Fast Image Filters for Depth of Field Post-Processing

Todd Jerome Kosloff

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2010-69
http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-69.html

May 13, 2010

Copyright © 2010, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Fast Image Filters for Depth of Field Post-Processing

by

Todd Jerome Kosloff

B.S. (University of Tulsa) 2003

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Brian A. Barsky, Chair
Carlo Sequin
Jonathan Shewchuk
Ming Gu
Richard F. Lyon

Spring 2010

Fast Image Filters for Depth of Field Post-Processing

Copyright 2010
by
Todd Jerome Kosloff

Abstract

Fast Image Filters for Depth of Field Post-Processing

by

Todd Jerome Kosloff
Doctor of Philosophy in Computer Science

University of California, Berkeley

Brian A. Barsky, Chair

The original and primary motivation for the work described in this thesis is depth of field post-processing. Previous methods for simulating depth of field were either too slow, or of low quality. Depth of field post-processing is a critical component of high quality rendering. Without depth of field, everything is in perfectly sharp focus, lending an unnatural, overly crisp look. In fact, lack of depth of field is an important tip-off that an image is computer generated. Depth of field is challenging to achieve when both high quality and high speed are desired simultaneously. This is because high quality methods are traditionally based on brute force, and fast methods make too many approximations. Intuitively, it seems odd that depth of field is so computationally expensive. Depth of field is a type of blur, and blur inherently removes information from an image, producing an output lacking high frequencies. Producing a simpler image should not be so expensive, and this thesis shows that, indeed, high quality depth of field blurring can be achieved in real time.

Central to this thesis are the concepts of gathering and spreading. Gathering is the process of forming an output pixel by taking a linear combination of input pixels. Spreading is the process of expanding each input pixel into a point spread function (PSF) of some kind, and accumulating those PSFs into the output image. Image filtering is traditionally thought of as gathering, but a central idea of this thesis is that depth of field (and a variety of other applications) is much better suited to spreading. In the course of developing the mathematical theory of filter spreading, we happened upon a new type of image filter that is equally well-described as gathering and spreading. This new method, which we refer to as the tensor filter, is conceptually simple and can leverage the benefits of any combination of gathering and spreading algorithms.

We have developed a variety of new fast, high quality image filtering algorithms. Except

for the tensor method, all of these are spreading methods. The reason why we have a variety of techniques is because there are a variety of PSFs that one may wish to use. Our methods achieve speed by exploiting structure in the PSFs. As such, truly arbitrary PSFs are problematic, but high quality results can be achieved with both polynomial and Gaussian PSFs, as well as PSFs that have an arbitrary outline but a constant-intensity interior.

While the original motivation for this work was depth of field, filter spreading actually has a variety of other applications. We have developed proof-of-concept applications for motion blur, implicit curves, and image warping. Filter spreading may also be useful for radial basis function evaluation and volume rendering.

Contents

List of Figures iii

1 Introduction 1
1.1 Overview . . . 1
1.2 Problems Addressed . . . 4
1.3 Problems Not Addressed . . . 4

2 Background 5
2.1 Depth of Field . . . 5
2.1.1 The Image Formation Process . . . 5
2.1.2 Post-Processing . . . 6
2.1.3 Bokeh . . . 6
2.2 Sampling Theory . . . 7
2.3 Spreading vs. Gathering . . . 7
2.3.1 Spreading . . . 7
2.3.2 Gathering . . . 8
2.4 Exploiting Kernel Structure . . . 11
2.5 Summary of Important Terms . . . 12

3 A Linear Algebra Theory of Image Filters 14
3.1 Introduction to Linear Filters . . . 14
3.2 Details Of Linear Filters . . . 17

4 Previous Work 19
4.1 Gathering . . . 19
4.2 Spreading . . . 21
4.3 Other Fast Image Filters . . . 22
4.4 Other Relevant Depth-of-Field Previous Work . . . 23

5 Details of Fast Gather Filters 27

6 Fast Spreading Filters 29
6.1 Normalization . . . 29
6.2 Rectangle Spreading . . . 29

6.2.1 Algorithm 29
6.2.2 Cost Analysis 31
6.3 Polynomial Spreading 32
6.3.1 Intuition 32
6.3.2 Algorithm 32
6.3.3 Impulse Response Design 33
6.3.4 Precision 35
6.3.5 Performance 36
6.4 Perimeter Spreading 36
6.4.1 Algorithm 36
6.4.2 Performance 38
6.5 Pyramid Spreading 38
6.5.1 Algorithm 38
6.5.2 Performance 44

7 Tensor Filter **45**
7.1 Algorithm 45
7.2 Performance 51

8 Comparisons **52**
8.1 Discussion Of The Relationship Between The Various Methods. 52
8.2 Visual Comparison Of Results 53

9 The AMD HD58xx Ladybug Launch Demo: A GPU Implementation Of The Polynomial Method **61**
9.1 Introduction 61
9.2 GPU Implementation of the Polynomial Method 61
9.3 Components Of The Demo 64

10 Conclusion **68**
10.1 Summary 68
10.2 Future Work 69

Bibliography **72**

List of Figures

2.1 Comparison of spreading vs. gathering. 7
2.2 When gathering is used, the effective PSF becomes distorted. The filter kernel in this example is a square (bottom inset) but the PSF can end up wildly different (top inset) from the desired square. Due to duality, this figure also illustrates the complex filter kernels required to force gathering to achieve the correct PSF. Under this interpretation, the bottom inset shows the PSF, and the top inset shows the necessary filter kernel. The complexity of these filter kernels precludes their use in fast gathering methods. 10

3.1 A convolution matrix. Multiplying this matrix by a vector convolves that vector with a Gaussian. 15
3.2 Matrix vector multiply can be implemented row-by-row. This corresponds to a gathering filter. 15
3.3 Matrix vector multiply can be implemented column-by-column. This corresponds to a spreading filter. 16

6.1 An example of fast rectangle spreading in action. 30
6.2 Tensor Product 31
6.3 Pseudocode implementation of the polynomial method. 33
6.4 The polynomial method requires only that these 9 values need to be written per PSF, for PSFs of any size 34
6.5 An example of how a sparse set of deltas becomes a smooth approximation to a Gaussian, after a few rounds of integration. 34
6.6 Illustration of the perimeter method for a circular PSF 37
6.7 When computing a subpixel Gaussian, the continuous distance between Gaussian center and pixel center is taken. It is critical that the Gaussian center is not snapped to a pixel location. 39
6.8 The pyramid method spreads PSFs to various pyramid levels, then upsamples the levels through the pyramid to yield a final, blurred image. 40

7.1 Using the SVD to find the best separable approximation of blocks within the matrix. From left right, the blocks are of increasingly small size. 46
7.2 A radially symmetric Gaussian can be factored into the outer product of two one dimensional Gaussians. 46

7.3 A blur matrix can be adequately reconstructed by Gaussians placed down the diagonal. For illustrative purposes, this example is for 1D blurring, i.e. we are only blurring horizontally. Left: the blur matrix constructed out of Gaussians down the diagonal. Center: Every other Gaussian removed, to make the remaining Gaussians more visible. Right: The result of blurring an image with the matrix on the left. . . 47
7.4 The tensor method for uniform blur is equivalent to downsampling followed by upsampling. 49
7.5 Illustration of the tensor method with a quadtree used to lay out the sites. 51

8.1 Various blur methods used on a low dynamic range image. Observe that each method produce similar result to the others. 54
8.2 Various blur methods used on a low dynamic range image. Observe that each method produce similar result to the others. 55
8.3 Various blur methods used on a high dynamic range image. Observe that the methods produce significantly different results. 56
8.4 Various blur methods used on a high dynamic range image. Observe that the methods produce significantly different results. 57
8.5 The results of using the perimeter method with various arbitrary-outline constant-intensity PSFs. 58
8.6 Various configurations of the tensor method. 59
8.7 Gathering vs. Spreading 60

9.1 Illustration of straightforward integration in DirectX 11. For clarity, this example is shown for a single row. In the full implementation, integration is performed for all rows in parallel, and then for all columns in parallel. Top: a simple input, chosen because the integral is trivial. Middle: Integration proceeds sequentially, from left to right. Bottom: result of integration. 63
9.2 We need more than one thread per row to fully utilize the GPU. Therefore we integrate simultaneously in four domains. Top: the four domains are highlighted. Middle: integration proceeds from left to right within each domain. Bottom: result of parallel integration. Observe that except for the first domain, we do not yet have the correct integral. A fixup step will be required. 63
9.3 The second, third, and fourth domains need to be fixed up. The fixup stage proceeds for each domain simply by adding the last value of the previous omain. The domains must be fixed up sequentially, but the fixup stage within each domain proceeds in parallel. Top: fixing up the second domain. Middle: fixing up the third domain. Bottom: fixing up the fourth domain. 64

10.1 A vision realistic PSF can be simplified considerably before noticeable changes in the blurred output occur. This gives hope that a reasonably simple polynomial can do a good job at approximating vision realistic PSFs. My polynomial method thus may be useful for accelerated vision realistic rendering. 69

Acknowledgments

I would like to thank the following people for their help in my research.

My students: Michael Tao, Arpad Kovacs, Lita Cho, Crystal Chang, Siyu Song, Edward Short, Diana Lozano Brian Sung Chul Choi, Griffin Foster, Hao Lin.

People at AMD: Justin Hensley, Jason Yang, Raja Koduri, David Hoff.

My committee: Brian A. Barsky, Carlo Sequin, Jonathan Shewchuk, Ming Gu, Richard F. Lyon.

Chapter 1

Introduction

1.1 Overview

Blur can be defined in many ways: a blurred image often lacks high frequencies, blurred images lack fine details, small objects cannot be seen after being blurred, because they are smeared out and blend in with their surroundings. Although computers can add blur to images, it is challenging to simultaneously achieve speed and quality. Speed means no more than a few seconds to blur an image if implemented on the CPU, or at least 30 frames per second when implemented on the GPU. Naive algorithms can take 30 seconds or more on the CPU, and fall far short of real time performance, even on the GPU. Quality means that the blurred image has the desired appearance, and this depends on the application. In general, we desire blurred images that are free of artifacts. Fast algorithms often introduce artifacts [71], as it is generally believed that quality must be sacrificed in order to achieve speed.

For depth of field postprocessing, quality means a pleasing or realistic point spread function (PSF). Having a pleasing point spread function is especially important for high contrast and high dynamic range (HDR) images. The point spread function can be seen very clearly in places where there is a bright point of light against a dark background. Similarly, strong edges will be blurred in a manner quite characteristic of the point spread function. For example, a disc-shaped PSF will lead to blurred edges appearing relatively abrupt, but a Gaussian PSF will lead to blurred edges fading out gradually. For motion blur, we also want an appropriate point spread function, ideally one that varies correctly to match the desired motion. In general, blurred images should look smooth. Another desired property is shift invariance. That is, if we are requesting a blur that is the same throughout an image, we should actually get that. Certain pyramid methods lack this property

[54], and an incorrect shimmering can be seen as objects move across the screen.

It is very easy to implement blur filters. It is even easy to achieve very high quality [67]. However, these simple, direct methods are very slow, unless the amount of blur is very small. For the special case where the amount of blur is the same, the filter is known as a convolution, and the Fast Fourier Transform can be used to accelerate it [27]. Filters have long been needed for texture map downsampling, to average all the texels that fall under a pixel [80]. This type of filter is referred to as a *gathering* filter [32][68]. Gathering filters have long been accelerated [80] [26] [29]. Unfortunately, gathering is the wrong approach for depth of field, motion blur, and other applications. *Spreading* (the process of expanding each input pixel into a point spread function and accumulating PSFs in the output image) is required instead [48], and spreading is a major emphasis of this thesis. Please see section 2.3 for a more detailed introduction to spreading and gathering.

Filter spreading is easy to implement [67], but it is not easy to implement to be fast. When the PSF is large, each input pixel makes contributions to a great many output pixels. Laboriously spreading to each of these output pixels, one by one, leads to very slow performance. It is reasonable at this point to question why any acceleration should be possible at all. Indeed, for the general case of arbitrary PSFs that vary arbitrarily across the image, no shortcuts are possible. A simple way to understand why performance is bound to be slow in the general case is to consider the sheer amount of input. All algorithms require at least the time to walk through all of their input, as a bare minimum. If each pixel has a different PSF and if each PSF is an arbitrary function, then these PSFs will form a vast amount of input to the blur algorithm. Even the simple task of briefly examining each pixel of each PSF will not possible in real time.

Fortunately, beautiful results can be achieved without resorting to arbitrary PSFs. The fast methods in this thesis exploit the underlying simplicity of common, useful PSFs to achieve speed, such as polynomial structure. If the PSF can be described as a piecewise polynomial with relatively few terms, then that PSF can be applied very efficiently. Piecewise polynomials can provide very good approximations of Gaussians; thus the polynomial method works well for applying Gaussian blur. We have also developed two other methods specifically for applying Gaussian blur. The first one uses a novel form of pyramidal image processing. The second of these methods exploits large-scale structure in the blur tensor. Another very useful class of PSFs, aside from polynomial and Gaussian, are those that have a complex outline, but with a constant-intensity interior. For example, PSFs that are shaped like circles or polygons are very common in real photographs. Therefore, we have developed a fast spreading method for blurring with this specific type of PSF.

Except for the tensor method, all these methods share a common approach. This approach

is first to spread the PSFs in compressed form, and then, after all of the spreading is done, to extract the final, blurred image via a postprocessing step. The compressed form of the PSF differs depending on which method is being used. The compressed form for the polynomial method is the derivative of the PSF. This leads to an efficient algorithm because the nth derivative of an nth order polynomial is a sparse set of deltas. The compressed form for the pyramid method is simply a low resolution representation of the PSF. Gaussians can be compressed in this way because they are band-limited and hence can be subsampled without loss of information. For the arbitrary-outline method, the compressed form is simply the outline. This method is much faster than naive spreading because the vast interior of the PSF is left implicit.

However, the tensor method is different. Rather than working with a compressed representation of the PSF, the tensor method works with a compressed representation of the blur operator. Since our image filters are linear operators, they can be represented as matrices. These matrices are huge and dense, which underlies the slow performance of naive methods. Because our matrix is acting on an image, rather than a vector, it is better to treat the matrix as an order-4 tensor. This tensor contains a great deal of structure, seeing as it is very smooth. It turns out that there is a way to compress the tensor, and then perform the blurring directly from the compressed representation. Although the resulting method is not as fast as the polynomial method, the tensor method achieves true Gaussian filtering, rather than making polynomial approximations.

During the development of these methods, it was not clear which method would be the fastest. After implementation, it became clear that the polynomial method is significantly faster than the others. Therefore we chose the polynomial method for GPU implementation. This GPU implementation was done in conjunction with AMD, as part of a launch demo for the ATI Radeon HD58xx line of video cards. This demo achieved real time performance by using the new features of DirectX 11, such as scatter and atomic addition.

This thesis is not a catalogue of equally useful methods. For example, the polynomial method is significantly faster, and thus more useful than the other methods. This thesis is description of the thought processes that went on while investigating fast blur algorithms. Beyond providing potentially useful algorithms, an important goal of this thesis is to lend insight into the nature of the space of blur algorithms, and the various ways in which acceleration can be achieved.

1.2 Problems Addressed

Given an input image and given a blur map, our filters transform the input image into a blurred image. A blur map is a monochromatic image of the same size as the input image where intensity corresponds to how much that pixel should be blurred. For depth of field post-processing, the blur map is created by combining the depth map with a lens model. By choosing an appropriate filter, different point spread functions can be achieved. Our methods are fast. In asymptotic terms, we usually mean constant time per pixel, independent of amount of blur. This can also be described as linear time per image with respect to number of input pixels. The traditional gathering filters were described as constant time per pixel, because they were used for texture map anti-aliasing, where we are interested in how long it takes to filter each texture mapped pixel. For blurring images, we find that linear time per image is a more appropriate viewpoint, because we are interested in how long it takes to blur an image.

In practical terms, our fastest method runs at real time in high resolution (upwards of 30 frames per second) in a GPU implementation. Our methods require between half a second per frame to a few seconds per frame in the CPU implementation.

1.3 Problems Not Addressed

Depth of field and motion blur cannot be fully solved simply with linear image filters, because of the nonlinear visibility term. To solve visibility in general requires expensive methods such as distributed or stochastic ray tracing [17] [21]. Fortunately, the visibility artifacts are relatively acceptable when filter spreading is used instead of gathering. We find that when filter spreading is used, the errors caused by ignoring visibility are not severe.

Another problem that is not addressed is the view-dependent shading inherent in real-world depth of field. A lens gathers a cone of light emitted from each scene point. The different rays within this cone will have different colors, unless the material properties are completely diffuse. Distributed ray tracing captures such view-dependent shading, but postprocessing methods such as the subject matter of this thesis do not. In practice, that means that our depth of field results have slight errors in the handling of specular highlights.

Chapter 2

Background

2.1 Depth of Field

In the real world, it is rare for images to be entirely in perfect focus. Photographs, films, and even the images formed on our retinas all have limited *depth of field*, where some parts of the image are in-focus, and other parts appear blurred. The term “depth of field” technically refers to the portion of the image that is in focus. In the computer graphics literature, the term “depth of field” is used to refer to the effect of having some objects in focus and others blurred. Computer generated images typically not have depth of field effects. This is because simulating depth of field requires significant computational resources.

2.1.1 The Image Formation Process

Typical rendering of computer generated images is based on a pinhole camera model. A pinhole camera model assumes that all light entering the camera must pass through a single point, or pinhole. Pinhole cameras allow only a very small amount of light to enter the camera, so they are of limited utility for real world cameras. Pinhole cameras are geometrically the simplest type of camera, and they are efficient to simulate. However, they lead to images where all objects are in perfect focus.

To allow more light to enter the camera, an aperture of some size is used, instead of a pinhole. A lens is needed, or else everything will be highly blurred. However, a lens has to be focused at a single depth. The light from objects located at any other distance from the lens will not converge to a single point, but will instead form a disk on the camera’s sensor or film. However,

sensors and film have a finite resolution. As long as the disk is smaller than a film grain or pixel of a sensor, the object will appear in perfect focus. However, when the disk is large enough to be resolved, blur occurs.

Distributed, or stochastic ray tracing is the natural way to simulate depth of field [17] [21]. Each pixel sends out a number of rays to various points on the aperture. These ray are refracted by the lens and enter the scene. Since this is a direct simulation of the image formation process, high quality depth of field effects fall out naturally. Unfortunately, a great many rays are required to adequately sample an aperture of moderate to large size, thus distributed ray tracing is a slow algorithm.

2.1.2 Post-Processing

Post-processing is the technique of using image processing to add depth of field to an image that was originally rendered with everything in perfect focus [67]. A depth map must accompany the image, to determine how much to blur each pixel. A variety of depth of field post-processing techniques are present in the literature, e.g. [67] [17] [69] [32] [46] [53] [54] [48] [75]. The work in this thesis is motivated because none of those methods simultaneously achieved enough quality and speed.

2.1.3 Bokeh

Bokeh is a Japanese word that photographers use to describe the appearance of the out-of-focus regions of a photograph. Different lenses and different shaped apertures lead to different bokeh. Bokeh is determined entirely by the PSF of the lens. Photographers consider some lenses to have good bokeh, and other lenses to have bad bokeh. A typical PSF that leads to good Bokeh is a disc of constant intensity, possibly soft-edged. A typical PSF that leads to bad Bokeh is a ring. A good depth of field algorithm will produce good bokeh, and an ideal depth of field algorithm would produce whatever arbitrary bokeh may be desired. Because fast depth of field algorithms achieve speed by using structured PSFs, care must be taken that the structure is appropriate. For example, a Gaussian PSF produces acceptable bokeh, but a square PSF does not.

