rows in the matrix can be arbitrarily complex. Therefore fast gather methods are not applicable, because the rows cannot be compactly expressed. The highlighted column, and indeed any column, is simply a Gaussian and can thus be encoded compactly.

The matrix view of image filtering illustrates the following important facets of the image filtering problem: 1. Image filtering in general has $O(N^2)$ time complexity, because the matrix has N^2 entries. Note that N is the number of pixels in the image. If the filter kernels are small, most matrix entries are zero, meaning the matrix is sparse. This is why naive filtering is sufficiently fast for small filters. 2. We can equivalently describe any linear filter as spreading or gathering, by building the matrix and then taking either the rows or the columns; however, only the rows or the columns will be simple to describe (not both). 3. Fast gather filters operate by compactly representing the rows. 4. The columns of the spreading matrix are the same as the rows of the more traditional gather matrix, so the columns of the spreading matrix can be represented compactly. This suggests the possibility of fast spreading algorithms that operate on similar underlying principles as the fast gather filters.

Both spreading and gathering are straightforward to implement in software and on graphics hardware. Spreading simply involves accumulating filter kernels of various intensities. Spreading can be implemented on graphics hardware by rendering a vast quantity of point sprites, with alpha blending turned on and configured as additive blending. Gathering, on the other hand, simply involves iterating and accumulating over each pixel within the support of the filter kernel. Gathering can be implemented on graphics hardware by sampling the input image using texture units. Unfortunately, these straightforward implementations are $O(N^2)$, as they inherently must be if they correspond to multiplication by an N by N matrix. When the blur is small, the matrix is sparse, so the proportionality constant is significantly less than 1. Naive spreading or gathering is thus acceptable when blur is small. When blur is large, the matrix is not sparse, and the proportionality constant approaches 1. Large blur therefore necessitates accelerated filters that operate in constant-time with respect to blur size.

The literature contains a variety of constant-time image filters, such as [Williams 1983] and [Crow 1984]. These filters were intended for texture map downsampling, an application which clearly requires filters to be of this gathering type, taking weighted averages of the texture over the region that falls under a given pixel. These methods can be thought of as utilizing a compressed representation of the blur matrix, where each row is encoded very compactly as a position and size.

In this paper, we describe how to construct fast spreading filters, i.e. methods that compactly represent the columns, rather than the rows of the matrix.

3 Background

Several researchers have described research related to our work. Perlin [Perlin 1985] describes the benefits of using repeated box filters for image filtering, and Heckbert [Heckbert 1986] describes a technique that uses the idea of spreading to implement *spatially invariant* filtering with a limited-memory footprint. Although these both have the notion of reversing the filtering operation (*spreading* versus *gathering*), neither discuss the superiority of spreading for certain applications such as depth of field and motion blur, and neither develop the reduced precision requirements in any detail.

To implement filter spreading as a constant-time filter we use summed-area tables, introduced by Crow [Crow 1984]. Once generated, a summed-area table provides a means to evaluate a spatially varying box filter in a constant number of texture reads. Additionally, we use the work of Heckbert [Heckbert 1986], who extended Crow's work to handle complex filter functions via repeated integration. As part of our implementation, we use the technique introduced by Hensley et al. [Hensley et al. 2005] to compute summed-area tables using GPUs. There have been other constant time filters before ours, beside Crow's and Heckbert's – these are all gather filters.

Mipmapping [Williams 1983] is a ubiquitous constant-time filter used in computer graphics. Several researchers [Ashikhmin and Ghosh 2002; Yang and Pollefeys 2003] have used a technique that combines multiple samples from mipmaps to approximate various generic filters. Using mipmaps in this fashion suffers from artifacts that are introduced since a small step does not necessarily introduce new data to the filter; it only changes the weights of the input values. These artifacts are compounded by the fact that small steps in another direction can introduce a large amount of new data. Both of these issues with mipmaps create noticeable artifacts when they are used to implement generic filters. As an example, the authors of [Demers 2004] jitter the sample position in an attempt to make the artifacts in implementation less noticeable, which resulted in a loss of visual fidelity. Most recently, [Lee et al. 2009] show how the quality of blurring using mipmaps can be improved by using 3x3, rather than 2x2 filters during mipmap construction, and circular sampling, rather than simple bilinear interpolation, while reading from the mipmap. This solution produces smooth blur in typical cases, though they don't describe any way of selecting alternative filter kernels.