2.2 Sampling Theory

When an image is blurred, we intuitively know that it contains less information compared to one that is in perfect focus. Indeed, a blurred image can be *subsampled* without loss of information. That is, we can reduce the resolution of the image without losing any details. The full-resolution blurred image can later be reconstructed from the samples. One consequence of sampling theory is that there is no need to laboriously render every pixel of a blurred image. We can instead render a reduced resolution image, and reconstruct. This is the basis for the pyramid method. Sampling theory will also be important for the tensor method.

2.3 Spreading vs. Gathering

(a) An image blurred by gathering. Notice the banding artifacts caused by depth quantization.

(b) An image blurred by spreading. Notice the lack of artifacts, even though it has the same quantization as (a).

Figure 2.1: Comparison of spreading vs. gathering.

2.3.1 Spreading

Spreading is a type of filter implemented as follows: for each input pixel, determine the point spread function, and place it in the output image, centered at the location of the input pixel. The point spread functions are summed to determine the output picture.

Spreading filters are appropriate for depth of field and motion blur, due to the image formation process. In the case of depth of field, each pixel in the input image can be roughly thought of as corresponding to a point in the input scene. Each point in the input image emits or reflects light, and that light spreads out in all directions. Some of that light enters the lens and is focused towards the film. This light forms the shape of a cone. If the point is in perfect focus, the apex of the cone will hit the image plane (film, retina, sensor, etc). When the apex of the cone hits the image plane, the scene point is imaged in perfect focus. When a different part of the cone hits the sensor, the point will image as a disk of some kind, i.e. a point spread function. The image formation process suggests that spreading is the right way to model depth of field. The filters in this thesis were motivated by the need for depth of field post-processing.

A bright point of light ought to image as a PSF with size and shape determined by that point. In a post-processing method, it is easy to simply spread each input pixel and obtain the desired effect. For a gather filter to handle PSFs correctly, it would have to consider that potentially any pixel in the input image could contribute to any output pixel, each with a different PSF. The effective filter kernel for this high quality gather method would have a complicated shape that depends on the scene. This complicated shape prevents acceleration methods from being effective.

The image formation process for motion blur works as follows: each point in the scene (pixel in the input image) moves through some path during the time when the shutter is open. This means that each input pixel draws out a path of light on the sensor. That is, the point spreads out into a path-shaped point spread function. Clearly, motion blur is naturally implemented as a spreading filter.

See Figure 2.1 (left) for an example of an image blurred by spreading.

2.3.2 Gathering

Gathering is a type of filter that is implemented as follows: the color of each output pixel is determined by taking a linear combination of input pixels, typically from the region surrounding the location of the desired output pixel. The input pixels are weighted according to an application-dependent filter kernel.

Gathering is the appropriate type of filter for texture map antialiasing, as will be clear when we examine how antialiasing works. Texture map antialiasing works as follows: when a textured object is viewed from a distance, the texture will appear relatively small on the screen. Several or even many texels will appear under each pixel in the rendered image. Since the texture

mapping is many to one, the appropriate thing to do is to average all the texels that fall under a pixel. The averaging should use appropriate weights, generally weights that fall off with distance. Clearly, texture map antialiasing is a gathering process, because each output pixel is found by averaging several input pixels. Gathering can be used to blur images, but the results will suffer from artifacts See Figure 2.1 (right) for an example of an image blurred by gathering. Observe the banding artifacts. Each band is a discrete region of blur. The continuous blur map is rounded to the nearest integers, to make the filters faster. With gathering, each output pixel is determined independent of the others, so the bands are easily visible. With spreading, the output pixels near the boundaries of the bands contain contributions from pixels on either side of the bands, so the bands are not visible.

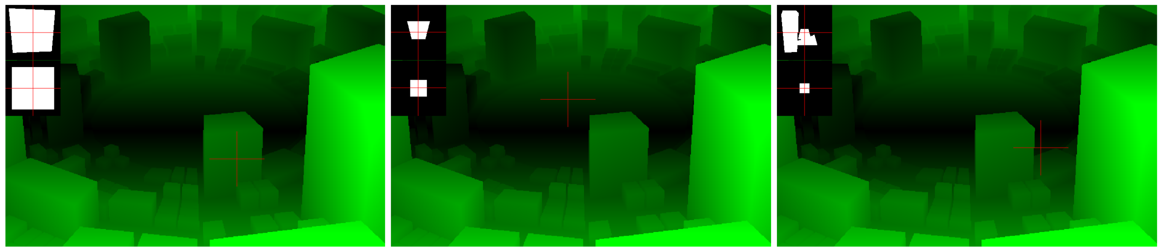
Gathering As an Approximation For Depth of Field

Fast gathering methods are common [26, 29, 80], but until the methods described in this thesis were introduced, fast spreading was virtually unknown. Therefore fast depth of field was implemented as gathering, because this was seen as the best way to make it fast. The approach to using gathering is simply to determine the PSF, but then use it as a filter kernel for gathering [36]. This is simple and fast, and indeed produces blurred images with a blur that varies according to depth. However, this generates incorrect results. Artifacts can be seen wherever the blur is variable from one pixel to the next. For high contrast images, artifacts can be seen even for smoothly varying blur. Bright points ought to image as perfect images of the PSF, but due to the use of gathering, the PSF appears distorted. The worst artifacts occur at depth discontinuities. Blurred foreground objects ought to have soft silhouettes, but when gathering is used, sharp discontinuities appear at the silhouettes.

Gathering As a Correct Solution For Depth of Field

Distributed ray tracing is usually implemented as backwards ray tracing, i.e. rays are traced from the input pixels, through the lens, and into the scene. Output pixels are thus created as linear combinations of points from the input scene. Clearly this is gathering, because each output pixel is determined as a weighted average of colors from the input scene. However, distributed ray tracing produces correct results, and we in fact consider it to be the gold standard by which post-processing is measured. So if gathering produces artifacts, how does ray tracing avoid the artifacts? Distributed ray tracing achieves correct results by taking a very complex linear combination. It is interesting to observe that a very complex set of weights is required to achieve the effect of spread-

ing. Ray tracing can pull in scene points from anywhere in the scene, in exactly the correct way. The downside is that this is very slow, and methods to accelerate it are inadequate. Each ray must be laboriously traced, and when the blur is large, a great many rays are required. Fast gathering methods, on the other hand, exploit simplicity in the filter kernel weights to achieve speed. Thus fast gathering methods are doomed to be inaccurate, and accurate gathering methods are doomed to be slow.



(a) A region with essentially uniform blur. Observe that the PSF and filter kernel are essentially the same.

(b) A region with smoothly varying blur. Observe that the PSF is distorted in a smoothly varying way.

(c) A region with discontinuous blur. Observe that the PSF is extremely complex and discontinuous.

Figure 2.2: When gathering is used, the effective PSF becomes distorted. The filter kernel in this example is a square (bottom inset) but the PSF can end up wildly different (top inset) from the desired square. Due to duality, this figure also illustrates the complex filter kernels required to force gathering to achieve the correct PSF. Under this interpretation, the bottom inset shows the PSF, and the top inset shows the necessary filter kernel. The complexity of these filter kernels precludes their use in fast gathering methods.

There is an interesting duality that occurs when gathering is substituted for spreading. We have already established that complex filter kernels are required for gathering to emulate spreading. Interestingly, complex PSFs are inadvertently introduced when simple filter kernels are used during gathering. These complex PSFs are essentially the same as the aforementioned complex weights. These complex PSFs are the distorted PSFs that we wish to avoid. See Figure 2.2 for an illustration of this duality.

Symmetric Filters

There exists a class of filters that we call symmetric. These filters are equally well described as spreading and gathering. For symmetric filters, the filter kernel is the same as the PSF. Spatially varying symmetric filters do not have particularly simple PSFs, nor do they have particularly simple filter kernels. However, as long as the filter is smoothly varying, very high performance

can be achieved through a very simple technique that we call the tensor method. Symmetric filters are an interesting and unexpected development of this research.

Any time the PSF is of the same size and shape throughout the entire image, we have a symmetric filter. Spatially-varying symmetric filters are not very intuitive, and are best understood in terms of the blur matrix. Simply put, we can construct a symmetric blur matrix by placing Gaussians down the diagonal. Another way to think of it is to treat the fundamental operation not as a gather or as a spread, but instead as a gather followed by a spread. That is, for each pixel, first gather some filter kernel, then use that filter kernel to spread. The symmetric case works best when the blur varies smoothly.

Converting Between Spreading And Gathering

Spreading and gathering are really just two different ways of looking at linear filters, whether by columns or by rows. A spreading matrix will have simple columns, but a gathering matrix will have simple rows. However, all matrices, whether spreading or gathering, have both columns and rows. If we were to construct the entire matrix, whether row by row or column by column, we could easily extract either rows or columns. In this manner, we can in theory convert between spreading and gathering. However, the conversion is not straightforward in practice, because the matrix is very large. For example, if the image under consideration is 1000x1000, then the matrix has dimensions 1000x1000x1000x1000. Building this matrix would be prohibitive both in time and space.

We can speculate about ways to carry out the conversion more efficiently, though we have not implemented any conversion method. We can imagine building the matrix directly in compressed form, using compressed rows or columns, and then reading compressed rows or columns out. As long as the compression is good enough, this approach could potentially be reasonably fast and efficient.

2.4 Exploiting Kernel Structure

The fast spreading methods achieve speed by working with compressed PSFs. First, a compressed form of the PSF is found. Next, the image is filtered with the compressed PSF to yield an intermediate buffer. Finally, the blurred image is constructed from the intermediate buffer by reversing the compression step. We can create a variety of different methods depending on what

compression scheme we choose.

One class of methods compresses the PSFs by taking their derivative. Clearly this only leads to fast methods when the derivative is sparse. Derivatives are simplest in 1D. Box functions have particularly simple derivatives, being nonzero only in two places. Piecewise-polynomial PSFs are also useful, as they can describe a variety of shapes, and become sparse when differentiated several times.

One way to differentiate a 2D function is to take a partial derivative. For example, we can take the horizontal derivative of a circle, yielding a function that is nonzero only along the perimeter. This is simple, but very large circles have very large perimeters, so the compressed PSF might not be sparse enough. Another way to differentiate a 2D function is to hope that it is separable. We can factor it into two vectors, then differentiate the vectors. This is very useful for polynomials, as tensor-product polynomials are a standard way to create 2D surfaces.

Another class of methods compresses the PSF by sampling it at the Nyquist limit. The PSF may be large, but so long as it is band-limited, we can express it compactly. This idea will lead to the pyramid method.

2.5 Summary of Important Terms

Image filter: An image filter is any process that transforms one image into another. This is so general that very little can be said about it without specializing.

Linear filter: A linear filter is an image filter that determines each output pixel by taking a linear combination of input pixels. Linear algebra tells us that any linear filter can be described by a matrix action on a vector, where the filter is the matrix and the image is the vector.

Tensor: A tensor is a matrix that can have order other than 2. Specifically, we will make use of an order 4 tensor. This is so that we can consider our image as an order 2 vector, i.e. we maintain the 2D structure. Tensor multiplication can be easily defined by analogy with how ordinary matrix-vector multiply works.

Blur filter: Blur filters are a type of image filter that makes images look out-of-focus. In more technical terms, blur filters attenuate high frequencies.

Point spread function: Point spread function (PSF) is another word for impulse response. The PSF is the image of a single pixel after having been filtered. If we know the PSF for ever pixel, we have fully defined a linear filter.

Filter kernel: Seeing how each output pixel of a linear filter can be described as a linear

combination of input pixels, we can define a filter in terms of what weights the linear combination uses. The set of weights needed to determine a single output pixel is that pixel's filter kernel.

Spreading: Spreading filters are the type of linear filter where each input pixel is expanded into a point spread function, and the point spread function are accumulated into the output image. Spreading is the appropriate type of filter for depth of field, motion blur, and other applications.

Gathering: Gathering filters are the type of linear filter that is best described as computing output pixels by taking a linear combination of input pixels. Note that gathering filters are appropriate for texture mapping downsampling.

Chapter 3

A Linear Algebra Theory of Image Filters

3.1 Introduction to Linear Filters

We take a linear algebra approach to analyze and discover blurring acceleration techniques. Consider the all-in-focus image to be a vector, and consider the blurring operation to be a matrix acting on that vector. The matrix is a function of the depth map, the focus and aperture parameters, and the desired PSFs. For an image of N pixels, the resulting matrix has N^2 entries. Except in the case of only small amounts of blur, enough of these N^2 entries are nonzero that direct application of the matrix leads to very slow performance. See Figure 3.1 for an example of a blur matrix (a convolution, in this case). Fast blur methods can be found by analyzing structure in the rows, columns, or blocks of this matrix.

In the above discussion, an image was considered a vector, ignoring the important issue of how to convert an image into a 1D vector. There are several different ways to treat an image as a vector, we could stack the rows, or stack the columns, or we could keep the 2D structure intact, and consider the linear operator as an order-4 tensor. The latter option is best, as it preserves the structure of the operator, in that adjacent elements always correspond to adjacent pixels. For clarity, we will generally present in terms of the order-2 matrix, operating on a 1D image. However, the insights apply to the order-4, 2D case in a straightforward way.

Matrix-vector multiply can be understood as operating row-by-row (Figure 3.2) or column-by-column (Figure 3.3). In the row approach, each element of the output vector is computed as a

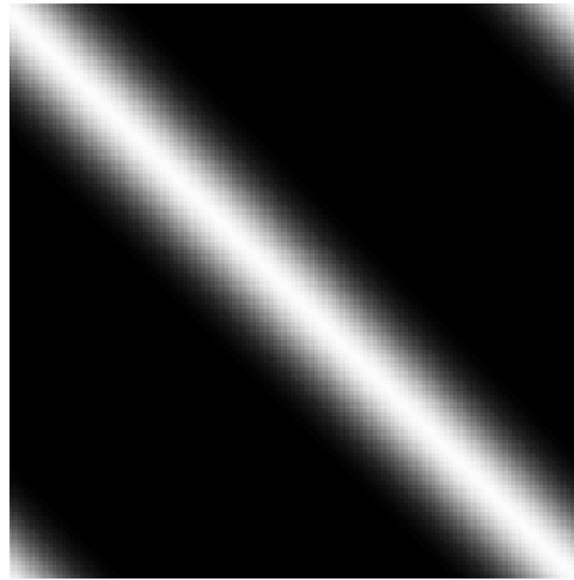


Figure 3.1: A convolution matrix. Multiplying this matrix by a vector convolves that vector with a Gaussian.

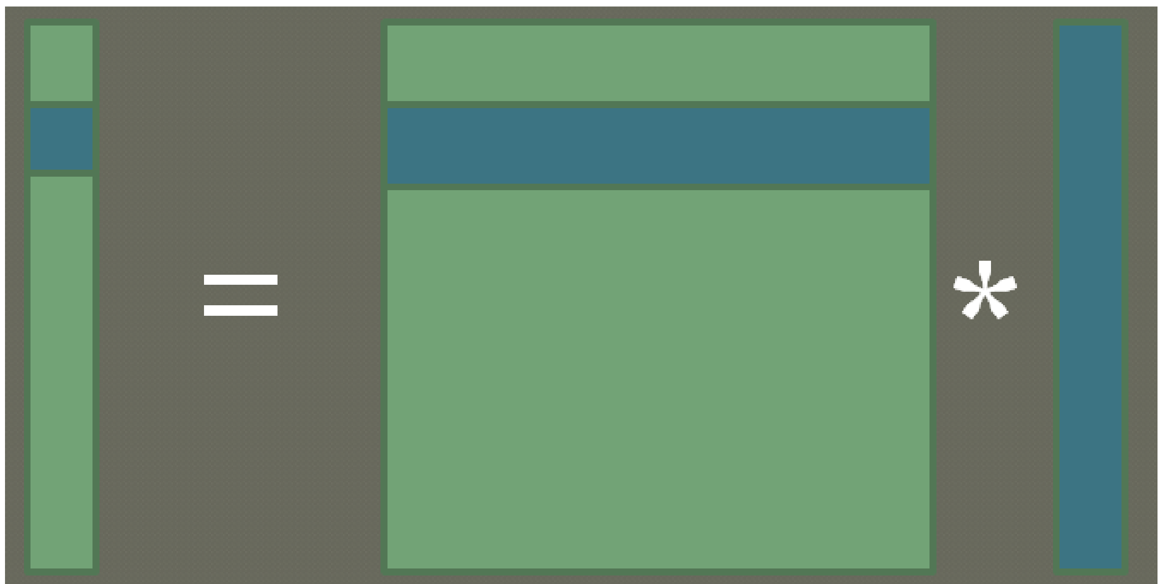


Figure 3.2: Matrix vector multiply can be implemented row-by-row. This corresponds to a gathering filter.

linear combination elements of the input vector, weighted by a row of the matrix. In the column approach, the output vector is a linear combination of the columns of the matrix, weighted by the elements of the input vector. The row approach corresponds to gathering, and the column approach

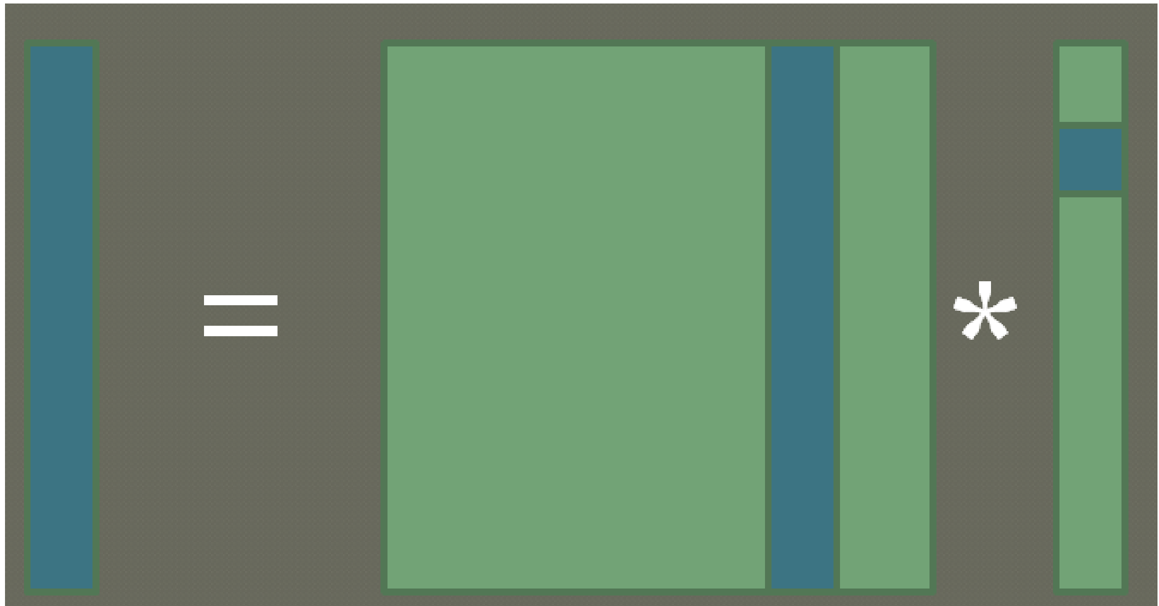


Figure 3.3: Matrix vector multiply can be implemented column-by-column. This corresponds to a spreading filter.

corresponds to spreading.

Much work has been done on fast blurring for antialiasing minimized texture maps. These methods work by exploiting structure in the filter kernel used for averaging. The filter kernels correspond to the rows of the blur matrix, though this fact is generally not discussed. These methods can be seen as achieving speed by representing the matrix compactly via simple, high-level descriptions of the rows.

Texture filtering methods work well for texture mapping, but unfortunately don't serve well for depth of field. The rows in a depth of field blurring matrix are incoherent, because PSFs lie down the columns, rather than the rows. Attempts to shoehorn the texture filter methods into depth of field rendering lead to artifacts, which are particularly visible at depth discontinuities.

The difference between the matrix built row-by-row and the matrix built column-by-column can be described as gathering and spreading. In the row matrix, input pixels are averaged to produce an output. In the column matrix, input pixels are spread out amongst the surrounding region. In some cases, such as when the filter kernel or PSF is the same for each pixel, the matrix is symmetric, so this distinction vanishes.

3.2 Details Of Linear Filters

Let I be an image that we wish to blur (input). Let O be the blurred image (output). In general, linear image filters can be written as follows:

$$O_{x,y} = \sum_{i,j} I_{x+i,y+j} \times \mathscr{K}_{x,y,i,j}$$

Each pixel in the output image is a linear combination of pixels in the input image. The input pixels are weighted with entries of $\mathscr{K}$. We will find it convenient to consider O and I to be matrices, and $\mathscr{K}$ to be an order-4 tensor. Assuming an appropriate definition for matrix-tensor multiplication, we can rewrite the filter as follows:

$$O = \mathscr{K} I$$

This formalization is very general, and we can design any type of linear filter by creating an appropriate $\mathscr{K}$. Visualizing and discussing $\mathscr{K}$ is difficult, because it is four dimensional. For purposes of discussion, we will make analogy to the two dimensional case, where O and I are vectors, and $\mathscr{K}$ is a matrix. This allows us to speak of rows and columns. We define the $(x,y)th$ "row" of $\mathscr{K}$ as the matrix found by allowing (i,j) to vary while keeping (x,y) fixed. We define the (i,j)th "column" of $\mathscr{K}$ as the matrix found by allowing (x,y) to vary while keeping (i,j) fixed.

Image processing often deals with convolution of an image with a 2D filter kernel. $\mathscr{K}$ represents a convolution when each row or column is a copy of the filter kernel, shifted by 1 relative to the previous row or column. It is interesting to note that if we construct $\mathscr{K}$ by placing the filter kernel in the rows, we get exactly the same $\mathscr{K}$ as if we placed the filter kernel in the columns.

This thesis is concerned with spatially-varying filters. Therefore $\mathscr{K}$ will be made up of different filter kernels, rather than translates of a single kernel. In the spatially-varying case, we get one $\mathscr{K}$ if we put the filter kernels in the columns, and a different $\mathscr{K}$ if we instead put the kernels in the rows. We will refer to $\mathscr{K}$ as a spreading filter if it was constructed by columns. We will refer to $\mathscr{K}$ as a gathering filter if it was constructed by rows. We will refer to the columns of a spreading filter as point spread functions (PSFs).

For a $width$ by $height$ image, $\mathscr{K}$ has dimensions $width$ by $height$ by $width$ by $height$. Therefore evaluating the matrix-tensor multiply is exceedingly expensive in general. Usually the PSFs are substantially smaller than the image, so $\mathscr{K}$ is often somewhat sparse. Direct filtering with small PSFs is therefore efficient. Direct filtering with moderate-to-large blurs is still quite expensive, even though a substantial fraction of $\mathscr{K}$ is zero. This is because $\mathscr{K}$ is extremely large, so if even a small fraction is nonzero, there are still very many nonzero entries.

This thesis introduces methods for efficiently evaluating the product $\mathscr{K} I$, even when the

PSFs are large and spatially-varying. We achieve speed by working with a compressed representation of $\mathscr{K}$. Not just any compression scheme will do, we need a form of compressed matrix where the product $\mathscr{K}I$ can be implemented without ever decompressing $\mathscr{K}$.

Broadly speaking, we consider three types of compression. The first type compresses the rows. Compressing the rows makes sense for gather filters; we simply represent the filter kernel in some compact form. Several examples of this first type of compression are known in the literature, these are the constant-time texture mapping anti-aliasing filters. The second type compresses the columns. Compressing the columns makes sense for spreading filters; we simply represent the PSF in some compact form. Spreading filters are desirable for creating blurred images, and are thus a major component of this thesis. The third type compresses blocks in $\mathscr{K}$. This makes sense when $\mathscr{K}$ is smooth. $\mathscr{K}$ will be smooth when the PSFs are smooth and the variation between adjacent PSFs is gradual.

Chapter 4

Previous Work

4.1 Gathering

Fast image filters of the gathering type have been extensively investigated, primarily for texture map antialiasing.

Antialiasing is an area of computer graphics that has important ideas in common with image filters. In fact, gathering filters were originally created to aid in texture map antialiasing. Antialiasing deals with the fact that a pixel can only have a single color, yet multiple objects often exist within a single pixel. The naive solution is just to take a single sample, typically at the center of the pixel. This leads to aliasing, which looks like flickering, jagged edges, and moiré patterns. Much work has been done to come up with solutions to the aliasing problem.

The most common antialiasing method is to take multiple samples inside the pixel. An important question is how to distribute the samples within the pixel. For a given number of samples, it is found that random (stochastic) distributions are better than uniformly spaced distributions. Among the random distributions, Dippe and Wold found that a Poisson disk distribution produces the best results [21]. The Poisson disk distribution is an example of what is known as a blue noise distribution. Blue noise means that the samples lack low frequency correlation, possessing only high frequencies. Blue noise is a desirable property because low frequencies in the distribution leads to artifacts. Point-sampling anti-aliasing methods can be used as gather methods for image filtering. That is, blue noise distributions can be used to gather samples from an input image in order to filter it. Stochastic sampling is not applicable to spreading, so we will not explore it further.

The fast spreading methods that this thesis is concerned with are in a sense duals of gathering filters. Gathering filters were originally intended for texture map antialiasing, but they

can also be used to blur images.

Crow's summed area table is a gathering method that enables arbitrarily large box filters in constant time [19]. First, a table is built from the image. Second, each output pixel is determined by adding and subtracting four values from the table. Summed area tables are fast and simple, but are limited to box filters, and support gathering only. This thesis is mostly about spreading filters, rather than gathering, and our goal is to be far more general than mere box filters.

Hensley described a real-time implementation of summed-area tables that runs on the GPU [36]. He uses a parallel technique called recursive doubling to split the operation up into multiple threads. He maps the algorithm onto the DirectX graphics pipeline to allow easy implementation. It is possible to use Hensley's method as one step in our fast rectangle spreading and fast polynomial spreading methods. Hensley's method, being a summed-area table method, is limited to rectangular or polynomial PSFs.

Heckbert extended summed area tables to enable gathering with arbitrary polynomial filter kernels [33]. Our polynomial PSF spreading method is related to Heckbert's method but ours is spreading whereas his is gathering. Heckbert did not address the spreading issues that this thesis does.

Williams introduced MIP-maps, a very popular gathering filter usually used for texture mapping [80]. A MIP-map is a pyramid, i.e. it contains multiple copies of the input image, each copy at an increasingly reduced size. Image pyramids have a long tradition in computer graphics, dating back at least to [2]. To gather a kernel of a given size using MIP-maps, color is read from an appropriate pyramid level. To get around the fact that the levels are discrete (and that the pixels are large in the reduced size images), trilinear interpolation is used. Unfortunately, MIP-maps are gathering-only, and the trilinear interpolation leads to a less-than-ideal filter kernel. We instead introduce spreading methods with flexible filter kernels.

Both Fournier and Fiume [26] and Gotsman [29] introduced high quality gather filters. While these two methods differ in details, they operate under essentially the same underlying principles. They achieve speed by precomputation. The simplest type of precomputation is to filter the image ahead of time, and then read from the filtered image at runtime. These two papers improve on the basic precomputation idea by prefiltering not with the actual desired filters, but rather with basis functions such as polynomials or Gaussians. These basis functions span a space of filters, and output images can be created by taking linear combinations of the prefiltered images. If the precomputation approach were reversed to turn it into a spreading method, the cost of these methods would be prohibitive for our purposes, as the precomputation is a time consuming process.

4.2 Spreading

Splatting is a technique originally introduced for volume rendering for medical imaging. Each sample in the 3D volume is rendered as a small blob, or splat [83]. When the samples are far apart, larger splats are used, and when the samples are close together, smaller splats are used. The purpose of using splats, rather than mere point samples is to fill the spaces in between samples, so that there are no holes in the output image. Splatting is very similar to filter spreading, as in both cases Gaussians are accumulated in the image plane. Splatting generally does not use acceleration techniques, so performance is slow if the splats are large and overlap substantially. We suggest that it is possible to accelerate splatting using our fast filter spreading techniques.

Krivanek described a method of introducing efficient depth of field rendering into a splatting framework [50]. The basic rendering framework is to display point-sample models via splatting. Depth of field is introduced by allowing the size of the splats to grow in size within the blurred regions. To prevent performance from degrading when splats are large, reduced resolution versions of the scene are used in the blurred regions. Krivanek's method is an object-space technique, but it bears some similarity to our tensor-based image filter.

Hierarchical splatting is a technique for performing volume rendering at interactive rates [51]. This technique achieves the real time rates by taking advantage of large regions of approximately homogeneous materials. When rendering such regions, there is no reason to draw a great number of small, overlapping splats. Instead, a fewer number of larger splats can be used. Additional speed can always be achieved, at the expense of some quality, by forcing this compression to happen, even in regions where it introduces error. Hierarchical splatting bears some similarity to our tensor technique, as both compress nearby splats into fewer, larger splats. However, hierarchical splatting is purely a volume rendering technique, whereas our methods are suitable more general use, both for volume rendering and for a variety of other applications.

Potmesil and Chakravarty were the first to describe depth of field postprocessing methods in computer graphics [67]. They used a filter spreading method, although they did not use the term filter spreading to describe what they were doing. They used a very detailed optical model to derive PSFs that include realistic complex diffraction fringes. Potmesil and Chakravarty used a direct spreading technique, with no acceleration. As such, their method is quite slow. One aim of this current thesis is to introduce faster algorithms for implementing methods such as Potmesil and Chakravarty's image filter.

Lee introduced several improvements to Potmesil and Chakravarty's method [53]. First,

they used layers to help resolve visibility. The did so in a novel way, using a technique they call per-pixel layers. As pixels are spread, each pixel of the splat is potentially written to a different layer, depending on the relative depths involved. Pixels closer to the front get written to a foreground layer, pixels written to the back get written to a background layer, and pixels in the middle get written to a midground layer. After spreading, the layers are composited to produce the final result. Lee's method is fairly slow, as they use a direct spreading method, unlike the fast methods of the current thesis. Our fast spreading methods may be of some use in accelerating Lee's layering scheme.

We have recently published our first paper that is explicitly about filter spreading [48]. In this paper, we introduce the notion that filter spreading can and should be accelerated via appropriate techniques. Specifically, we used a variant of summed area tables, modified to perform spreading. The basic inverted summed area table (SAT) method only works for rectangular PSFs. We extend this to allow two additional kinds of PSF. The first kind is that which has an arbitrary outline, but a constant-intensity interior. The arbitary outline introduces a quality/performance tradeoff, as this extended method is somewhat slower than the rectangle method. Our alternative extension allows hybridizing with arbitrary PSFs. In regions where the truly desired PSF is especially likely to be visible, we spread the truly desired PSF. In regions where contrast is low, the PSF cannot be seen clearly, so we use the fast rectangular PSF to achieve speed. In this paper we used a simple heuristic to determine at each pixel whether to use an accurate or whether to use a fast PSF.

In [47], Kosloff, Hensley, and Barsky have extended the fast filter spreading method to enable arbitrary polynomial PSFs. The technique uses the fact that the Nth derivative of an Nth order polynomial is sparse. The technique easily handles piecewise polynomials, and allows the use of more pieces and higher order to achieve smoother, more flexible PSFs. Lower order polynomials are useful for situations where speed and simplicity are paramount.

4.3 Other Fast Image Filters

The fast Gauss transform [77] is a method from the applied mathematics community for computing the sum of a great many Gaussians efficiently, with provable error bounds. The fast Gauss transforms works by binning the input into boxes and approximating the contents of each box with a Taylor expansion. The output is similarly represented. Various theorems are employed to transform the input boxes into the output boxes. Finally‘, the Taylor series are evaluated to extract the output. The fast Gauss transform can be used as an image filter for implementing Gaussian blur. Unfortunately, to produce reasonable image quality, it requires a great many terms in the Taylor

expansion, and so performance is inadequate compared to methods custom made for fast computer graphics.

The Fast Fourier transform (FFT) is a traditional signal processing tool [27]. A mathematical identity known as the convolution theorem shows that image filters can be applied efficiently with the help of the FFT. However, convolution is limited to filters that are spatially invariant, i.e. they have the same filter kernel for all pixels. Convolution is a useful tool, but for many applications, spatially varying filters are required, so the FFT is not a solution.

Recursive filters are a class of algorithms that efficiently compute each pixel by reusing previously computed pixels. Recursive filters can in principle work with a wide variety of filter kernels, but in practice care must be taken to ensure that the filter remains fast. Tan showed how to use recursive filters for Gaussian kernels, using several different formulations [78]. Piponi showed how to use recursive filters for polygonal shaped filter kernels, particularly hexagons [65]. Tan's method is strictly speaking neither spreading nor gathering, so it is not applicable to the applications of this proposal. Piponi's method comes in both a spreading and a gathering form, and is thus useful for our purposes. However, Piponi's method works only for polygonal PSFs, and is efficient only for polygons with a small number of edges. While Piponi's method is useful, we seek more general techniques.

Concurrent with the work described in this thesis, Piponi described the relationship between gathering and spreading filters, and showed how to convert from one to the other [65]. He observed that gathering and spreading are related via a matrix transpose. He then showed how the code that implements linear filters can be modified in a mechanical fashion to compute the transpose.

4.4 Other Relevant Depth-of-Field Previous Work

One common class of techniques for calculating depth of field involves working with the original 3D scene representation. We refer to these as object-space techniques, because they work directly with the underlying scene objects. Typical object space techniques include distributed ray tracing, accumulation buffer, and light field techniques. These method are outside the scope of this thesis, because such methods are not image filters at all. Please see Barsky's survey for more information on object-space techniques [7].

Much work on producing blurred images has been carried out in the name of depth of field postprocessing. Some of these methods require nothing more than images and depth maps, we refer to these as image space techniques [8]. Please refer to Demers' survey for a further review of

previous approaches to depth of field rendering [20].

Discretization, i.e. the partitioning of the continuous space of blur levels into a manageable set of bins, is an important issue in image filtering and depth of field [10] [9]. Discretization can improve performance and help simplify image filters, both the gathering type and the spreading type. Discretization must be done very carefully, or artifacts will result. Barsky's previous work showed that a nonlinear spacing of the bins is preferable, due to the logarithmic nature of the human visual system. Our previous work on discretization also contained work on occlusion. Occlusion is the phenomenon where foreground objects block the view of background objects. In the case of depth of field, occlusion can be very complicated to deal with correctly. Please see Barsky's previous work for a detailed discussion of occlusion, as occlusion is mostly outside the scope of the current thesis.

Generalized depth of field (gDOF) is the notion that there is no need for synthetic depth of field to be bound by the same physical and optical limitations that bound real depth of field. Specifically, gDOF allows effects such as tilted focus planes, nonplanar focus regions, and multiple non-adjacent in-focus regions. There are at least as many different ways of rendering gDOF as there are ways of rendering realistic depth of field. Some of these methods involve image filters, and are thus relevant to this thesis.

One method that works well involves simulated heat diffusion [46]. Pixel colors are treated as heat, and simulated heat flow leads to the appearance of blur. This method has the advantage that it produces very general results; an arbitrary blur field can be used, and heat diffusion leads to a reasonable-looking blurred output.

A different type of gDOF technique is based on distributed ray tracing. The novelty lies in using nonlinear rays, meaning the rays curve rather than simply follow straight lines. Nonlinear distributed ray tracing provides an interesting tradeoff, enabling higher quality than the heat diffusion method, at the expense of somewhat less flexibility.

Barsky used wavefront data from human patients to drive a depth of field simulation [6]. The wavefront data gives us the point spread functions for the patient's eyes. Those point spread functions were used to realistically blur real and synthetic images. Speed was not the issue, and render times were measured in minutes. It may be possible to speed up vision realistic rendering using the fast filter spreading methods of this thesis.

A light field [55] or lumigraph [28] is a method for storing the rays of light that are present in a region of space. A light field captures the appearance of objects from many points of view. A light field can thus be used to render an object from various vantage points. Aside from being a

general-purpose image-based rendering tool, light fields are of particular interest to depth of field. If we capture all the rays of light that pass between the lens and the sensor of a camera, we have all the information we need to to render images with depth of field, with various focus and aperture parameters. Distributed ray tracing can be viewed as tracing all the rays in the lens/sensor light field, and then accumulating those rays on the sensor.

Shinya [74] described a depth of field postprocessing method that correctly handles visibility. Shinya's method involves creating a ray distribution buffer (what we now call a lens/sensor light field) for an image with a given depth map. When two rays land at the same position in the ray distribution buffer, visibility is handled via Z-buffering. Shinya's method achieves excellent results, but is very expensive in terms of memory and speed.

Kolb simulated particular systems of camera lenses, including aberrations and distortions, using distributed ray tracing [44]. Methods based on distributed ray tracing faithfully simulate geometric optics, but due to the number of rays required, are very slow. The accumulation buffer uses rasterization hardware instead of tracing rays, but also becomes very slow for large blurs, especially in complex scenes.

A realtime postprocess method was developed by Scheuermann and Tatarchuk, suitable for interactive applications such as video games [72]. 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 artifacts.

A more recent realtime depth of field postprocess method was developed by Kraus and Strengert, who used pyramids to perform fast uniform blurring [49]. By running the pyramid algorithm multiple times at different pyramid levels, they approximate a continuously varying blur. Bertalmio et al. showed that depth of field can be simulated as heat diffusion [11]. Later, Kass et al. used a GPU to solve the diffusion equation for depth of field in real time using an implicit method [43]. 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 used a fast pyramid method at the periphery, and a slower method with better PSF at the center of the image [60]. 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.

Zhou described a depth of field postprocessing method based on separable image filters [82]. Separable filters perform 2D filtering by first filtering horizontally, then vertically. When using direct filtering without acceleration, the separable approach is significantly faster than the non-separable approach. Note that separable image filters have performance that scales linearly with the width of the point spread function. Note that this is better than naive blurring, with performance that scales with the area, rather than the width. We aim, however, to produce constant-time image filters, whose performance does not degrade at all with increased filter size.

Rokita described a fast image filter that operates via repeated 3x3 gathering [69]. 3x3 gathering is fast and simple, and can be implemented in hardware. By repeatedly performing the convolution, Gaussian filters larger than 3x3 can be achieved. Rokita's method is fast and simple, but it is not a spreading method, and is limited to very smooth kernels such as Gaussians, due to the central limit theorem.

Lee created an improved gather filter for depth of field postprocessing by extending MIP-maps [54]. First, while creating the MIP-maps, 3x3 filters are used for downsampling, leading to enhanced filter quality compared to the usual 2x2 filters. Second, higher quality is achieved by reading more entries from the MIP-map, compared to what Williams originally proposed. The quality is enhanced compared to the original MIP-map formulation, but still falls short of other methods, even the simplistic summed-area tables. Our methods have higher quality than Lee's enhanced MIP-maps, and are spreading, rather than gathering.

Lee showed that raytracing, which is typically designed to take scene geometry as input, can in fact be used as a postprocess method. This technique treats the depth buffer as an image based scene representation. The depth buffer is essentially a heightfield, enabling efficient ray traversal. This technique can be considered a gathering method, because it gathers numerous rays to form a single pixel. However, the gathering produces similar results to spreading, because ray tracing is an accurate simulation of the optical image formation process. Lee's raytracing method is specifically designed to simulate depth of field, and is thus not applicable to the wide range of applications that our filter spreading techniques are designed for.