Kraus and Strengert [Kraus and Strengert 2007] show that the MIPmapping hardware in GPUs can be cleverly used to achieve pyramidal image filters that do not suffer from the aforementioned artifacts. By carefully controlling the downsampling (analysis) filter, and by also using an appropriate pyramidal upsampling (synthesis) filter, bilinear, biquadratic, and even bicubic filters can be achieved in constant time. They describe a GPU implementation that achieves realtime frame rates. However, Kraus and Strengert's method provides only Gaussian-like filters, with no choice over point spread function.

Fournier and Fiume [Fournier and Fiume 1988] describe a constant time filter that operates by pre-convolving the input image with a cleverly-arranged hierarchy of polynomial basis functions. The image is divided uniformly into a coarse grid of boxes. Each box is considered as a vector, of which the dot product is computed with each polynomial basis function. The resulting scalars are stored in a table. This process is repeated for variously scaled boxes, resulting in a pyramidal table. This table contains all the information necessary to filter the image with arbitrary piecewise-polynomial filters in constant time. To compute a pixel of the output image, an appropriate level of the pyramid is first selected. Then, the desired filter kernel is approximated in a piecewise-polynomial fashion, aligned with the grid of table cells centered at the output pixel. Finally, the color of the output pixel is simply the linear combination of polynomial coefficients weighted by the pre-computed scalars.

Gotsman described in [Gotsman 1994] a constant time filter that also operates by pre-convolving the input image with a collection of basis functions. These basis functions are derived from the singular value decomposition (SVD) of the desired filter kernel family. The SVD ensures that the basis will be optimal in a least-squares sense. To find the color of an output pixel, all that is required is an appropriate linear combination of corresponding pixels from the pre-convolved image set.

While fast Fourier transforms, including highly optimized implementations such as FFTW [Frigo and Johnson 2005] are a standard signal processing tool for performing fast convolution, they are not applicable, because we desire filters that have a spatially varying filter kernel and are thus not convolutions.

Kass, Lefohn, and Owens [Kass et al. 2006] describe a constant time filter based on simulated heat diffusion. Color intensities are treated as heat, and desired blur is treated as thermal conductivity. An implicit integrator enables arbitrary blur size in constant time. Heat diffusion inherently produces Gaussian PSFs. Interestingly, heat diffusion is a form of blur that is neither gathering nor spreading.

Constant time filters generally have the same general structure: build a table of precomputed averages, then read sparsely from the table to determine blurred pixel colors. While constant time filters generally operate as gather filters, in principle they could all be turned into constant time spreading filters by reversing the order of operations. Specifically, constant time spreading involves first writing sparsely to a table, then constructing the blurred colors implied by the table.

We build our constant time spreading filter on the principles of Heckbert's repeated integration method, rather than one of the others, because only repeated integration enables arbitrary piecewise-polynomial kernels that do not need to be aligned with a grid. Furthermore, repeated integration is itself a simple and efficient process and can be implemented on the GPU, and so can be performed online rather than ahead of time.

Kosloff, Tao, and Barsky described a variety of fast image filters [Kosloff et al. 2009]. One of their filters spreads rectangles of constant intensity. The method described in the present paper spreads arbitrary polynomial PSFs. It should be noted that a rectangle of constant intensity is the simplest example of a polynomial PSF. Therefore the present method can emulate the method of [Kosloff et al. 2009] as a special case.

4 The Mathematics of Fast Filter Spreading

This section describes the mathematics underlying our method. These mathematics are similar to those used by Heckbert, but we present them here in the context of spreading.

For each input pixel, we want to spread a filter kernel into the output image. The kernel can vary in shape and size for each input pixel. In the general case, this implies that filtering requires $O(n)$ time per pixel, where n is the area of the filter. In terms of the matrix, filter spreading involves accumulating the columns of the blur matrix into the output image. The columns have size $O(n)$, leading to the aforementioned $O(n)$ time per pixel. This corresponds to filtering a signal f with filter g, i.e. $f * g$. By working only with piecewise-polynomial kernels, we can reduce this to $O(1)$ time. Consider, for example, a filter that is a constant-intensity rectangle. We can describe a rectangle by merely specifying its four corners, which takes

$O(1)$ time to do, independent of the size of the rectangle. Likewise we can describe polynomial filters by specifying the coefficients, which again are $O(1)$ with respect to filter size, though in this case the constant is proportional to the degree of the polynomial. Compactly describing the filter corresponds to compactly describing the columns of the blur matrix. We describe the mathematics primarily for a 1D signal, but application to a 2D image is straightforward.

Rather than spreading the filter kernel itself, we spread the derivative of the kernel. Observe that the derivative of a (discrete) 1D box filter is simply two deltas, one at the left end of the box and one at the right. The result of filtering with deltas is a signal that is the derivative of the signal we want. Therefore we integrate to generate our final signal. If instead we wish to spread Nth order polynomials, we spread the Nth derivative of the kernel, and integrate N times. This entire process can be expressed compactly as $f * g = \int^n (f * \frac{d^n g}{dx^n})$ (see [Heckbert 1986] for a proof). Even though a direct computation of $f*g$ requires $O(n)$ time per sample, computing $\int^n (f * \frac{d^n g}{dx^n})$ is equivalent and only requires $O(1)$ time per sample, because $\frac{d^n g}{dx^n}$ is sparse and integration is a constant-time per sample operation. For an illustration of repeated integration by filter spreading, see Figure 3.

Equivalently, we can describe our spreading technique in terms of a matrix factorization. Our original matrix, which is large and dense, is factored into two simpler matrices. One of these simpler matrices is sparse, as it contains only the derivatives of the filter kernels. The second simple matrix is triangular and contains values of 1.0, corresponding to the integration step. We mention this explicit factorization as it may be of theoretical use in the future and is useful for understanding our technique, but we will not rely on it in building our algorithm.

To apply this process to 2D images, we first construct our filters in 1D. We then create 2D filters as a tensor product of 1D filters. The 2D filter is spread, and integration proceeds first by rows, then by columns. Integrating by rows and then by columns works because summed area table construction is separable. The tensor-product nature of our filters is a restriction imposed by the separable nature of summed area tables. Although our filters are composed of arbitrary polynomials, these polynomials are defined by a rectangular parameterization. This precludes, for example, radially symmetric polynomials. The tensor-product restriction is less limiting than it may seem, however, because it is possible to construct filter kernels by placing separable polynomials adjacent to each other, leading to a piecewise-separable filter kernel. It may appear that introducing additional polynomial pieces will degrade performance. In practice, this is not the case; when more blocks are used, the order of the polynomials can be decreased, keeping costs approximately constant.

5 The Fast Filter Spreading via Repeated Integration Algorithm

The mathematics described in the previous section imply a straightforward, easy-to-implement algorithm. This section describes that algorithm and shows how to use it with a variety of filter kernels.

5.1 Algorithm

To blur an image using our method, we iterate over each pixel in the input image. A filter kernel is selected for that pixel, in an application-dependent manner. For example, in the case of depth-of-field postprocessing, the filter kernel is the point spread function of that pixel for the desired lens model. Deltas for that filter kernel are determined, using one of the methods described section 5.2. These deltas are accumulated into a buffer at locations centered around the pixel in question. After each pixel has been spread, the buffer is integrated to yield the final, blurred image.