Chapter 5

Details of Fast Gather Filters

This thesis is primarily concerned with spreading filters, but it is necessary to first discus gather filters, to establish the basic steps of fast filtering. Fast gathering filters were invented first, and I construct fast spreading filter by reformulating the gather filters.

These methods are usually described as prefiltering; some kind of preprocessing is performed to build a table, then the table can be efficiently queried to retrieve blurred pixels. The simplest such table is simply an array of images, blurred with different filter kernels. If this array is computed ahead of time, blurred pixels can be directly read, assuming the desired filter kernel was included in the set. For kernels not in the set, we can take a linear combination of the available kernels. Gotsman's method works along these lines, using the SVD to find an optimal set of kernels to span the desired set. Prefiltering can be quite time consuming, and so we seek faster methods for real time use.

MIP-maps are a form of prefiltering where the prefiltered images are of reduced size. A half sized image is created by averaging 2x2 blocks. The half-sized image is recursively downsampled to form the MIP-maps. Creating the MIP-maps is much more efficient than prefiltering full-sized images, because the MIP-maps are small. Fast gathering can be performed by reading pixels from appropriate levels of the MIP-map, as pixels from the coarse levels contain averages over some region of the input. Trilinear interpolation is used when the desired pixel does not lie precisely on any pixel in the MIP-map. MIP-maps are very fast and are directly supported by graphics hardware, but the effective filter kernel is blocky and spatially-variant in an uncontrollable way, leading to ugly-looking blur.

Summed-area tables (SATs) [19] are a clever way of tabulating the sum of all rectangular regions within an image. A SAT has the same width and height as the image, but the precision

requirements are higher. Each pixel in the SAT stores the sum of all pixels in the entire rectangular region above and to the left of the pixel. The table can be constructed incrementally and separably, such that only two quick passes through the image are required. To read the sum of an arbitrary rectangular region, all that is necessary is to read the four corners of that rectangle. The four corners must be added and subtracted appropriately to yield the correct sum.

Summed area tables are limiting because they only enable constant-intensity filter kernels. Repeated integration tables [33] are a generalization of summed area table that allows polynomial-valued filter kernels. The key insight is to realize that the summed area table is an integral of the image. We then can see that the four corners form the separable derivative of the rectangular filter kernel. The generalization is to perform multiple rounds of integration to yield a table. To read from the table, we then construct the separable Nth derivative of the filter kernel. This derivative tells us where to read from the table, and how to weight the values that are read. The integrals and derivatives cancel, yielding the correct blurred image. However, the overall process is much faster than direct filtering, because the derivatives of the filter kernel are sparse.

We developed a variant of summed area tables that enables filter kernels of arbitrary shape, but constant intensity. The table is built by integrating only horizontally. Rather than storing sums of areas, this table stores sums along scanlines. To read a shape of arbitrary outline, values must be read from the table at each pixel along the perimeter of the filter kernel. The ability to handle filter kernels with arbitrary outline is compelling, but performance suffers somewhat, as now the entire perimeter must be read, rather than just the four corners.

All prefiltering methods are gathering methods. This is because output pixels are generated one at a time, by reading the table. The table is indexed by the location of the desired pixel, and some description of how large and possibly what shape the filter kernel should be. The filter kernel that should be applied to the input image to produce an output pixel is by definition a gather kernel, corresponding to a row of $\mathscr{K}$.

All of these gathering methods share a common structure; build a table from the input image, and then perform a sparse gather on the table. The table is constructed such that the efficient sparse gather has the same effect that a slow, dense gather would have on the original image. We will show how to construct fast spreading filters by reversing the order of operations. That is, spreading filters work by first performing a sparse *spread* to a table, and then constructing a final, blurred image from that table. It is the nature of the table that a sparse spread will have the same effect that a slow, dense spread would have in a naive spreading algorithm.

Chapter 6

Fast Spreading Filters

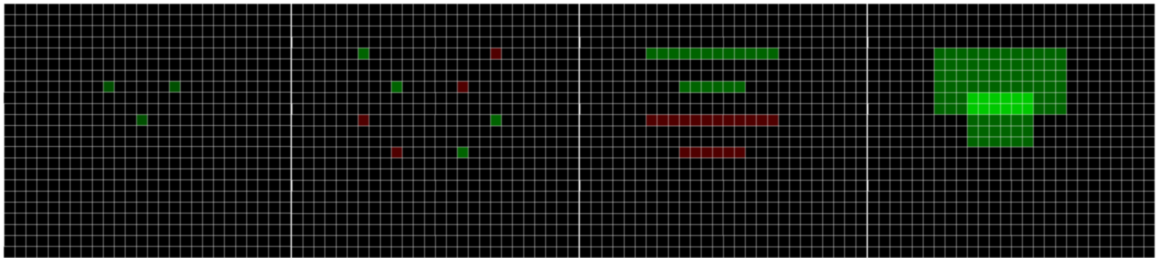
6.1 Normalization

All blur algorithms have the potential to produce overly bright and overly dim pixels unless special care is taken to normalize the filter. Normalization is simple in gather filters; simply divide the blurred pixel by the volume under the filter surface (integral). The analogous approach in spreading is to make sure that each PSF sums to one, by dividing through by the volume under the filter surface (integral). This is sufficient when all PSFs are the same, but when overlapping PSFs are different size or shape, hyper and hypo intensities show up at the overlap. Fixing this requires an additional normalization step; filter an additional color channel. This additional channel is initialized to have value 1.0 everywhere. After filtering is complete, normalize the colors by dividing by the result of filtering this additional channel. The spreading filters and the tensor filter in this thesis all require this extra normalization step. The normalization process is identical for all filters, so the details will not be repeated in the description of the individual algorithms.

6.2 Rectangle Spreading

6.2.1 Algorithm

The idea behind the rectangle spreading algorithm is that we will achieve speed by restricting our PSFs to rectangles of constant intensity. Rather than directly spreading the rectangles, we will just write accumulate the four corners of each rectangle to an intermediate buffer. Some of these corners will be positive, and others will be negative, to indicate whether we are entering



(a) A simplified input image with only three pixels. (b) The result of spreading the four corners of each pixel. (c) After integrating horizontally. (d) After integrating vertically, we have the final output image.

Figure 6.1: An example of fast rectangle spreading in action.

or leaving the rectangle. After writing these corners, a reconstruction step must be performed on the intermediate buffer to transform it into the final, blurred image. See Figure 6.1 for a simplified example. In this figure, green indicates positive values, and red indicates negative values.

Below is the pseudocode for the delta spreading phase.

```
intermediate_buffer = allocate_zeroed_array(width,height);

For x = 0 to width-1
For y = 0 to height-1
   {
   radius = blur_map[x][y];
   color  = input_image[x][y];

   top    = clamp(y - radius,0,height-1);
   bottom = clamp(y + radius,0,height-1);
   left   = clamp(x - radius,0,width-1);
   right  = clamp(x + radius,0,width-1);

   area = ( (radius*2+1)*(radius*2+1)  );

   intermediate_buffer[left][top] += color/area;
   intermediate_buffer[right][top] -= color/area;
   intermediate_buffer[left][bottom] -= color/area;
   intermediate_buffer[right][bottom] += color/area;
   }
```

After all pixels have been spread, transform the intermediate buffer into the blurred image by integrating the buffer horizontally, and then vertically. Below is the pseudocode for the

integration phase.

```
For y = 0 to height - 1
For x = 1 to width - 1
intermediate_buffer[x][y] += intermediate_buffer[x-1][y];

For x = 0 to width -1
For y = 1 to height - 1
intermediate_buffer[x][y] += intermediate_buffer[x][y-1];

output_image = intermediate_buffer;
```

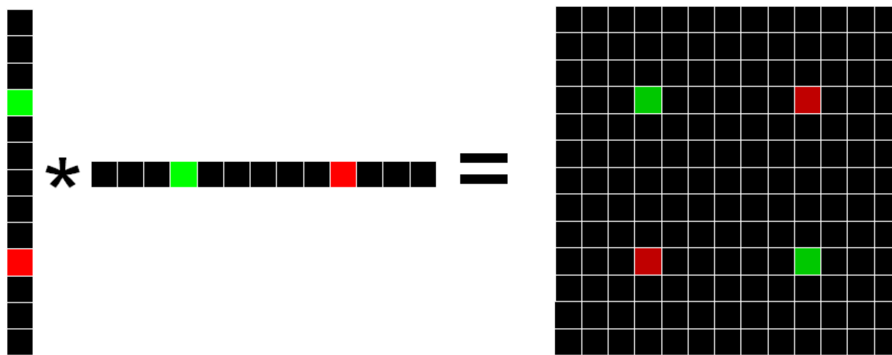


Figure 6.2: Tensor Product

To understand why the rectangle spreading method works, we must think in terms of integrals and derivatives, and it is easiest to work in 1D. If we take the derivative of a 1D box function, we find a positive impulse at the left end, and a negative delta at the right end. The fast rectangle spreading method can be viewed as first taking the derivative of the box filter, then filtering with that derivative, then finally integrating. This works because integrals, derivatives, and filtering are all linear. The 1D method gets extended into 2D by taking the tensor product. First, the four corner deltas can be derived by taking the tensor product of the 1D delta vector with itself (see Figure 6.2). Second, the integration is performed in a separable manner, first horizontally, then vertically.

6.2.2 Cost Analysis

The spreading step requires 4 writes per pixel. The integration step requires 1 write per pixel for the horizontal phase, and 1 write per pixel for the integration phase. Therefore the rectangle

spreading method has a cost of 6 writes per pixel. Observe that this cost is constant with respect to the amount of blur. Rectangle spreading is thus $O(1)$ per pixel, or $O(N)$ in the number of pixels.

6.3 Polynomial Spreading

6.3.1 Intuition

The polynomial spreading method achieves speed by using the fact that piecewise polynomial impulse responses become sparse after we take enough derivatives. We can view the polynomial method as a generalization of the rectangle spreading method, where rectangles are simply first order, constant-valued polynomials. For example, a constant-valued impulse response has a sparse derivative, a linearly-varying impulse response has a sparse second derivative, and a quadratic impulse response has a sparse third derivative. The method filters in real time by filtering with the sparse Nth derivative, rather than directly with the impulse response. The filtered image is then transformed into the correct output by taking the Nth integral.

To apply this process to 2D images, the filters are first constructed in 1D. Then, the 2D filters are created 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 the filters is a restriction imposed by the separable nature of summed area tables. 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 impulse responses by placing separable polynomials adjacent to each other, leading to a piecewise separable impulse response. 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.

6.3.2 Algorithm

To blur an image using the polynomial method, iterate over each pixel in the input image. An impulse response is selected for that pixel, in an application-dependent manner (see Figure 6.4). For example, in the case of depth-of-field postprocessing, the impulse response is the point spread function of that pixel for the desired lens model. Deltas for that impulse are determined, by taking the derivative until sparsity emerges. In practice, the sparse derivative is found directly without the need to ever explicitly construct the impulse response. 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 (see Figure 6.5).

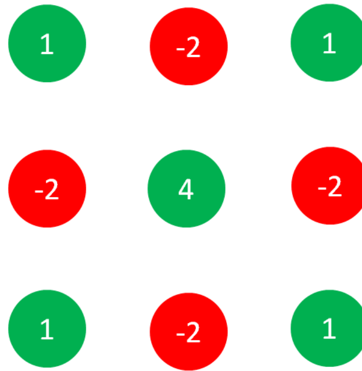
The pseudocode below implements our algorithm. In this pseudocode, *input* is the input image, *buffer* is the buffer into which we accumulate, and *output* is the final, blurred image. The aforementioned images and buffer are two dimensional floating point arrays of size *width* by *height*. *delta* is a structure 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 RGBA vectors and SIMD operations.

```
//Initialization:
//Clear all entries in buffer and output to 0.
//(clearing code omitted)
//Phase I: Spreading
for(int i=0; i<width; i++)
for(int j=0; j<height;j++)
    //The filter size is application-defined
    int filter_size = get_filter_size(i,j);

    //Section 6.2 describes make_deltas
    delta d[] = make_deltas(i,j,filter_size);
    for k = 0:d.length
        buffer[d[k].x][d[k].y] += d[k].intensity*
                                  input_color[i][j];

//Phase II: Integration
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 6.3: Pseudocode implementation of the polynomial method.

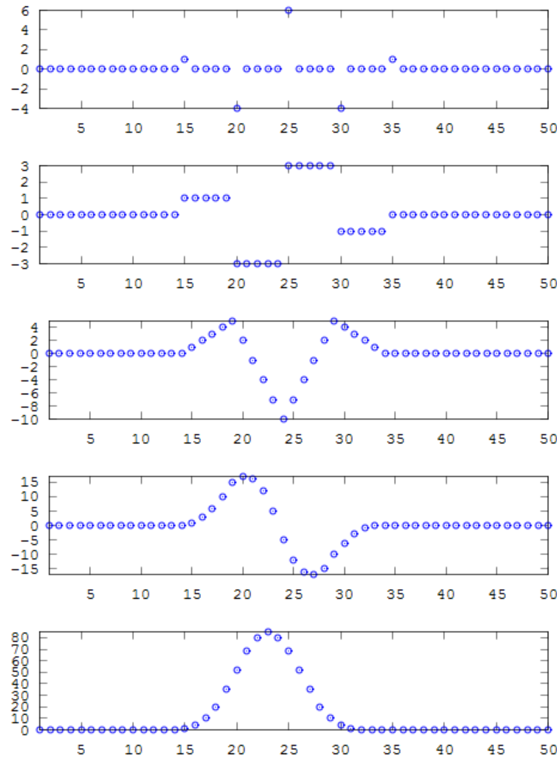
6.3.3 Impulse Response Design

The polynomial method is widely applicable to a range of image-processing operations because it can apply a variety of different impulse responses. In this section, we describe how to



(a) Deltas

Figure 6.4: The polynomial method requires only that these 9 values need to be written per PSF, for PSFs of any size



(a) Deltas

Figure 6.5: An example of how a sparse set of deltas becomes a smooth approximation to a Gaussian, after a few rounds of integration.

construct deltas for those impulse responses, without the need to first construct the impulse response and then differentiate.

The simplest possible impulse response for this method is a box. This is a simple example of the polynomial method and is exactly the same as the rectangle spreading method. 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 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.

By convolving a box filter with itself N times, it is possible 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.

Instead of four corners, there will be a grid of $(order+1) \times (order+1)$ deltas distributed uniformly across the filter support. First, evaluate $(-1)^i \binom{order}{i}$ to determine the signed intensities for a one dimensional filter, where i ranges from 0 to $order$. Next, 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 impulse response is to subsample the desired response onto, say, an $m \times m$ grid. Then use an $m \times m$ grid of repeated box filters as the approximate impulse response. Effectively, the repeated box filter is a reconstruction kernel, used to upsample the low-resolution impulse response to the desired, possibly large size. This works well, because Gaussians make good reconstruction kernels.

In general, it is possible to spread Nth order piecewise polynomials, using Nth order repeated integration. To generate the deltas, differentiate the polynomial N times. For "well-behaved" polynomials 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 it is necessary to split the filter into lower order components and higher order components, filter each separately, and then combine.

6.3.4 Precision

The polynomial method can require a great deal of precision in the variable type used for accumulating deltas and performing integration. It is critical that no bits of precision are lost during any of the operations, because otherwise intensities will leak, leading to unacceptable artifacts. This becomes problematic because during the course of integration, the values involved can become

quite large. For high order polynomials, integration can cause overflow, rendering the algorithm useless. I find that single precision floating point numbers are sufficient for second order integration. Double precision polynomials can work at much higher order. Doubles should be sufficient for any reasonable application.

6.3.5 Performance

The performance of the polynomial method depends on the order of the polynomial used, and on the specific PSF being used. We found that the most useful PSF for this method is an approximation of Gaussians. The number of deltas in this case is $(order+1)^2$. I found second order polynomials to have the right balance of performance and quality, so 9 deltas need to be spread, incurring 9 writes.

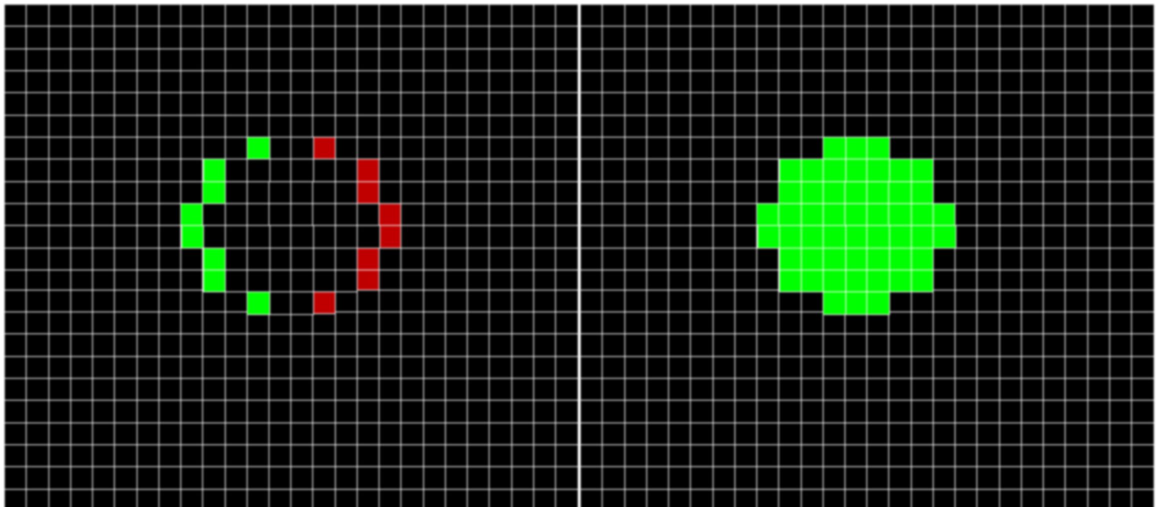
Multiple integration proceeds by integrating one time after another. Since each integration step has a horizontal and a vertical step, the total cost of integration is $2 \times order$ per pixel. Therefore second order polynomials require 4 writes for integration. In total, the second order polynomial requires 9 writes to spread the deltas, and 4 writes during integration, for a total of 13 writes. Observe that the polynomial method is a constant-time filter, because no matter what order we choose, performance does not depend on filter size.

6.4 Perimeter Spreading

6.4.1 Algorithm

PSFs in actual photographs are rarely if ever Gaussian, and certainly not boxes. In fact, real PSFs commonly are circular or polygonal, with a roughly constant intensity. Piecewise polynomial is not particularly convenient for such cases. PSFs of arbitrary perimeter can be achieved with a variation of the rectangle spreading method. Simply write deltas at the left and right border of the PSF, during the spreading phase, then integrate only horizontally. See Figure 6.6 for an illustration of how this process works. The deltas on the left have a positive sign, and the deltas on the right have a negative sign. This is much more expensive than the rectangular approach, as cost grows linearly with PSF size, rather than stays constant. This cost is unavoidable, however, as it is impossible to describe an arbitrarily-shaped PSF any more compactly.

Like the previous methods, this method can be understood as working with integrals and derivatives. We are simply working purely horizontally, obviating the need for the tensor product.



(a) The deltas for circular PSF.

(b) The result of performing horizontal integration on the deltas.

Figure 6.6: Illustration of the perimeter method for a circular PSF

The deltas at the left and right edge of a PSF are obtained simply by taking the horizontal derivative of the PSF. The horizontal integral undoes the differentiation, much like the integration step in the polynomial method. It is important to note that the differentation must be performed ahead of time, with the deltas stored in a table. This amortizes the time-consuming process of scanning through the PSF to find the deltas.

Below is the pseudocode that implements perimeter spreading.