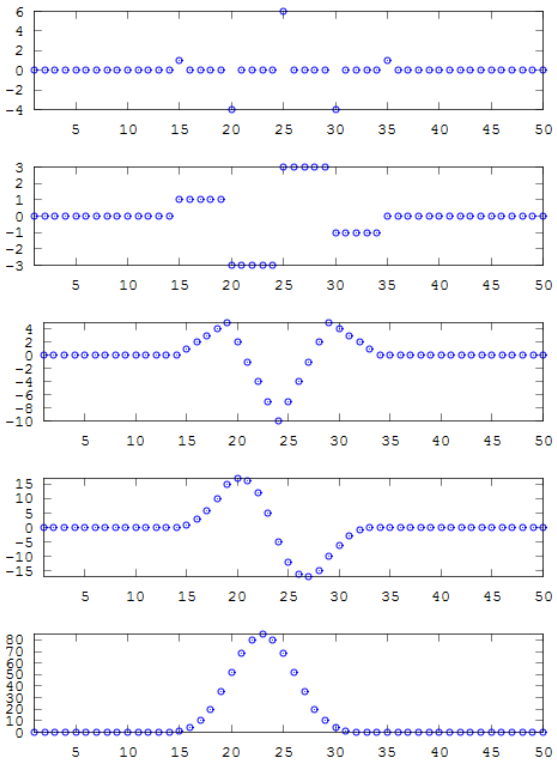


Figure 3: *Visualization of cubic repeated integration in 1D, for a single filter kernel. Top: the five deltas that constitute the fourth derivative of the filter kernel. Moving downwards, each image shows the results of successive rounds of integration, culminating in the filter kernel itself. Observe that each round of integration leads to a higher order polynomial, from dirac deltas to piecewise constant, through piecewise linear and piecewise quadratic, and finally piecewise cubic.*

The pseudocode in Figure 4 implements our algorithm. In this pseudocode, *input* is the input image, *buffer* is the buffer into which we accumulate, and *output* is the final, blurred image. The aforementioned images and buffer are two dimensional floating point arrays of size *width* by *height*. *delta* is a struct containing pixel coordinates *x* and *y*, and a floating point number *intensity*. For clarity of exposition, this code operates on monochromatic images. RGB images are easily and efficiently handled by slightly modifying this code to use RGB vectors and SIMD operations.

5.2 Filter Design

Our method is widely applicable to a range of image-processing operations because our method can apply a variety of different filter kernels. In this section, we describe how to construct deltas for those kernels, in order of increasing complexity.

5.2.1 Constant Intensity Rectangles

The simplest possible kernel for our method is a box filter. This is a first order method with deltas located at the four corners of the desired box. The intensity of all four corners is the same, and is

```
//Initialization:
//Clear all entries in buffer and output to 0.
//Phase I.
for(int i=0; i<width; i++)
for(int j=0; j<height;j++)
{


    //The filter size can vary per pixel
    //according to the application's needs.
    int filter_size = get_filter_size(i,j);
    //The deltas are a collection of
    //points centered about pixel location i,j.
    //Their positions are scaled to represent
    //filters of the requested size.
    //The positions and intensities
    //of the deltas are created
    //by differentiating the
    //desired kernel n times.

    delta d[] = make_deltas(i,j,filter_size);
    for k = 0:d.length
    {
    buffer[d[k].x][d[k].y] += d[k].intensity;
    }

}

//Phase II.
for(int i=0; i < order; i++)
for(int y=1; y < height;y++)
{
    accum = 0;
    for(int x=0; x<width; x++)
    {
    accum += buffer[x][y];
    output[x][y] = accum + output[x][y-1];
    }
}
```

Figure 4: *Pseudocode Implementation of Repeated Integration Spreading*

determined by the color of the pixel. At the top left, top right, bottom right, and bottom left corners, the signs are positive, negative, negative, and positive, respectively.

This filter is extremely fast and simple, but box filters are rarely the ideal choice of filter.

5.2.2 Repeated Box Filters

By convolving a box filter with itself N times, we achieve an Nth order B-spline basis function, which approximates a Gaussian. Nth order box filters are useful, for example, for image smoothing and depth-of-field postprocessing. See Figure 5 to see the effect of first, second, and third order repeated box filtering.

Instead of four corners, there will be a grid of $(order + 1) \times (order + 1)$ deltas distributed uniformly across the filter support. First, we evaluate $(-1)^i \binom{order}{i}$ to determine the signed intensities for a one dimensional filter, where i ranges from 0 to $order$. Next, we expand the vector of signed intensities into two dimensions by taking the outer product of the vector with itself.

A simple way to use Nth order box filters to approximate an arbitrary low-frequency filter kernel is to subsample the desired kernel onto, say, an $m \times m$ grid. We will then use an $m \times m$ grid of repeated box filters as our approximate filter kernel. Effectively, our repeated box filter is a reconstruction kernel, used to upsample the low-resolution kernel to the desired, possibly large size. This works well, because Gaussians make good reconstruction kernels.

5.2.3 Arbitrary Piecewise-Polynomials

In general, we can spread Nth order piecewise polynomials, using Nth order repeated integration. To generate the deltas, we differentiate the kernel N times. For well-behaved kernels that can be described by controlling only the Nth derivative, the process works smoothly. However, degenerate scenarios can occur, requiring lower order derivatives to be directly controlled as well. In such cases we must split our filter into lower order components and higher order components, filter each separately, and then combine.

5.3 Normalization

After delta spreading and integration, a normalization step is necessary to ensure that no unwanted brightening or dimming occurs. In a gather filter, normalization is simply a matter of dividing through by the integral of the kernel. In spreading, however, each pixel receives contributions from other pixels via an unknown and potentially complex set of kernels with various size and shape. To normalize, we apply our filter to an all-white normalization image (intensities 1.0 everywhere). We then divide the filtered input image by the filtered normalization image.

5.4 Precision Requirements

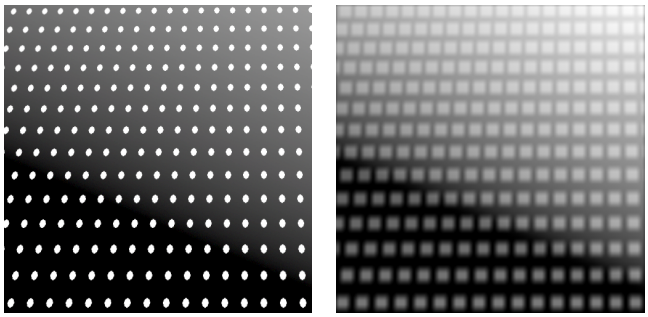
Unlike Heckbert's method, our method's precision requirements do not increase with image size. Rather, our precision requirements increase with filter size. The filter is generally much smaller than the image, so our method will generally require much less precision than Heckbert's method. To determine how much precision our method needs, consider integrating the accumulated delta image N times. After each round of integration, the values will have an increasingly large magnitude. We must have enough precision to represent the largest magnitude encountered. The largest value depends on the particular filter kernel being used and on the particular image being filtered, but here we will show the requirements for the repeated box filter on an all-white image (the worst case). Assuming 8 bits per channel, for the repeated box filter, we have $value = 255 * ((width - 1)/(n))^n$ in one dimension, and the square of that in two dimensions, where $width$ is the width of the filter, and n is the order.

For example, consider using our method with 32 bit two's complement integers, which can represent numbers as large as 2,147,483,647. In 1D, we can have filters as large as 611 pixels for 3rd order filtering, 216 pixels for 4th order, and 122 pixels for 5th order. This precision requirement is independent of image size. By comparison, Heckbert's repeated integration method requires 35 bits of precision for a 3rd order filter, 44 bits for 4th order, and 53 bits for 5th order, on a 1D image of size 512, regardless of filter size.

6 GPU Implementation Details

We have implemented repeated integration spreading on the GPU using DirectX 10.

We perform delta spreading via bufferless vertex rendering with alpha blending. To accumulate deltas, each delta is rendered as a



(a) Input Image (b) First-order spreading filter

(c) Second-order spreading filter (d) Third-order spreading filter

Figure 5: *An example of image blurring using filter spreading. Image (b) shows the effect using a first-order filter. Blocky artifacts can clearly be seen since a first order is simply a box filter. Image (c) shows the effect using a second-order filter. The amount of blockyness is greatly reduced since the spreading filter is a* Bartlett *filter. Image (d) shows the effect of using a third-order filter (a quadratic function). The third order filter dramatically reduces the artifacts of the first and second order filters.*

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 is used, configured to act as additive blending.

Any GPU implementation of SAT generation could be used for the repeated integration step. Currently we are using the recursive doubling approach [Hensley et al. 2005]. To repeat the integration, we can simply run the SAT generation routine iteratively, using the output of the previous iteration as the input to the next.

Roundoff error can accumulate quite significantly during the course of accumulating signed intensities, and during repeated integration. As the integration scan traverses the image, contributions from any given filter kernel should go to zero outside the region of support. Unfortunately, roundoff errors sometimes lead to leakage, as the signal does not quite return to zero in the least significant digits. One solution is to use integers rather than floating point numbers, because integers do not suffer from roundoff error. Unfortunately, the alpha blending hardware in today's GPUs only operates on floating point numbers. While this will be fixed in the next generation of GPUs, we use a workaround in the meantime.