```
//Preprocess the PSF at each scale (offline)
For each desired PSF radius r
scaled_psf = scale PSF to radius r;
i = 0;
For x = 1 to scaled_width-1
For y = 0 to scaled_height-1
difference = scaled_psf[x][y] - scaled_PSf[x-1][y];
if (difference != 0)
deltas[r][i].value = difference;
deltas[r][i].x = x - scaled_width/2;
deltas[r][i].y = y - scaled_height/2;
i++
deltas[r].count = i;

//At runtime
```

```
intermediate_buffer = allocate_zeroed_array(width,height);

For x = 1 to width - 1
For y = 1 to height- 1
 radius = blur_map[x][y];
 color  = input_image[x][y];
 For i = 0 to deltas[r].count - 1
u = clamp(x + deltas[r][i].x,0,width-1);
   v = clamp(y + deltas[r][i].y,0,height-1);
intermediate_buffer[u][v] += color*detas[r][i].value


For y = 0 to height - 1
For x = 1 to width - 1
intermediate_buffer[x][y] += intermediate_buffer[x-1][y];


output_image = intermediate_buffer;
```

6.4.2 Performance

The cost of spreading is 2 writes per row of the PSF. Therefore for a PSF of height H, $2 \times H$ writes are required for the spreading phase. The integration phase, being horizontal only, requires just a single write per pixel. Observe that the perimeter spreading method is not a constant-time filter, because cost does depend on the height of the PSF. Still, linear in height is a lot better than the naive method, which is linear in the area of the PSF.

6.5 Pyramid Spreading

6.5.1 Algorithm

Our next fast spreading method is based on pyramids, and can be viewed as running MIP-mapping [**?**] in reverse. The pyramid has its final level set to the resolution of the image. First, the pyramid is initialized to all zeros. Next, each pixel in the input image is spread by selecting a level of the pyramid, and writing a PSF to that level. Larger PSFs are written to coarser levels, thereby keeping the cost approximately irrespective of the PSF size. It is straightforward to outfit this method with a speed/quality tradeoff: although writing to finer levels enables more control over

the PSF appearance, writing to coarser levels is faster. The final step is to upsample the coarse levels through the pyramid, collapsing the pyramid into a final, blurred image. Care must be taken to use upsampling filters that are sufficiently wide, otherwise block artifacts can appear. See Figure 6.8 for an example of a pyramid and the resulting blurred output.

Since the number of levels that a pyramid has are finite, it is difficult to represent PSF sizes that lie between two pyramid levels. We considered two possible solutions to this. The first approach is to write to both of those two levels, each at an intensity proportional to where the continuous value truly lies. The second solution is to write to the finer level, using a slightly larger PSF to compensate for being at too fine a level. Both of these methods increase quality, but incur additional cost. In practice, the first method was found to produce superior results insofar as nterpolating between two smooth PSFs of different size happens to produce the appearance of an intermediate-sized PSF. Whereas, although writing a slightly larger PSF to a slightly finer level does indeed lead to a PSF of the correct intermediate size, this PSF will undergo a visible discontinuity if its size is increased or decreased such that it lands in a different level. Even though the size is correct, PSFs at different levels appear different, due to different rasterizations at different levels.

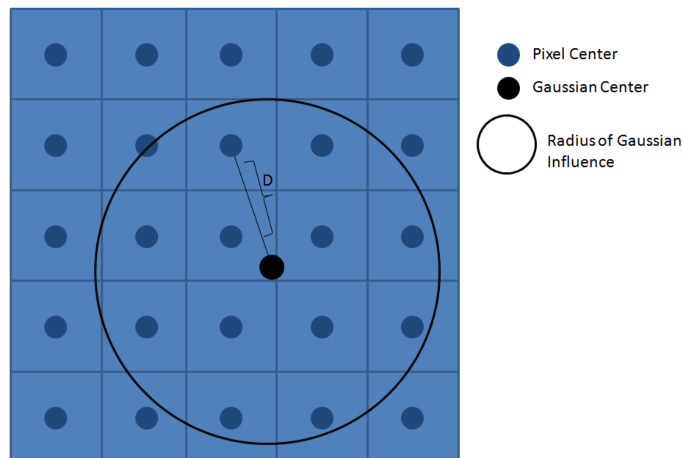
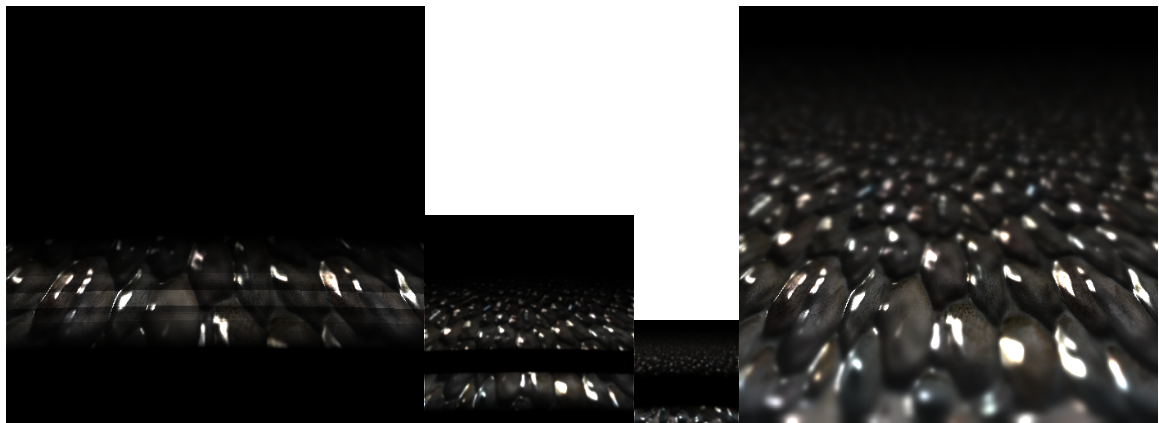


Figure 6.7: When computing a subpixel Gaussian, the continuous distance between Gaussian center and pixel center is taken. It is critical that the Gaussian center is not snapped to a pixel location.

It is worth delving into the details of how to properly spread PSFs to a given level, and how to upsample the pyramid. Although these operations could be implemented in various ways, great care must be taken to produce the smoothest possible images, free of grid artifacts.

During spreading, we want to place a PSF located at a position dictated by the location of a pixel in the input image. However, since the PSF is being spread to a coarse pyramid level,



(a) Finest pyramid level. (b) Middle pyramid level. (c) Coarsest pyramid level. (d) Output of upsampling this pyramid.

Figure 6.8: The pyramid method spreads PSFs to various pyramid levels, then upsamples the levels through the pyramid to yield a final, blurred image.

the PSF must be rendered with subpixel accuracy. Issues of how to properly anti-alias the PSF can be simplified if we restrict ourselves to Gaussian PSFs, with Gaussian anti-aliasing filters. A Gaussian convolved with another Gaussian is simply a larger Gaussian, so our filtered PSF is simply a Gaussian of slightly larger size. The analytical nature of Gaussians makes them straightforward to render with subpixel precision (see Figure 6.7). For each pixel within the support of the Gaussian, determine the distance from the floating-point- valued center to the integer-valued pixel center; This distance provides the distance within the Gaussian, with subpixel accuracy.

During upsampling, the pixels from the coarse layer should blend together in the finer layer such that no grid artifacts are introduced. This is done by calculating each pixel in the finer layer as a weighted average of pixels in the coarser level. The weighted average is performed using a subpixel Gaussian in a manner very similar to the spreading step.

Below is the code for spreading a PSF to level i of the pyramid.

```
//Initialize pyramid

//pyramid[i] is a 2^i x 2^i image

//Therefore the levels have resolutions as follows.
//The following entries fill the pyramid_widths table.
```

```
//level  resolution
//0      1x1
//1      2x2
//2      4x4
//3      8x8
//4      16x16
//5      32x32
//6      64x64
//7      128x128
//8      256x256
//9      512x512
//10     1024x1024


//X and Y are initially at the full resolution of the
//output image.
//Scale them so that we are within the coordinate system
//of the current pyramid level.
X = (X/width)*pyramid_widths[i];
Y = (Y/width)*pyramid_widths[i];


int radius = 3;


float_type U;
float_type V;


//Spread a 7x7 Gaussian
for(int u = -3; u <= 3; u++)
for(int v = -3; v <= 3; v++)
{


//Find the floating point pixel location of
//where we are spreading to
U = X + u;
V = Y + v;


//Round U and V to the integer pixel grid,
// add .5 to get the pixel center
// and subtract from X and Y, the
```

```
//floating point center of the Gaussian
//This yields du and dv are the
//floating point offset from the Gaussian
// center to the pixel center
float_type du = (int)U - (X-.5);
float_type dv = (int)V - (Y-.5);


//Compute the length of the offset
float_type dist = sqrt( du*du + dv*dv);

//The standard deviation was chosen by trial and error to be 1/3.
float_type stddev = 1/3.0;

//Evaluate the Gaussian
float_type G = exp(-(dist*dist)/(2*stddev*stddev));


//Divide through by the volume under the Gaussian
//for normalization purposes.
float_type weight = 1/(2.5*stddev*sqrt(2*3.14159));

//Cast the pixel location to an integer so we can write to the image
int iU, iV;
iU = U;
iV = V;

pixel_red(pyramid[i], iU,iV) += r*weight*G;
pixel_green(pyramid[i],iU,iV) += g*weight*G;
pixel_blue(pyramid[i],iU,iV) += b*weight*G;
pixel_fourth(pyramid[i],iU,iV) += 1.0*weight*G;


}
```

Below is the code for upsampling through the pyramid.

```
//Upsample

//Assuming that the pyramid level is of unit width,
// calculate the width of one pixel
float_type one_pixel = 1.0/pyramid_widths[i];
```

```
//Iterate over the pyramid levels that we will upsample
for(int i = 0; i <= 8; i++)
{

//Iterate over each of the pixels in
//the level that we are upsampling to
for(int x = 0; x < pyramid_widths[i+1]; x++)
for(int y = 0; y < pyramid_widths[i+1]; y++)
{
int X = x;
int Y = y;

int radius = 3;
int U;
int V;

    //Calculate the size of one pixel,
    //assuming the pyramid level is of unit width.
float_type one_pixel = 1.0/pyramid_widths[i];

    //Iterate over all the pixels
    //within the support of the filter kernel.
for(int u = -radius; u <= radius; u++)
for(int v = -radius; v <= radius; v++)
{

   //Scale the pixel location
   //to match the coarser level,
   //and shift to the location
   //within the support of the filter kernel.
U = X/2.0+u+.5;
V = Y/2.0+v+.5;


     //Compute the floating point offset based on where
     //we are sampling the filter kernel.
 float_type du, dv;
 du = (X-.5) - U*2;
 dv = (Y-.5) - V*2;

     //Evaluate the magnitude of that offset.
     float_type dist = sqrt((float_type) (du)*(du) + (dv)*(dv));

    //Evaluate the Gaussian filter kernel.
```

```
float_type stddev = radius*one_pixel;
stddev = radius/3.0;
float_type G = exp(-(dist*dist)/(2*stddev*stddev));
G /= (stddev*sqrt(2*3.14159));


//Write the weighted value to the next pyramid level.
    pixel_red(pyramid[i+1],X,Y) +=    pixel_red(pyramid[i],U,V)*G;
pixel_green(pyramid[i+1],X,Y) +=  pixel_green(pyramid[i],U,V)*G;
pixel_blue(pyramid[i+1],X,Y) +=   pixel_blue(pyramid[i],U,V)*G;
pixel_fourth(pyramid[i+1],X,Y) += pixel_fourth(pyramid[i],U,V)*G;