Our workaround is to multiply the floating point numbers by a power of two (e.g. 2^8), to reduce the impact of roundoff. Multiplying by a power of 2 shifts zeros into the least significant bits, moving the important information into more significant bits that are less susceptible to roundoff. As a postprocess, we divide through by the same power of 2, removing the insignificant bits that were corrupted by roundoff error. A side effect of this workaround is that it wastes bits. Our method can potentially require many bits of precison when used for high order polynomials, so double precision computation is preferable. While GPUs support doubles, doubles are not currently available through DirectX. We find single precision to be enough at least through third order for the filter kernels we are interested in.

Our current implementation performs second order blurring on a 2.6ghz dual core Athlon with an ATI Radeon HD4870 GPU at 45 frames per second at a resolution of 800x600 . During a single frame, 8ms is spent spreading deltas, and 12ms is spent integrating.

7 Separable Spreading

It is well known that images can be blurred more efficiently if the blurring is performed in a separable manner. That is, it takes less time to blur a 2D image by first blurring the rows in 1D and then the columns in 1D, rather than blurring the image directly in 2D [Riguer et al. 2003][Zhou et al. 2007]. Our fast spreading filter can be made even faster by using it in a separable context. The resulting algorithm is faster than direct separable blurring and faster than our non-separable fast filter spreading method. Perhaps more importantly, because separable filtering operates in 1D, precision requirements are dramatically reduced. The largest number that must be representable is now only the square root of the largest number required for 2D filter spreading. This enables the use of large, higher order filters, without excessive precision requirements.

Unfortunately, during the second (vertical) step of separable spreading, colors from numerous pixels have already been combined. This means that no matter what filter size we use during the vertical step, some colors will be blurred either too much or too little. In practice, this only poses a problem when the amount of blur changes very rapidly from one pixel to the next. For applications where there is rapid spatial variation in blur, we must decide whether to use the more efficient separable blur, or the more accurate 2D blur. We feel that this is an interesting and useful tradeoff.

To partially mitigate the artifacts introduced by separable blurring, we blur the blur map, as well as the images, during the horizontal step. In other words, during horizontal blurring, information about how much the source pixels were meant to be blurred is propagated along with the colors. The vertical blurring thus receives a more accurate view of how much to blur each pixel.

See Figure 8 (e) and (f) for examples.

8 Results and Example Applications

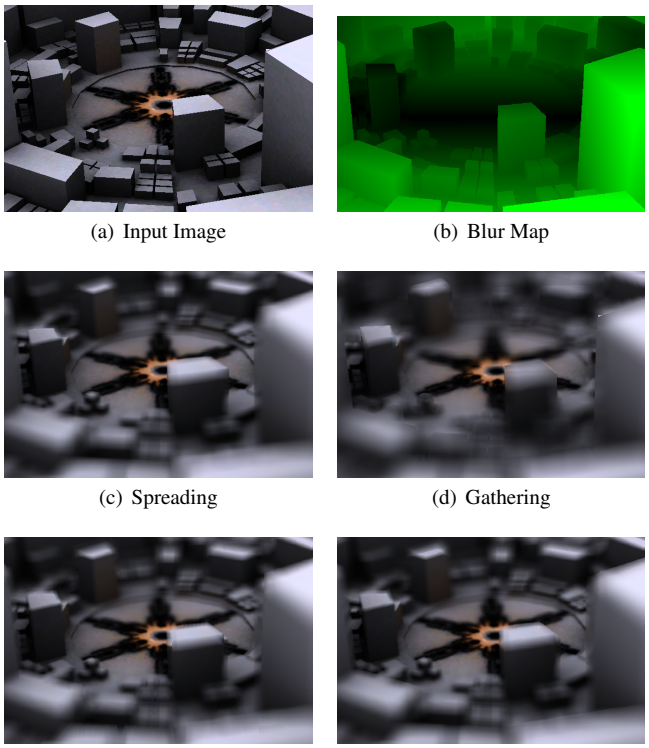
In this section we present several applications that can take advantage of filter spreading.