}
}
```

6.5.2 Performance

The cost of spreading a Gaussian depends on the resolution of that Gaussian. However, larger Gaussians can get written to coarser levels of the pyramid, so all Gaussians have approximately the same resolution, and hence the same cost.

We found that a 7x7 Gaussian is necessary to produce the smoothest results. The effect of intermediate sizes are achieved by writing to two different levels. This means that $7 \times 7 \times 2$, or 98 writes, are required per pixel. This means that the pyramid method is an expensive method. This is necessary if we wish our PSFs to be effectively perfect Gaussians.

The cost of upsampling the pyramid must also be taken into account. For each level of the pyramid except for the coarsest, upsampling from the next coarsest level must occur.

Since the entire pyramid can fit into a space somewhat less than twice as big as the finest level, we can bound the cost by calculating the cost of upsampling $2 \times N$ pixels. Each pixel that must be upsampled requires spreading 7x7 pixels, to achieve high quality. Therefore we can bound the cost of upsampling by $98 \times N$ writes.

Chapter 7

Tensor Filter

7.1 Algorithm

This next method is known as the tensor method, because it is derived from an analysis of the blur tensor. To build intuition, we will develop the algorithm for the case of applying a 2D matrix applied to a 1D image vector. The full version of this method can be viewed as arising from a similar derivation, but for applying a 4D tensor to a 2D image. Fortunately, it will not be necessary to work in 4D. The full algorithm will be clear once the simplified version has been elucidated.

The tensor method works by exploiting structure in the matrix. We need to find a way to simplify the matrix such that the matrix vector multiplication is more efficient. A simple type of matrix is one that is separable, meaning that it can be factored into the outer product of two vectors. Given the factorization, the matrix-vector multiplication can be applied simply by multiplying the input vector by the factors. Although blur matrices are not separable, we can still use this idea of separable matrices to our advantage.

The first idea was to break the matrix into blocks, and then approximate each block as being separable, but this turned out not to be a good approximation, thus the next idea was to use the singular value decomposition (SVD) to get a better approximation. The SVD can be used to find the best low-rank approximation to a matrix, for any given rank. A matrix of rank N can be efficiently multiplied (if N is sufficiently small), because it is the sum of N separable matrices. We used the SVD to find the best low-rank approximation for each of the blocks (see Figure 7.1). Although this SVD method does work, it does not lead to the same level of performance gains that our other methods achieve. Furthermore, the blur matrices for reasonable sized images are so large that taking the SVD would be prohibitively expensive. Finally, all these problems notwithstanding,

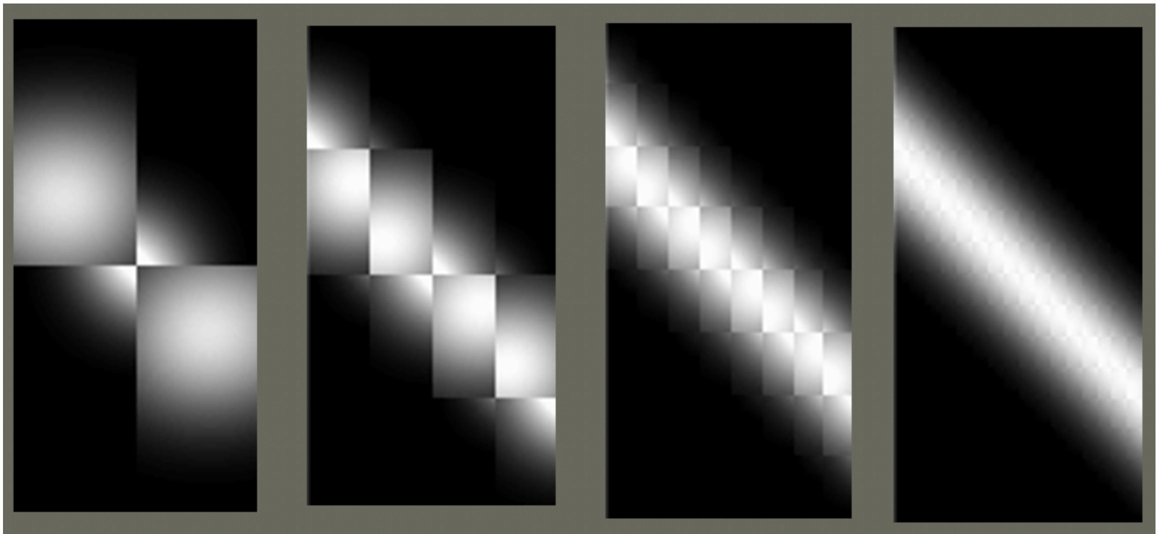
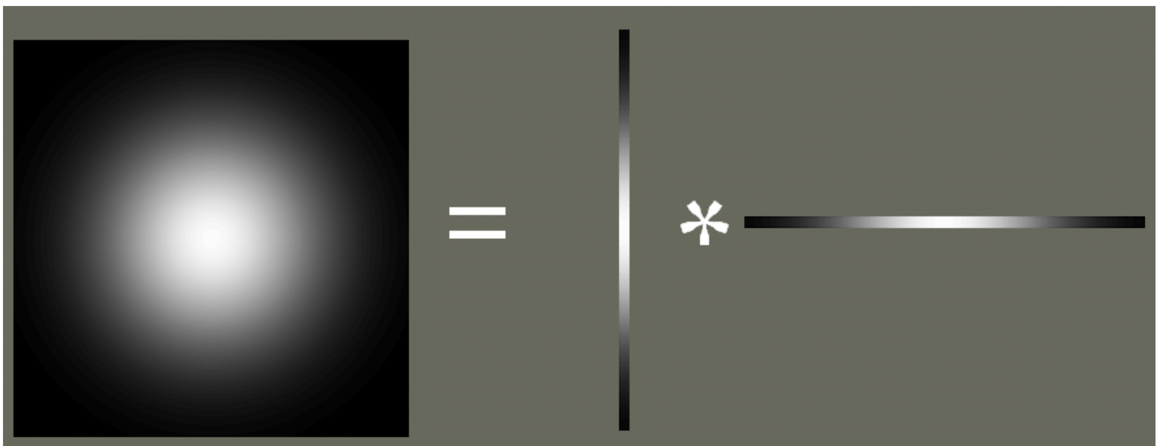


Figure 7.1: Using the SVD to find the best separable approximation of blocks within the matrix. From left right, the blocks are of increasingly small size.

the SVD would have to be taken ahead of time, offline, as a preprocess, making this method useless for dynamic scenes.



(a) A radially symmetric Gaussian can factored into two vectors. Multiplying by these two vectors is much more efficient than multiplying by the original matrix.

Figure 7.2: A radially symmetric Gaussian can be factored into the outer product of two one dimensional Gaussians.

A better idea is to directly exploit the smoothness of the matrix by downsampling it. A matrix can be understood as smooth if we view it as a grayscale image, and that image appears

smooth. The larger the magnitude of the blur we are applying, the smoother the matrix is; this enables coarser samplings without losing any detail. Given the coarse matrix, we then need a way to apply that matrix to the full-resolution vector. This is done by observing that we *could* reconstruct the original matrix by using reconstruction kernels. Reconstruction kernels restore the original matrix by spreading a Gaussian for each sample. However, we do not in actuality want to actually reconstruct the original matrix, but merely want to calculate the effect of its multiplication. Fortunately, our Gaussian reconstruction kernels are separable (see Figure 7.2), so we can apply them to the vector directly from their factorization as the outer product of two vectors.. We have effectively represented our matrix as overlapping separable matrices. The overlapping nature of our sub-matrices is a critical difference compared to the SVD method. The massive preprocessing of the SVD method is also avoided, since the subsampled matrix can be computed much more easily than the SVD.

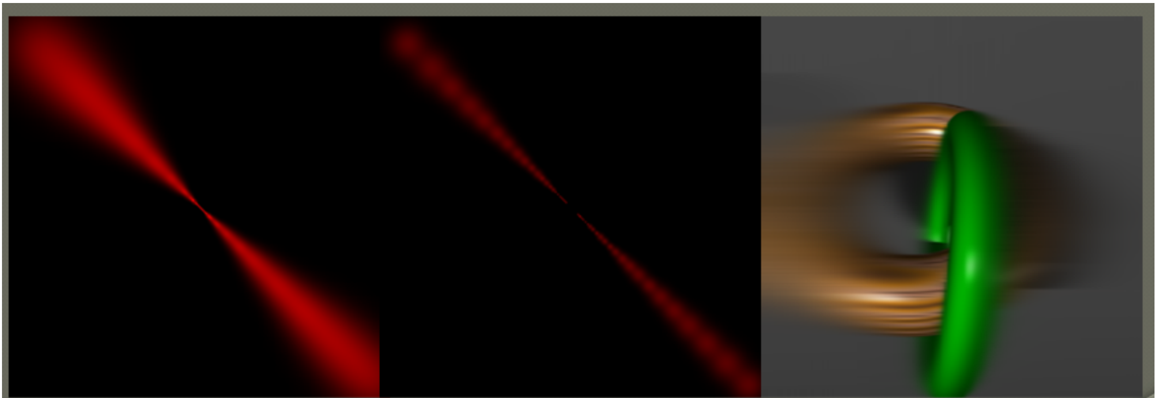


Figure 7.3: A blur matrix can be adequately reconstructed by Gaussians placed down the diagonal. For illustrative purposes, this example is for 1D blurring, i.e. we are only blurring horizontally. Left: the blur matrix constructed out of Gaussians down the diagonal. Center: Every other Gaussian removed, to make the remaining Gaussians more visible. Right: The result of blurring an image with the matrix on the left.

Fortunately it is unnecessary to sample the matrix on a full 2D grid. Rather, samples can be placed only down the diagonal (see Figure 7.3). This is because the blur matrix happens to have all of its energy (nonzero values) centered around the diagonal. The band of energy has varying size, depending on how much blur is required. To handle this varying size, the reconstruction kernels must also have varying size. Substantial time is saved by only sampling on compared to sampling on a full grid.

The application of a separable Gaussian matrix has a direct interpretation in terms of

spreading and gathering. First, we multiply by the row vector. Multiplication by the row vector is a gathering operation, computing a weighted average of the elements of the input vector. The filter kernel for this gathering operation is itself a Gaussian. Second, this scalar weighted average is multiplied by the column vector to determine the output. Since our separable matrices are overlapping, we sum them to get the blurred image, because our tensor is constructed as the sum of separable blocks. This means that application of the column vector involves summing Gaussians, a spreading operation. Therefore each separable matrix involves a gather, followed by a spread.

Given this interpretation, it is straightforward to extend the method from 1D images to 2D images. Simply select a number of sites (locations) in the image, and specify a Gaussian size for each site. The locations and size of the sites will need to be determined by consulting the blur map. For each site, perform a gather followed by a spread. It is clear that this works just as well in 2D as it did in 1D.

The remaining challenge is figuring out how many Gaussians to place and where to place them. If we place too many, performance will be slow. But if we place too few, there will be gaps in the blurred image. If we don't place them with just the right spacing, the blurred image will contain artifacts. To simplify first consider how the placement should work for the case where the blur amount is the same throughout the image. This simplification enables constant spacing and a constant size for all the sites. We can control the amount of blur varying the spacing between sites, close together for less blur, or farther apart for more blur. The Gaussians must be large enough to cause them to overlap without leaving gaps, but the Gaussians should not be too large, however, or else performance will degrade and excessive blur will occur.

We could consider storing the results of the gathering phase, and viewing each of these values as a pixels in an image. Although not actually necessary, it is useful for building intuition. The resulting image would be a lower-resolution version of the input image. The Gaussians in the row vectors would be the downsampling filters. Later, applying the column vectors recreates a full resolution image, effectively upsampling with a Gaussian reconstruction kernel. When viewed in this manner, the tensor filter algorithm is simply a new way of viewing the simplistic blur method of downsampling followed by upsampling (see Figure 7.4). The blur is caused because the low-resolution image is incapable of representing fine details.

To extend this method to the general case of an arbitrarily varying blur map, we need a way to place sites with a density that varies according to the blur map. This is similar to the importance sampling problem in rendering [45], which places more samples in important areas to gather light more effectively. Our problem is also similar to stippling from non-photorealistic

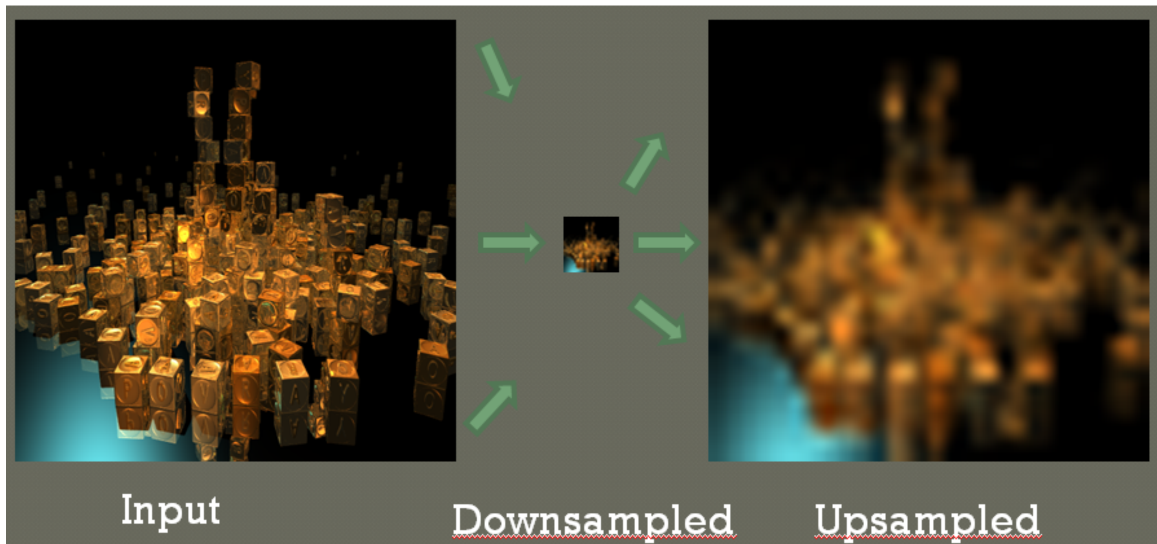


Figure 7.4: The tensor method for uniform blur is equivalent to downsampling followed by upsampling.

rendering [45], where points are placed to indicate variations in shading. We considered borrowing a couple of methods from the importance sampling and stippling literature, but we eventually settled on something far simpler.

For each pixel, consider the possibility of inserting a site. We will not place a site at every pixel, but we will rather skip a number of pixels based on the desired amount of blur. We calculate a variable called *skip_level*, which is simply a scaled version of the blur map value for that pixel. The scale factor was determined by trial and error. Next, the decision about whether or not to insert a site is made by modding the pixel location with *skip_level*. This very simple method has the effect of placing points with exactly the right density, in a spatially-varying way. The simplicity means that the method is fast and easy to implement.

Below is the code that implements the tensor filter.

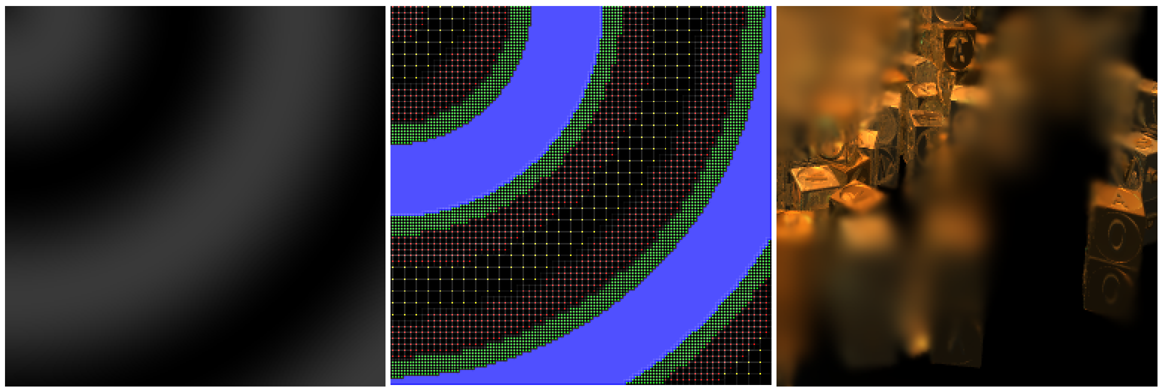
```
for(int x = 0; x < width; x++)
for(int y = 0; y < height; y++)
{


int skip_level = .8*pixel_red(blur_map,x,y,width*3,3)/4.0;

if ( (skip_level < 1) || (x % skip_level == 0 && y % skip_level == 0))
```

```
{
float_type radius = .8*pixel_red(blur_map,x,y,width*3,3);

    gather(x,y,radius,r,g,b,a);
spread(x,y,radius,r,g,b,a);
}
}
```

This very simple method of placing points has a limitation which is that there may be no sites at all on small but very blurred objects; in fact, such objects can be missed completely. To really make this method useful in the general case, a more sophisticated site placement method is needed, one that can ensure sure that no objects are missed. This method is useful not as a tool to be used in practice, but rather as a new and interesting way of thinking about the structure of the blur operator.

One solution to this problem is to use a more sophisticated method for placing the sites. A useful way to think of this problem is to consider that we are compressing the blur map. Since the sites are sparsely placed, there are relatively few degrees of freedom for controlling the amount of blur. A useful fact from study of human perception is that the higher the blur, the more difficult it is to perceive differences in the amount of blur. Therefore in regions with high blur blur, we can manage by with fewer sites. The quadtree method exploits this structure by representing blurred regions of the blur map with large nodes (see Figure 7.5). This is the standard method for compressing an image with a quadtree. We then place a site at the center of each leaf node. The quadtree building process ensures that no detail is lost, because sites are always placed where needed.

Each application of a site involves a gather followed by a spread. Consequently it is possible to accelerate the tensor filter by using fast gathering and spreading techniques. For gathering, any method that enables Gaussian filter kernels can be used, and for spreading, any method that enables Gaussian PSFs can be used. In practice, Heckbert's repeated integration technique is the fastest choice for gathering, and the polynomial method is the fastest option for spreading. Although accelerating the tensor filter in this manner does make it faster, it still leads to a slower method than simply using fast polynomial spreading as the entire filter. The reason to use the tensor filter is its simplicity; it is the easiest to implement of all the fast blur methods, and gives a high quality Gaussian blur. Adding in fast gather and spread methods defeats the simplicity by adding complexity. Therefore, we suggest using the tensor method in its original form, rather than in accelerated form.



(a) A blur map.

(b) Sites selected via a quadtree that compresses the blur map.

(c) Output of using the tensor method with sites located at the center of the quadtree leaf nodes.

Figure 7.5: Illustration of the tensor method with a quadtree used to lay out the sites.

7.2 Performance

It is easiest to analyze performance if we stick to the case of uniform blur. For each site, there is a gather and a spread. The uniform case is easy because the size of the gathers and spreads are constant. The number of sites is inversely proportional to the size of the blur, but the cost of each site is directly proportional to the size of the blur. Therefore the total cost of the tensor filter is constant, because the added cost of blurring larger Gaussians is offset by the fact that fewer Gaussians are required.

The case of continuously variable blur is difficult to analyze. Fortunately, the quadtree method does not distribute sites in a truly continuously variable manner. The quadtree has discrete levels, and each level implies a region of the blur map with uniform spacing. While the sites are placed on a uniform grid (within a level), they do not have uniform size. Their size can vary within a range. This range is bounded from below by the spacing of the grid; we don't allow Gaussians to be so small relative to their spacing that gaps appear. The range is bounded from above by the fact that if the Gaussian is too large, it will be placed in a coarser pyramid level. Since the size of the Gaussian is bounded, our analysis for constant-sized Gaussians is applicable. Therefore the cost of each level is bounded by a constant with respect to blur size. The total cost for blurring an image is therefore bounded by the sum of these constants, so this is overall a constant time algorithm.

Chapter 8

Comparisons

8.1 Discussion Of The Relationship Between The Various Methods.

This thesis describes a number of new image filters. The relative advantages and disadvantages of them may not be clear, so this section explains what properties each method has that makes it stand out.

When blur is small, there is no need for complex algorithms. PSFs up to 5x5 or 7x7 are easy and efficient when implemented directly. The small-blur situation is fortunate, because when direct spreading is used, any PSF can easily be used, with no restrictions whatsoever. However, performance drops very quickly as blur gets larger, so direct spreading is appropriate only for the smallest of blurs.

For cases where we want large blur to be efficient, and simplicity is paramount, the rectangle spreading method is best. The rectangle spreading method is very simple to implement, requiring only a few lines of code. The GPU implementation is relatively straightforward as well. The downside is that image quality is not ideal, because of the very simple rectangular-shaped PSF.

The perimeter method is unique in that it allows for circular PSFs, which allow for an accurate bokeh simulation of typical lenses. The perimeter method can get relatively slow for large blurs, however. Therefore the perimeter method occupies a high point in the quality / performance tradeoff.

The polynomial method spreads polynomial-valued PSFs in constant time with respect to size. Cost is proportional to the number of terms in the polynomial. Polynomials are quite appropriate for creating approximations to Gaussians. The polynomial method is probably the simplest and easiest to understand of the methods that can truly spread approximate Gaussians. The ability to use

arbitrary polynomials is potentially of great utility, due to its generality. Performance enjoys a convenient tradeoff, as higher quality can always be achieved by the use of higher order polynomials, at the cost of some performance.

The pyramid method has the unique advantage of being the only method that can spread true Gaussians. When appropriately large filters are used during each stage of the pyramid method, the resulting PSF is effectively a true Gaussian. The pyramid method has the disadantage of being somewhat slower than the other methods. The pyramid method is still constant-time per pixel, but it has a high constant.

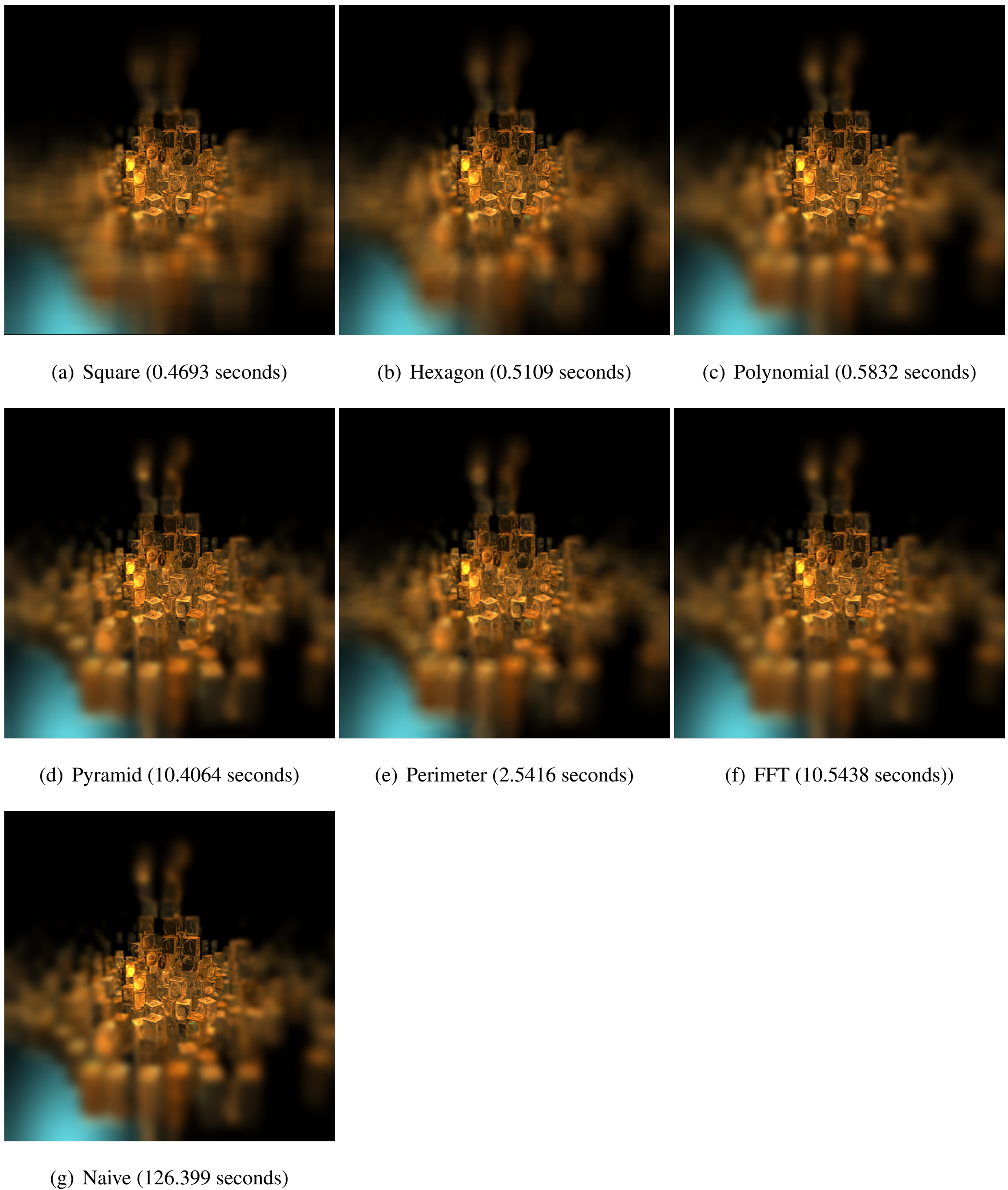
The tensor method is unique because it uses true Gaussians, but is much simpler and faster than the pyramid method. The downside is that the tensor method works best for symmetric matrices, and hence is not truly a spreading method, because spreading matrices are not symmetric. Being a true spreading method is not critical in all circumstances, especially for those situations where the blur map is constant or varies smoothly. The tensor method is less appropriate when the blur map is rapidly changing. The tensor method is unique in its mathematical structure, as it is conceptually very elegant compared to the other methods.

8.2 Visual Comparison Of Results

We have run our algorithms on several different images, both high dynamic range (HDR) and low dynamic range (LDR). For LDR comparisons, see Figures 8.1 and 8.2. For the HDR comparisons, see Figures 8.3 and 8.4. The difference in quality between the various algorithms is much more apparent for HDR images. This is because bright spots in an HDR image become highly visible images of the PSF. Therefore any approximations in the PSF will be visible. We conclude that more expensive, higher quality methods are needed for HDR images, but faster, approximate methods can be used for LDR images.

We have also included a few competing algorithms, to see how our methods compare. The competing methods include naive blurring, FFT with layers, and Piponi's hexagon method [65]. The polynomial method is clearly the best for Gaussian blur. Our perimeter method is the only algorithm we are aware of that can efficiently apply continuously-varying circle-shaped PSFs. Our rectangle spreading method (square spreading) is the fastest method available in our tests, which might make it applicable for situations where speed is more important than quality. Piponi's hexagon method is the only method that we know of that can apply polygonal shaped PSFs in constant time.

The perimeter spreading method can apply any PSF of arbitrary shape, so long as it has



(a) Square (0.4693 seconds)
(b) Hexagon (0.5109 seconds)
(c) Polynomial (0.5832 seconds)
(d) Pyramid (10.4064 seconds)
(e) Perimeter (2.5416 seconds)
(f) FFT (10.5438 seconds))
(g) Naive (126.399 seconds)

Figure 8.1: Various blur methods used on a low dynamic range image. Observe that each method produce similar result to the others.

a constant-intensity interior. The effects of using the perimeter method with different PSFs can be

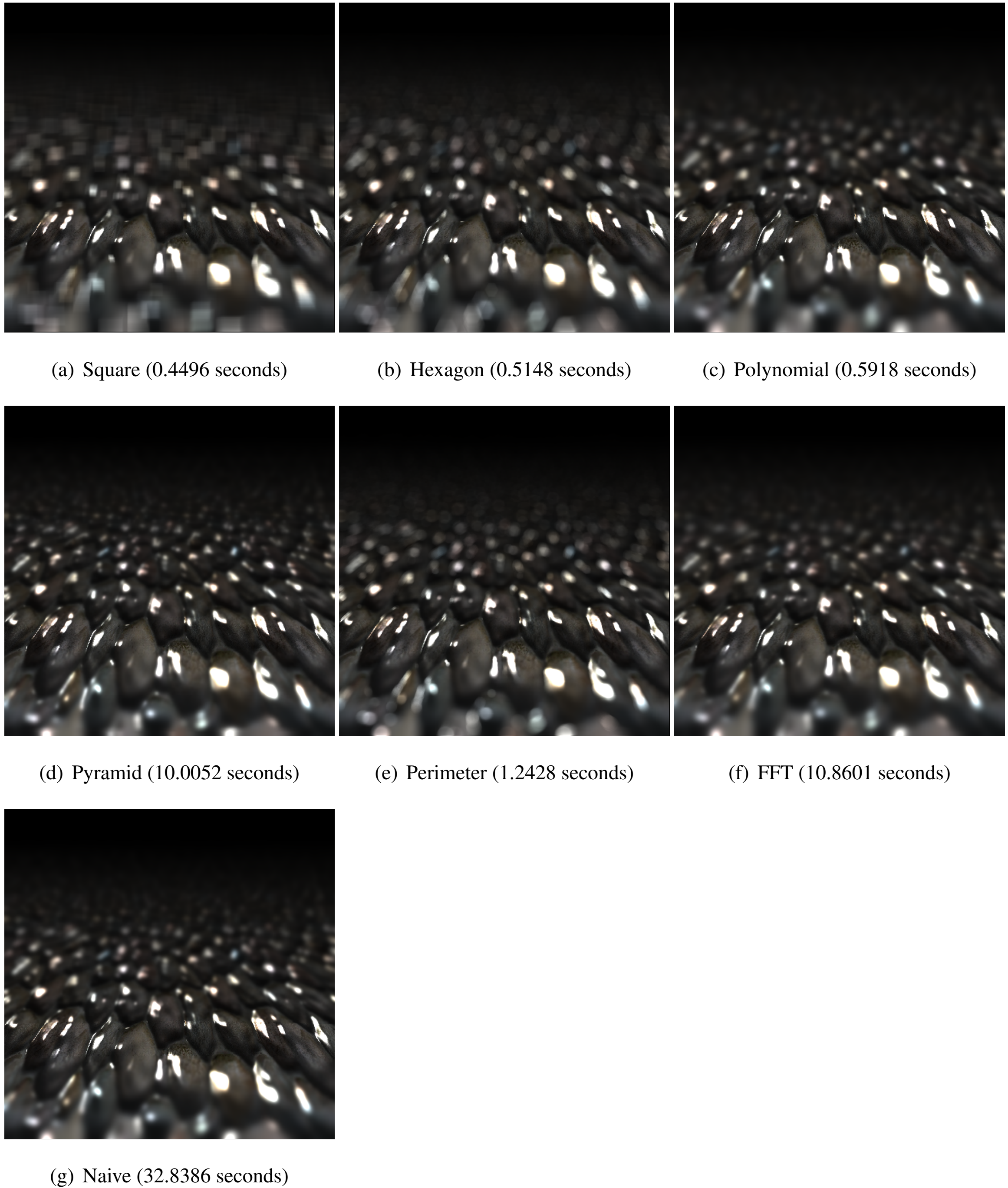
(a) Square (0.4383 seconds) (b) Hexagon (0.5448 seconds) (c) Polynomial (0.6022 seconds)

(d) Pyramid (11.7141 seconds) (e) Perimeter (2.2822 seconds) (f) FFT (11.2467 seconds)

(g) Naive (99.7249)

Figure 8.2: Various blur methods used on a low dynamic range image. Observe that each method produce similar result to the others.

seen in Figure 8.6. The circle and hexagon PSFs are useful because they match the effect of using



(a) Square (0.4496 seconds) (b) Hexagon (0.5148 seconds) (c) Polynomial (0.5918 seconds)

(d) Pyramid (10.0052 seconds) (e) Perimeter (1.2428 seconds) (f) FFT (10.8601 seconds)

(g) Naive (32.8386 seconds)

Figure 8.3: Various blur methods used on a high dynamic range image. Observe that the methods produce significantly different results.

real cameras with high quality lenses. Therefore the perimeter method is useful for photorealistic

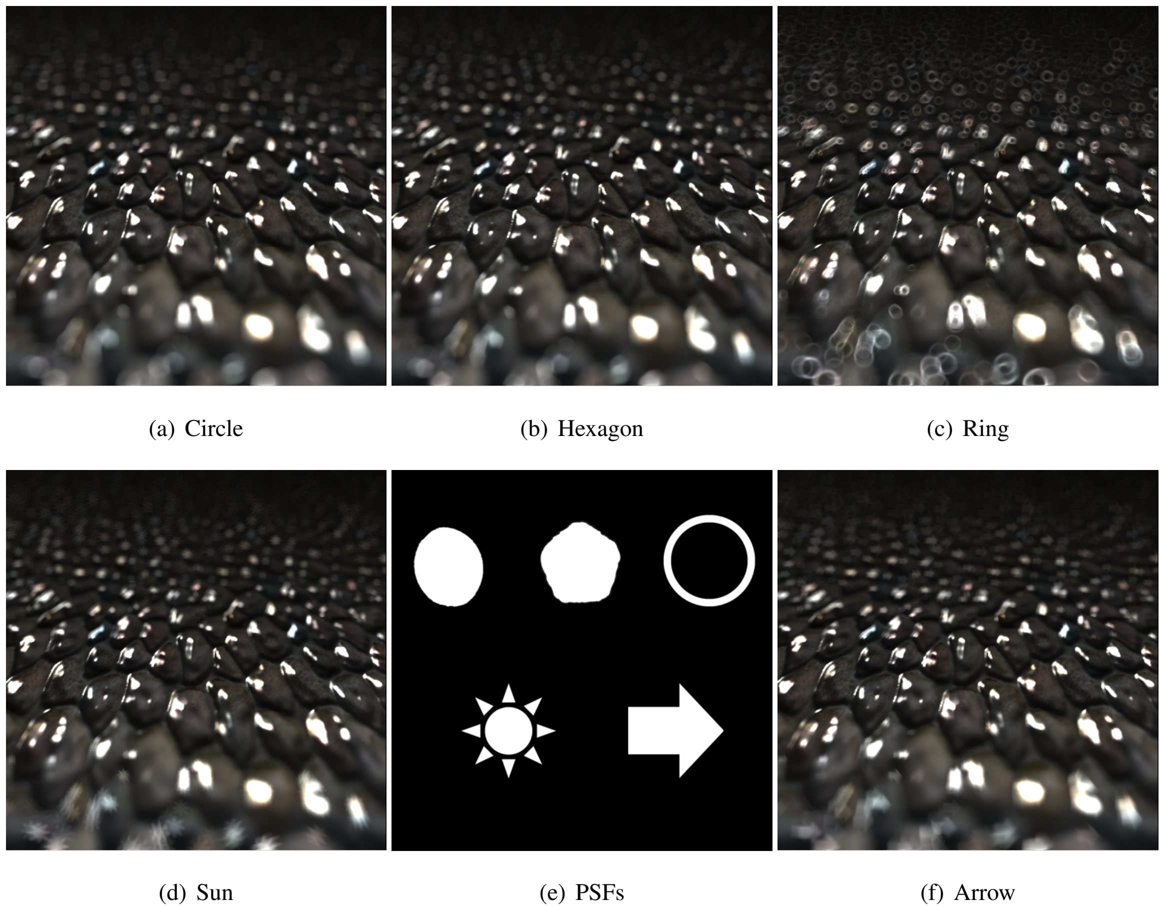
(a) Square (0.5008 seconds) (b) Hexagon (0.5155 seconds) (c) Polynomial (0.5749 seconds)

(d) Pyramid (6.5418 seconds) (e) Perimeter (0.6365 seconds) (f) FFT (11.0106 seconds)

(g) Naive (9.98269)

Figure 8.4: Various blur methods used on a high dynamic range image. Observe that the methods produce significantly different results.

blur. The ring-shaped PSF is interesting because it produces strikingly different results. A ring-



(a) Circle (b) Hexagon (c) Ring

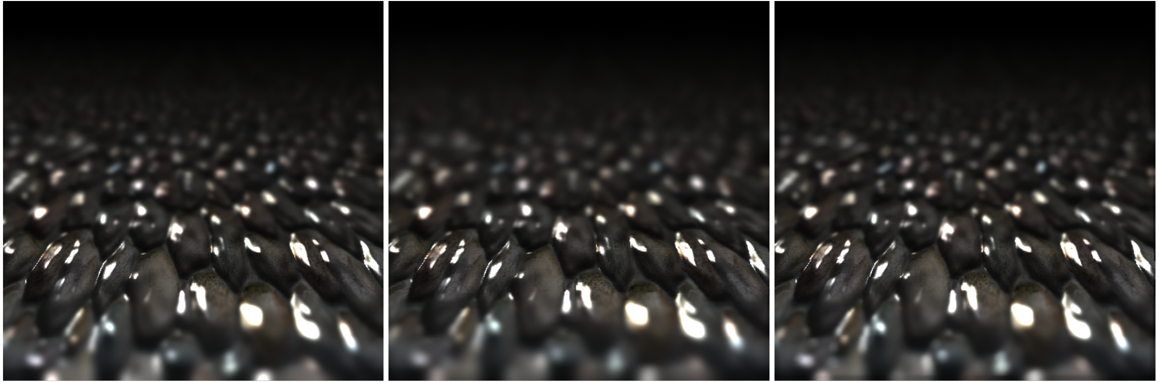
(d) Sun (e) PSFs (f) Arrow

Figure 8.5: The results of using the perimeter method with various arbitrary-outline constant-intensity PSFs.

shaped PSF is known in the photography community for producing notoriously bad bokeh; it is interesting to be able to simulate this efficiently. The sun and arrow PSFs are unusual special effects. PSFs like these are possible with real cameras if a cardboard cutout is placed in front of the lens.

We configured the tensor method three different ways. First, using naive gathering and spreading, second using the polynomial method for spreading and Heckbert's repeated integration method for gathering, and third, the pyramid method, both in a gathering and in a spreading formulation. Not surprisingly, using the polynomial method was fastest, because the polynomial method is faster than the both naive spreading and the pyramid method.

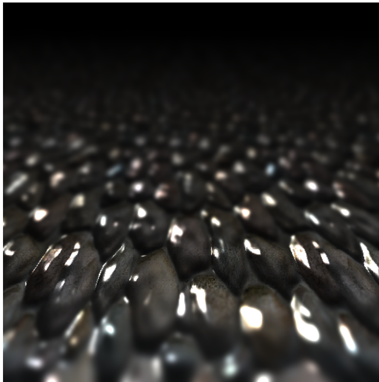
Finally, we compare gathering vs. spreading, both for the polynomial method and for the circle method (see Figure 8.7).



(a) Tensor blurring with naive gathering and spreading. (7.4686 seconds)

(b) Tensor blurring with polynomial gathering and spreading (0.8064 seconds)

(c) Tensor blurring with pyramid gathering and spreading. (6.9167 seconds)

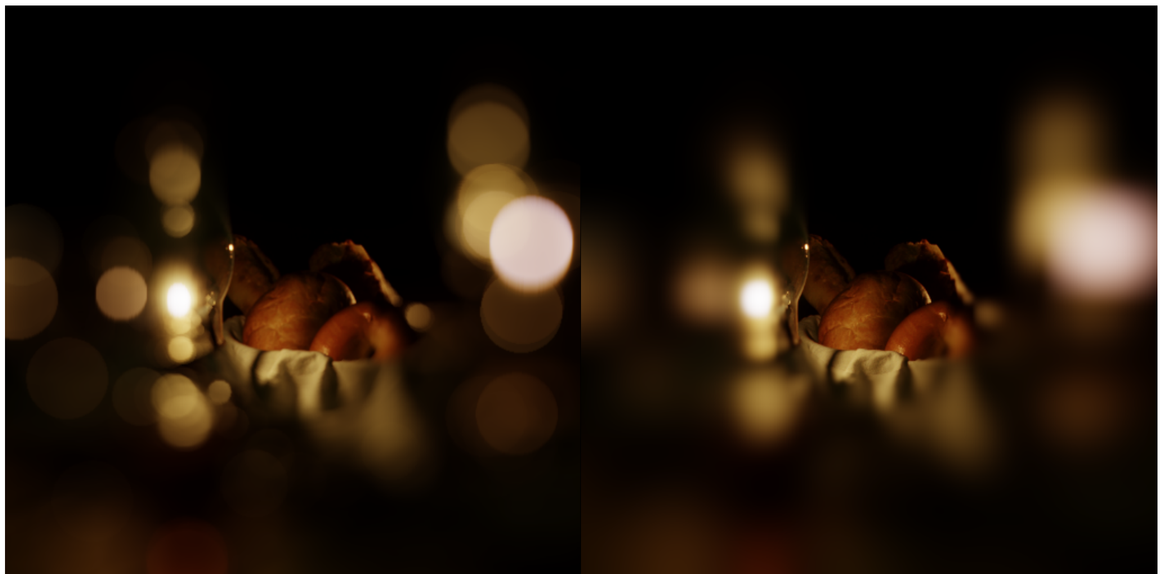


(d) For comparison, naive spreading without the tensor method. (37.8613 seconds)

Figure 8.6: Various configurations of the tensor method.

While we were developing the polynomial, pyramid, and tensor methods, it wasn't clear which one would be best. After implementing them, the polynomial method turned out to have significantly better performance than the other two, with comparable quality.

(a) Circle gathering (b) Polynomial gathering



(c) Circle spreading (d) Polynomial spreading

Figure 8.7: Gathering vs. Spreading

Chapter 9

The AMD HD58xx Ladybug Launch Demo: A GPU Implementation Of The Polynomial Method

9.1 Introduction

When a new video card is released, there are generally no video games yet available to take advantage of the new capabilities. In order to demonstrate the video card, several demonstration applications are generally released at the same time as the video card. For the ATI Radeon HD58xx series of video cards, one of the demos, known as the ladybug demo, was based on depth of field postprocessing using our polynomial spreading method. It turns out that creating an optimized GPU implementation is far more challenging than it may seem. First, a parallel form of the algorithm must be found. Second, the unique architecture of the GPU must be kept in mind. Great care must be taken to ensure that the the entire GPU is utilized; straightforward algorithms often lead to large portions of the GPU left unused.

9.2 GPU Implementation of the Polynomial Method

Of all our filters, the polynomial method is the most useful in practice. This is because it has the best blend of performance and quality. Therefore we chose the polynomial method for GPU implementation.

GPUs traditionally have followed the typical graphics pipeline. The basic operation is rendering a triangle, which involves a vertex processing stage, rasterization, and pixel shading. The vertex shader stage and the pixel shader stage have been programmable for several years, allowing flexibility in rendering and even the possibility of general purpose computation on the GPU. While the graphics pipeline is great for straightforward rendering tasks, it is not a good match for many general purpose computation tasks, including my filters. We did manage to implement the polynomial method in DirectX 10 (which strictly follows the graphics pipeline), but the implementation was complex and slow. DirectX 11 introduces a new mode of execution known as the compute shader. The compute shader simply executes code in parallel; there is no graphics pipeline involved at all.

In DirectX 10, delta spreading must be accomplished by rendering point primitives. The blending units are turned on and configured to perform additive blending, rather than the more common alpha blending. Rendering point primitives is inefficient on GPUs, because the rasterizers are optimized for rendering large triangles. The integration step is performed via a standard approach known as recursive doubling [36]. Recursive doubling is a multipass technique that requires $log(N)$ passes where N is the width of the screen. Besides the large number of passes, recursive doubling is suboptimal because it performs more computations than are strictly needed, as the price paid for making things parallel.

DirectX 11 introduces means to perform more efficient delta spreading as well as more efficient integration. Delta spreading can be implemented in a very efficient, straightforward way, using two new features of DirectX 11: scatter and atomic addition. Scattering means that a shader can write to arbitrary locations in memory. Atomic addition means that when multiple threads attempt to accumulate to the same position in memory, the operations will be automatically serialized, and no conflicts will occur.

Initially, we tried implementing delta spreading in DirectX 11 as follows. For each input pixel, a thread is launched. Within a thread, all the deltas for that pixels are sequentially accumulated at the appropriate location. It turned out that a slightly different implementation leads to better performance. Rather than dispatch the entire operation at once, it is better to divide it up by delta. For example, in second order polynomial spreading, 9 deltas are used per pixel. First, the top left delta for all pixels will be accumulated in parallel. Next, the top center delta for all pixels will be accumulated in parallel. This leads to more passes, but even so, performance is significantly improved. The reason is probably because splitting it up leads to a better serialization compared to whatever the atomic addition operation is automatically doing.

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	2	3	4	5	1	1	1	1	1	1	1	1	1	1	1
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Figure 9.1: Illustration of straightforward integration in DirectX 11. For clarity, this example is shown for a single row. In the full implementation, integration is performed for all rows in parallel, and then for all columns in parallel. Top: a simple input, chosen because the integral is trivial. Middle: Integration proceeds sequentially, from left to right. Bottom: result of integration.

The first integration method we tried was very straightforward (see Figure 9.1). For each row, launch one thread. Each thread then integrates the row sequentially, from left to right. A similar integration is later performed for each column. Note that because each thread ends up writing to all the elements in a row, scatter is required, so this simple method of integration was not available before DirectX 11. Unfortunately, one thread per row is not enough to keep the GPU occupied. Modern GPUs can execute in the neighborhood of 1,000 threads simultaneously. Additionally, threads are put to sleep when memory access is requested, to await response from the memory units. While one thread is asleep, there better be another thread on the queue waiting to execute, or else the GPU will be underutilized. In general, approximately 5000 threads must be in flight at any one time, to keep the GPU fully utilized.

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	2	1	1	1	2	1	1	1	2	1	1	1	2	1	1
1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4

Figure 9.2: We need more than one thread per row to fully utilize the GPU. Therefore we integrate simultaneously in four domains. Top: the four domains are highlighted. Middle: integration proceeds from left to right within each domain. Bottom: result of parallel integration. Observe that except for the first domain, we do not yet have the correct integral. A fixup step will be required.

Fortunately, there is a way to reformulate the integration step to use several threads per row. Each row is divided into several domains, and each domain is integrated in parallel. See Figure 9.2 for an illustration. Unfortunately, the correct result is not obtained, because the result

1	2	3	4	5	6	7	8	1	2	3	4	1	2	3	4
1	2	3	4	5	6	7	8	9	10	11	12	1	2	3	4
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Figure 9.3: The second, third, and fourth domains need to be fixed up. The fixup stage proceeds for each domain simply by adding the last value of the previous omain. The domains must be fixed up sequentially, but the fixup stage within each domain proceeds in parallel. Top: fixing up the second domain. Middle: fixing up the third domain. Bottom: fixing up the fourth domain.

of integrating the later domains depends on the result of the previous ones. A simple fixup step is required (see Figure 9.3, which adds the value at the end of the previous domain to the entirety of the next domain.

9.3 Components Of The Demo

Aside from an implementation of the repeated integration method, the key asset of this demo is the artwork. The artwork includes a garden scene with several flowers and flower buds. The garden serves as the backdrop for a highly detailed ladybug. Flowers, leaves, and a ladybug are the perfect scene for depth of field because these objects are all very small. To get the objects to take up a sufficiently large portion of the image, the camera must be placed very close to the objects. This scenario is the regime of macro photography. In macro photography, depth of field is extremely prominent, because the aperture is very large relative to the subject matter. Extreme depth of field means that there is a sharp contrast between the in-focus regions and the out-of-focus regions. These extremes make depth of field especially dramatic.