8.1 Depth of Field

Limited depth of field, i.e. the fact that only a subset of any given image is typically in focus, is an important effect for realistic computer generated images. The natural formulation for accurately simulating depth of field is to integrate across the aperture, generally by sampling at a number of discrete sample locations [Cook et al. 1984] [Haeberli and Akeley 1990]. Sampling the aperture is quite slow, so post-process methods can be used instead [Potmesil and Chakravarty 1982].

Our fast spreading filter is ideal for depth-of-field postprocessing.

Figure 6: *Lena blurred with a 5th order repeated box filter, a highly accurate approximation to a Gaussian.*



(a) Input Image (b) Blur Map

(c) Spreading (d) Gathering

(e) Separable Spreading Without Fix (f) Separable Spreading With Fix

Figure 8: *Various algorithms adding depth of field to the same image. Observe that our spreading method has fewer artifacts than the conventional gathering method (summed area tables). Our separable spreading method has lower precision requirements than our non-separable spreading method, but separable spreading introduces artifacts at the corners of objects. Blurring the blur map largely fixes the artifacts.*

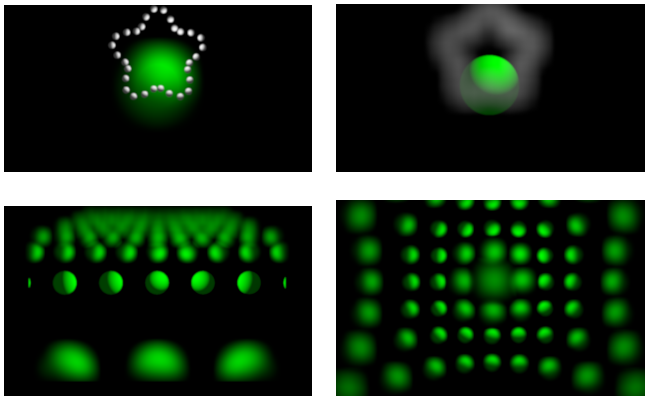


Figure 7: *Depth-of-field postprocessing. The top two images use layers to resolve visibility.*

At each pixel of the input image, we read depth from the depth map, and transform the depth value into the radius of the circle of confusion radius using a thin lens model. A filter kernel of size equal to the circle of confusion is then spread. This spreading approach to depth-of-field postprocessing, when combined with layer compositing as proposed by [Scofield 1994] and improved by [Lee et al. 2008], eliminates the intensity leakage and depth discontinuity artifacts common to depth of field postprocess methods [Demers 2004]. Results can be seen in Figure 7. The top two images in Figure 7 handle visibility using layering, and are thus free of artifacts. The bottom two images in Figure 7 do not use layers, and so do not strictly respect visibility. Therefore some artifacts can be seen where overlapping objects have vastly different blur levels.

In addition to reducing artifacts, via spreading, our depth of field method overcomes the precision requirements of Heckbert's repeated integration method, enabling us to use higher order filters on a GPU.

It is not always practical to modify a renderer to output layered images, so it is useful if a depth of field postprocess method can do a reasonable job in the non-layered case, even though some of the required pixels are occluded and thus not available. In Figure 8, we compare our spreading method, both in separable and non-separable form, with gathering via summed area tables. While some artifacts are unavoidable, spreading produces significantly fewer artifacts than gathering.

8.2 Motion Blur

Motion blur is another effect that makes computer generated images appears photorealistic. To accurately simulate motion blur, the computed image needs to be integrated over a discrete amount of time. Often, it is not practical to implement physically accurate motion blur, due to time constraints, so an approximation must be made. One common technique is to implement motion blur as a post-process.

While Heckbert proposes using repeated integration to implement motion blur, which is very similar to our approach, it is important to realize that our technique requires fewer bits of precision than his to get accurate results. This is because our technique is only dependent on the filter order and filter size, and is not dependent on input image size.

8.3 Edge Detection

Our method can be used to implement edge detection filters, simply by spreading the appropriate polynomial coefficients. For example, we can generate an Nth order approximation to a difference of Gaussians filter using an Nth order piecewise polynomial, requiring N integrations.

8.4 Other Blurring

Fast filter spreading can be used for any image blurring task, such as softening the edges of a photograph to draw attention to the center (Figure 6). Higher order filters, 5th order in this case, can be used to generate very high quality Gaussian blur.

9 Conclusions and Future Work

In this paper we have introduced a novel technique for filtering images that uses a *spreading* paradigm as opposed to the more commonly used *gather* paradigm. When using spatially invariant filters, both methodologies result in the same image, although the spreading implementation reduces the precision required when using a summed-area tables. When using spatially *varying* filters, the spreading paradigm results in a different, in some cases more useful, filtering operation than gathering. A depth of field effect is one example application where this difference can be exploited to produce images with high quality.

For future work, we plan to further develop other applications, such as soft shadow rendering, and develop a methodology to approximate arbitrary filter functions using constant time techniques such as repeated summed-area tables. In particular, we plan to bound the error associated with approximating an arbitrary function with a spreading filter of a set order. Additionally, we would like to further explore the relationship between the spreading filters and gathering filters, and how transformations from one space to the other space can be made.

Finally, we believe that exploiting the matrix structure of filtering operations is a fertile area for future work. We intend to examine other structures that we might find in the filter matrix, aside from simply rows and columns, as a way of finding new algorithms.

References

ASHIKHMIN, M., AND GHOSH, A. 2002. Simple blurry reflections with environment maps. *J. Graph. Tools 7*, 4, 3–8.

COOK, R., PORTER, T., AND CARPENTER, L. 1984. Distributed ray tracing. In *ACM SIGGRAPH 1984 Conference Proceedings*, 137–145.

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

DEMERS, J. 2004. *GPU Gems*. Addison Wesley, ch. "Depth of Field: A Survey of Techniques", 375–390.

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

FRIGO, M., AND JOHNSON, S. G. 2005. The design and implementation of fftw3. *Proceedings of the IEEE 93 (2) Special Issue on Program Generation, Optimization, and Platform Adaptation 93*, 2 (Feb.), 216–231.

GOTSMAN, C. 1994. Constant-time filtering by singular value decomposition. In *Computer Graphics Forum*, 153–163.

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

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

HENSLEY, J., SCHEUERMANN, T., COOMBE, G., SINGH, M., AND LASTRA, A. 2005. Fast summed-area table generation and its applications. In *Computer Graphics Forum*, vol. 24, 547–555.

KASS, M., LEFOHN, A., AND OWENS, J. 2006. Interactive depth of field. *Pixar Technical Memo 06-01* (January).

KOSLOFF, T. J., TAO, M. W., AND BARSKY, B. A. 2009. Depth of field postprocessing for layered scenes using constant-time rectangle spreading. In *Graphics Interface 2009*.

KRAUS, M., AND STRENGERT, M. 2007. Depth of field rendering by pyramidal image processing. In *Computer Graphics Forum 26(3)*.

LEE, S., KIM, G. J., AND CHOI, S. 2008. Real-time depth-of-field rendering using splatting on per-pixel layers. *Computer Graphics Forum 7*, 27, 1955–1962.

LEE, S., KIM, G. J., AND CHOI, S. 2009. Real-time depth-of-field rendering using anisotropically filtered mipmap interpolation. *IEEE Transactions on Visualization and Computer Graphics 3*, 15, 453–464.

PERLIN, K., 1985. State of the art in image synthesis. SIGGRAPH Course Notes.

POTMESIL, M., AND CHAKRAVARTY, I. 1982. Synthetic image generation with a lens and aperture camera model. In *ACM Transactions on Graphics 1(2)*, 85–108.

RIGUER, G., TATARCHUK, N., AND ISIDORO, J. 2003. *ShaderX2: Shader Programming Tips and Tricks with DirectX 9, W.F. Engel, ed.* Wordware, ch. 4, Real-Time Depth of Field Simulation, 529–556.

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

WILLIAMS, L. 1983. Pyramidal parametrics. *SIGGRAPH Comput. Graph. 17*, 3, 1–11.

YANG, R., AND POLLEFEYS, M. 2003. Multi-resolution real-time stereo on commodity graphics hardware. In *Conference on Computer Vision and Pattern Recognition*.

ZHOU, T., CHEN, J. X., AND PULLEN, M. 2007. Accurate depth of field simulation in real time. In *Computer Graphics Forum 26(1)*, 15–23.