First, there is a rolling demo mode. The user simply sits back and watches a predefined animation. A series of scripted camera paths were carefully designed to show off the depth of field most vividly. Second, there is an interactive mode, which plays a little bit like a simplified video game. The user can fly the camera around, with the camera behaving like a rigid body, driven by an underlying physics simulation. The user can press keys on the keyboard to manually adjust the size of the aperture. The user can use the mouse wheel to manually move the focus depth forward and back. There is also an autofocus mode, accessible by pressing space bar. While spacebar is being held, the focus continuously tracks the object located at the center of the screen. A crosshair

is displayed in interactive mode, made up to look like a camera's viewfinder. The crosshair shows where on the screen autofocus is trying to focus on, as a visual guide to the user.

(a) Autofocus mode with focus on the background

(b) Screenshot of splitscreen mode.

(c) Another screenshot of splitscreen mode.

Chapter 10

Conclusion

10.1 Summary

This thesis has introduced several new methods for blurring images quickly and with high quality. Each of these methods achieve speed by exploiting a different kind of structure in the problem. Most of our methods exploit structure in the PSF. These methods keep the PSF in a compact form until the last possible minute, constructing a final image with a postprocessing step. The polynomial method expresses PSFs as coefficients of a polynomial. The pyramid method represents PSFs as low-resolution Gaussians. The perimeter spreading method describes only the outline of a PSF, implicitly filling it in with a constant value. The tensor method is different. It attempts to access all the structure possible, by dealing with the full 4D blur operator.

Any of these methods are candidates for a real time GPU implementation. However, in initial CPU implementations, the polynomial method proved to have excellent quality and had the best performance. Therefore we chose the polynomial method for GPU implementation, and managed to achieve over 30 frames per second at high resolution.

In the course of developing fast blur filters, we uncovered an interesting linear algebraic theory. Since the blurring operation is linear, the process can be expressed as a (very large) matrix vector multiply. If we apply this matrix row-by-row, with a filter kernel in each row, we get a gather filter. If we apply the vector column-by-column, with a PSF in each column, we get a spreading filter. Viewing gathering and spreading in this way explains why gathering produces incorrect results if we use it in situations where spreading is indicated; we are effectively using the transpose of the correct filter. The linear algebra perspective on blur also led to the development of a completely new type of image filter, based on overlapping separable blocks.

10.2 Future Work

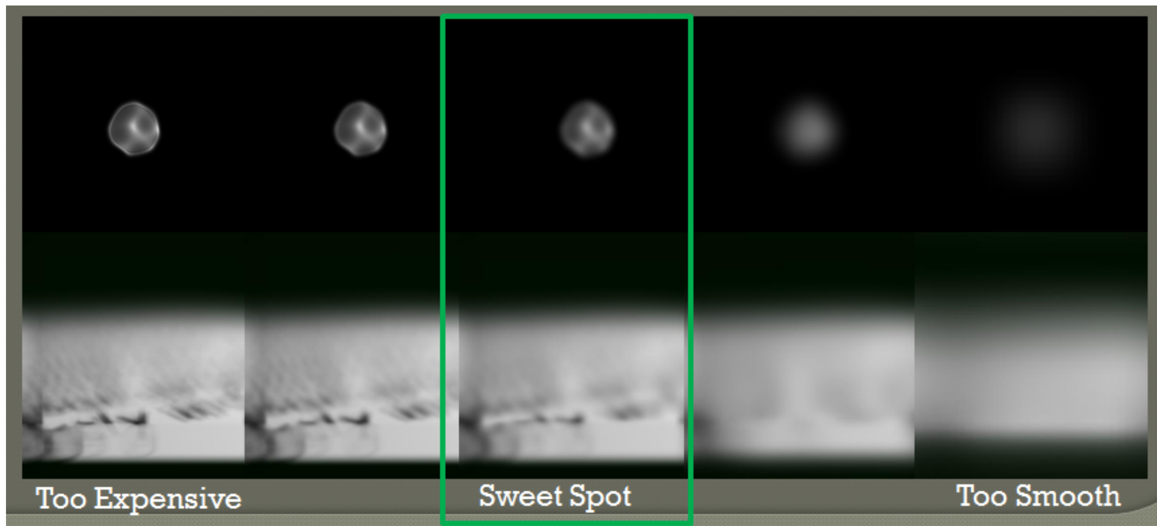


Figure 10.1: A vision realistic PSF can be simplified considerably before noticeable changes in the blurred output occur. This gives hope that a reasonably simple polynomial can do a good job at approximating vision realistic PSFs. My polynomial method thus may be useful for accelerated vision realistic rendering.

Max introduced a method for motion blur based on shear, blurring, and unshearing [57]. His method starts with the notion that there are methods for efficiently applying a 1D box filter, with uniform filter size. This type of filter can do an acceptable job of simulating motion blur for a trivial case of motion blur (linear, axis-aligned motion). However, if the image is first sheared, then blurred, then unsheared, then the motion blur can follow curved paths. Our fast spatially-varying filters could be of great use in an extended version of Max's method. The spatially-varying capabilities of my filters would enable different parts of the image to have different amounts of motion blur. The polynomial method would be of particular use here, because the smoothly curved polynomial impulse response can accurately simulate the effect of variable-speed motion blur, with various shutter behaviors.

Vision realistic rendering, which uses PSFs from actual human patients to simulate the way individual people see the world, is typically a slow process. My fast image filters have the potential to accelerate vision realistic rendering. The polynomial method, for example, could be used if vision realistic PSFs can be reasonably approximated by polynomials. An informal experiment (Figure 10.1) shows that vision realistic PSFs can be smoothed out considerably before noticeable degradation occurs in the blurred image. It may be possible to use the methods of this thesis to

blur images in real time with these smoothed vision realistic PSFs on a near-future generation of graphics hardware.

Fast convolution via FFT is a very appealing blur method. It is the only method to our knowledge that can efficiently apply arbitrary PSFs of arbitrary size. The great flaw of the FFT is that it requires the PSF to be the same size at all pixels throughout the image. One way of overcoming this limitation is to use layers. The image is binned by depth, each depth layer is blurred via FFT, and the layers are composited. One important issue is what depths to place the layers at. The natural way to do this is logarithmically with respect to blur size [6]; layers are placed closer together where blur is small, and further apart where blur is large. This is reasonable because the eye is better at distinguishing differences in blur when the amount of blur is small.

However, even with this logarithmic distribution, there is room for improvement. Ideally, the layers should adapt to the scene. For example, there is no need to place layers at depths where there are no pixels. In general, the more pixels lie at a given depth, the more likely it should be that a layer is placed at that depth. The problem of allocating layers is in fact very similar to the problem of quantizing a grayscale image to fewer levels of gray. If we consider the blur map to be a grayscale image, we want to produce a quantized image with as few gray levels as possible, while matching the original blur map as best we can. One promising quantization method is based on k-means clustering. Each pixel is considered as a point in a one dimensional space, where position in that space is determined by the blur value of that pixel. The points are then clustered using k-means clustering. Each cluster then becomes a layer.

While layers can allow for a variety of blur levels, the blur levels are still discrete. It would be very useful if FFT convolution could be used with continuously varying blur levels. Continuous blur levels may be possible with a warp, blur, unwarp approach. The warp would cause some parts of the image to shrink, and other parts of the image to expand. A uniform blur is applied via FFT to the warped image. Finally, the inverse of the warp is applied. The parts of the image that were shrunk by the warp will appear more blurred, and the parts of the image that were expanded will appear less blurred. Since the warp can be continuous, we can have a continuously varying level of blur. The PSF can be any size and of arbitrary shape, because an FFT convolution was used.

One challenge is in designing a warp that matches a desired blur map. Fortunately, results from the importance sampling literature can be used. In importance sampling, point samples are placed densely in important regions, and sparsely in less important regions. To achieve the desired point distribution, a uniform grid is warped according to the importance function. Our warp/blur/unwarp method can be used with this same kind of warping function, where a blur map

is used in place of the importance function.

The warping method has difficulty with regions that are completely unblurred. The requisite warp would need to expand the unblurred region to infinite size. Since this is impossible, unblurred regions have to be handled separately, i.e. they are removed before the warping method begins, and then reinserted at the end.

Another area of future work is to create additional applications, beyond depth of field and generic blurring. For example, our filters could be used to blur hard shadows, as an approximation to soft shadows. One challenge that this shadow method would face is that it would have to blur the shadows without blurring the underlying surface texture. We could handle this by rendering shadows in a separate pass, into a separate image. That way the shadow can be blurred and then later applied to the textured scene. This might make sense in a deferred rendering scenario. The polynomial method could also 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. The spatially-variant feature of the method would allow scale-adaptive edge detection in a single pass. Once we have edge detection in place, it might be possible to use knowledge of edges to improve magnification quality in image warping. Our filter kernels could simply adapt their shape to not blur across edges. Volume rendering via splatting is an area where fast filter spreading could be useful. Volume rendering is quite computationally expensive, so acceleration via methods like ours are likely to help quite a bit. Fast filter spreading could be used as a fast, approximate method of evaluating radial basis functions. RBFs are useful in 2D for applications such as scattered data interpolation and image warping, but the evaluation step can be expensive when there are many centers. A potential pitfall is that our filter kernels are piecewise-tensor-product polynomial, unlike radially-symmetric RBFs. However, we can approximate radially-symmetric kernels with arbitrary accuracy by subdividing our polynomials.

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, columns, and blocks, as a way of finding new algorithms.

Bibliography

[1] A. Adams, N. Gelfand, J. Dolson, and M. Levoy. Gaussian kd-trees for fast high-dimensional filtering. *ACM Trans. Graph.*, 28(3):1–12, 2009.

[2] E. Adelson, C. Anderson, J. Bergen, P. Burt, and J. Ogden. Pyramid methods in image processing. *RCA Engineer*, 29(6):33–41, November 1984.

[3] T. Akenine-Möller, J. Munkberg, and J. Hasselgren. Stochastic rasterization using time-continuous triangles. In *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 7–16, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association. ISBN 978-1-59593-625-7.

[4] M. Ashikhmin and A. Ghosh. Simple blurry reflections with environment maps. *Journal of Graphics Tools*, 7(4):3–8, 2002.

[5] D. A. Atchison and G. Smith. *Optics of the Human Eye*. Butterworth-Heinemann Ltd., Woburn, Mass., 2000.

[6] 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, 2004. ACM. ISBN 1-58113-914-4.

[7] B. A. Barsky, D. R. Horn, S. A. Klein, J. A. Pang, and M. Yuf. 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, 2003.

[8] B. A. Barsky, D. R. Horn, S. A. Klein, J. A. Pang, and M. Yuf. 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, 2003.

[9] 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. In *Graphical Models 67(6)*, pages 584–599, 2005.

[10] 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, 2003.

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

[12] M. Born and E. Wolf. *Principles of Optics*. Cambridge University Press, Cambridge, 7th edition, 1980.

[13] D. Bradley and G. Roth. Adaptive thresholding using the integral image. *journal of graphics, gpu, and game tools*, 12(2):13–21, 2007.

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

[15] J.-X. Chai, X. Tong, and S.-C. Chan. Plenoptic sampling. In K. Akeley, editor, *ACM SIGGRAPH 2000 Conference Proceedings*, pages 307–318, New Orleans, July 2000.

[16] Y. C. Chen. Lens effect on synthetic image generation based on light particle theory. *The Visual Computer*, 3(3):125–136, October 1987.

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

[18] R. L. Cook. Stochastic sampling in computer graphics. *ACM Transactions on Graphics*, 5(1): 51–72, January 1986.

[19] 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. ISBN 0-89791-138-5.

[20] J. Demers. *GPU Gems*, pages 375–390. Addison Wesley, 2004.

[21] M. A. Z. Dippe and E. H. Wold. Antialiasing through stochastic sampling. In B. A. Barsky, editor, *SIGGRAPH 1985 Conference Proceedings*, pages 69–78, San Francisco, CA, USA, July 1985.

[22] P. Dubois and G. Rodrigue. An analysis of the recursive doubling algorithm. In *High Speed Computer and Algorithm Organization*, pages 299–305. 1977.

[23] K. Fatahalian, E. Luong, S. Boulos, K. Akeley, W. R. Mark, and P. Hanrahan. Data-parallel rasterization of micropolygons with defocus and motion blur. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pages 59–68, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-603-8.

[24] 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.

[25] R. Fernando. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education, 2004. ISBN 0321228324.

[26] 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. ISBN 0-89791-275-6.

[27] 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.

[28] S. J. Gortler, R. Grzeszczuk, R. Szeliski, and M. F. Cohen. The lumigraph. In H. Rushmeier, editor, *ACM SIGGRAPH 1996 Conference Proceedings*, pages 43–54, New Orleans, August 1996.

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

[30] S. Greene. Summed area tables using graphics hardware. Game Developers Conference, 2003.

[31] 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. ISBN 0-201-50933-4.

[32] E. Hammon. Practical post-process depth of field. In *GPU Gems 3*, pages 583–606. Addison-Wesley, 2007.

[33] 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. ISBN 0-89791-196-2.

[34] P. S. Heckbert. Fundamentals of texture mapping and image warping. In *Technical Report No. Report no. UCB/CSD 89/516*. University of California, Berkeley, 1989.

[35] W. Heidrich, P. Slusallek, and H.-P. Seidel. An image-based model for realistic lens systems in interactive computer graphics. In W. A. Davis, M. Mantei, and R. V. Klassen, editors, *Proceedings of Graphics Interface 1997*, pages 68–75. Canadian Human Computer Communication Society, May 1997.

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

[37] J. Hensley, T. Scheuermann, G. Coombe, M. Singh, and A. Lastra. Fast summed-area table generation and its applications. In *Computer Graphics Forum*, volume 24, pages 547–555, 2005.

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

[39] A. Isaksen. Dynamically reparameterized light fields. Master's thesis, Department of Electrical Engineering and Computer science, Massachusetts Institute of Technology, Cambridge, Mass, November 2000.

[40] A. Isaksen, L. McMillan, and S. J. Gortler. Dynamically reparameterized light fields. In K. Akeley, editor, *Proceedings of ACM SIGGRAPH 2000*, pages 297–306, July 2000.

[41] F. A. Jenkins and H. E. White. *Fundamentals of Optics*. McGraw-Hill Inc., New York, 1976.

[42] J. T. Kajiya. The rendering equation. In *ACM SIGGRAPH 1986 Conference Proceedings*, pages 143–150, Dallas, 1986.

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

[44] 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. ISBN 0-89791-701-4.

[45] J. Kopf, D. Cohen-Or, O. Deussen, and D. Lischinski. Recursive wang tiles for real-time blue noise. *ACM Trans. Graph.*, 25(3):509–518, 2006.

[46] T. J. Kosloff and B. A. Barsky. An algorithm for rendering generalized depth of field effects based on simulated heat diffusion. *Lecture Notes in Computer Science*, 4707:1124–1140, 2007.

[47] T. J. Kosloff, J. Hensley, and B. A. Barsky. Fast filter spreading and its applications. In *Technical Report No. UCB/EECS-2009-54*. University of California, Berkeley, http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-54.pdf, 2009.

[48] T. J. Kosloff, M. W. Tao, and B. A. Barsky. Depth of field postprocessing for layered scenes using constant-time rectangle spreading. In *GI '09: Proceedings of Graphics Interface 2009*, pages 39–46, Toronto, Ont., Canada, Canada, 2009. Canadian Information Processing Society. ISBN 978-1-56881-470-4.

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

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

[51] D. Laur and P. Hanrahan. Hierarchical splatting: a progressive refinement algorithm for volume rendering. In *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, pages 285–288, New York, NY, USA, 1991. ACM. ISBN 0-89791-436-8.

[52] M. E. Lee, R. E. Redner, and S. P. Uselton. Statistically optimized sampling for distributed ray tracing. In *ACM SIGGRAPH 1985 Conference Proceedings*, pages 61–67, San Francisco, July 1985.

[53] S. Lee, G. J. Kim, and S. Choi. Real-time depth-of-field rendering using splatting on per-pixel layers. *Computer Graphics Forum*, 7(27):1955–1962, 2008.

[54] S. Lee, G. J. Kim, and S. Choi. Real-time depth-of-field rendering using anisotropically filtered mipmap interpolation. *IEEE Transactions on Visualization and Computer Graphics*, 3(15): 453–464, 2009.

[55] M. Levoy and P. Hanrahan. Light field rendering. In H. Rushmeier, editor, *ACM SIGGRAPH 1996 Conference Proceedings*, pages 31 – 42, New Orleans, August 1996.

[56] Z. Lin and H.-Y. Shum. On the numbers of samples needed in light field rendering with constant-depth assumption. In *Computer Vision and Pattern Recognition 2000 Conference Proceedings*, pages 588–579, Hilton Head Island, South Carolina, June 2000.

[57] N. L. Max and D. M. Lerner. A two-and-a-half-d motion-blur algorithm. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 85–93, New York, NY, USA, 1985. ACM. ISBN 0-89791-166-0.

[58] L. McMillan and G. Bishop. Plenoptic modeling: An image-based rendering system. In *ACM SIGGRAPH 1995 Conference Proceedings*, pages 39–46, August 1995.

[59] A. Mohan, D. Lanman, S. Hiura, and R. Raskar. Image destabilization: Programmable defocus using lens and sensor motion. In *IEEE International Conference on Computational Photography (ICCP)*, 2009.

[60] 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.

[61] R. Ng. Fourier slice photography. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 735–744, New York, NY, USA, 2005. ACM.

[62] K. Perlin. State of the art in image synthesis. SIGGRAPH Course Notes, 1985.

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

[64] D. Piponi. *Private Communication*.

[65] D. Piponi. Two tricks for the price of one: Linear filters and their transposes. *Journal of Graphics, GPU, and Game Tools*, 14(1):63–72, Aug 2009.

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

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

[68] A. Robison and P. Shirley. Image space gathering. In *High Performance Graphics*, 2009.

[69] P. Rokita. Fast generation of depth-of-field effects in computer graphics. *Computer and Graphics*, pages 593–595, September 1993.

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

[71] T. Scheuermann and J. Hensley. Efficient histogram generation using scattering on gpus. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 33–37, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-628-8.

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

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

[74] 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.

[75] C. Soler, K. Subr, F. Durand, N. Holzschuch, and F. Sillion. Fourier depth of field. *ACM Trans. Graph.*, 28(2):1–12, 2009.

[76] I. Stephenson. Improving motion blur: Shutter efficiency and temporal sampling. *journal of graphics, gpu, and game tools*, 12(1):9–15, 2007.

[77] J. Strain. The fast gauss transform with variable scales. *SIAM J. Sci. Stat. Comput.*, 12: 1131–1139, 2009.

[78] S. Tan, J. L. Dale, and A. Johnston. Performance of three recursive algorithms for fast space-variant gaussian filtering. *Real Time Imaging*, 9:215–228, 2003.

[79] A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice, 2nd Edition*. Addison-Wesley Publishing Co., Reading, Mass., 1990.

[80] L. Williams. Pyramidal parametrics. In *SIGGRAPH '83: Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, pages 1–11, New York, NY, USA, 1983. ACM. ISBN 0-89791-109-1.

[81] C. Wyman and C. Dachsbacher. Reducing noise in image-space caustics with variable-sized splatting. *journal of graphics, gpu, and game tools*, 13(1):1–17, 2008.

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

[83] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. Ewa volume splatting. In *VIS '01: Proceedings of the conference on Visualization '01*, pages 29–36, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7803-7200-X.