

# SIP: A Smart Digital Image Processing Library

by

Mengyao Zhou

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

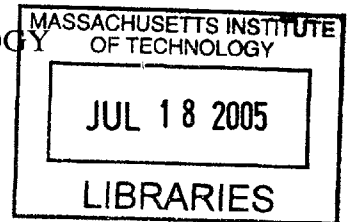
Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2005

© 2005 Mengyao Zhou. All rights reserved.



The author hereby grants to M.I.T. permission to reproduce and distribute publicly paper and electronic copies of this thesis and to grant others the right to do so.

Author \_\_\_\_\_  
Department of ~~Electrical~~ Engineering and Computer Science  
May 1, 2005

Certified by \_\_\_\_\_  
Lizhong Zheng  
Assistant Professor of Electrical Engineering  
M.I.T. Thesis Supervisor

Certified by \_\_\_\_\_  
Suhail Jalil  
Senior Staff Engineer/Manager  
~~VI-A Company~~ Thesis Supervisor

Accepted by \_\_\_\_\_  
Arthur C. Smith  
Chairman, Department Committee on Graduate Students

**BARKER**



SIP: A Smart Digital Image Processing Library

by

Mengyao Zhou

Submitted to the

Department of Electrical Engineering and Computer Science

May 1, 2005

In Partial Fulfillment of the Requirements for the Degree of  
Master of Engineering in Electrical Engineering and Computer Science

## ABSTRACT

The Smart Image Processing (SIP) library was developed to provide automated real-time digital image processing functions on camera phones with integer microprocessors. Many of the functions are not available on commercial camera phones and some are not found even in desktop image processing software. Five patents are pending for key functions in this library. These functions create realistic water reflections in varying weather conditions, perform localized magnification and pinching, transform photographs to perspective view, provide fast, high-quality spin radial blur, and provide a fast integer implementation for arbitrary rotation. Details on all pending patents are given in five chapters of this thesis. Other operations performed by the library include adding fog and shadow, creating a neon image, and creating a translucent corner fold. All library functions have been successfully implemented on an integer microprocessor for real-time performance in an existing camera phone system. The library also provides solutions to a number of long-standing problems in image processing, including direct application of transforms in subsampled YCbCr and YCrCb images.

M.I.T. Thesis Supervisor: Lizhong Zheng

Title: Assistant Professor of Electrical Engineering

VI-A Company Thesis Supervisor: Suhail Jalil

Title: Senior Staff Engineer/Manager



# Contents

<b>List of Figures</b> .....	<b>9</b>
<b>List of Tables</b> .....	<b>13</b>
<b>1 Introduction</b> .....	<b>15</b>
<b>2 Background and Purpose</b> .....	<b>17</b>
2.1 Existing System .....	17
2.2 Color Formats.....	17
2.3 Complications from Subsampled YCbCr and YCrCb .....	20
2.3.1 Simple but Limited Solutions .....	20
2.3.2 An Advanced Systematic Approach .....	20
2.4 Hardware Constraints.....	21
<b>3 Functions Developed</b> .....	<b>23</b>
<b>4 Magnification and Pinching (Patent Pending)</b> .....	<b>27</b>
4.1 Introduction .....	27
4.2 A New Approach.....	29
4.3 Case Studies .....	34
4.3.1 Magnify.....	34
4.3.1.1 Magnification Variations in a .....	34
4.3.1.2 Magnification Variations in k .....	35
4.3.2 Pinch .....	36
4.3.1.1 Pinching Variations in a.....	36
4.3.1.2 Pinching Variations in k.....	37
4.3.3 Changing Facial Expressions and Features.....	38
4.4 Comparison with Photoshop .....	39
4.4.1 Magnify.....	40
4.4.2 Pinch .....	41
4.5 Integer Implementation .....	43
4.6 Avoiding Integer Overflow .....	44
4.7 Implementation for Subsampled YCbCr and YCrCb.....	45
<b>5 Perspective Transformation (Patent Pending)</b> .....	<b>49</b>
5.1 Introduction .....	49
5.2 Perspective Projections.....	50
5.3 Two-Dimensional Perspective .....	54
5.4 A New Algorithm.....	56

5.5 Integer Implementation .....	60
5.6 Avoiding Integer Overflow .....	61
5.7 Implementation for Subsampled YCbCr and YCrCb.....	62
5.8 Results .....	64
<b>6 Neon Effect .....</b>	<b>67</b>
6.1 Introduction .....	67
6.2 Algorithm .....	68
6.3 Results .....	70
6.4 Comparison with Photoshop .....	71
6.5 Implementation for YCbCr and YCrCb .....	71
<b>7 Spin Radial Blur (Patent Pending) .....</b>	<b>73</b>
7.1 Introduction .....	73
7.2 A New Algorithm.....	76
7.3 Integer Implementation .....	83
7.4 Avoiding Integer Overflow .....	85
7.5 Results .....	87
7.6 Comparison with Photoshop.....	89
7.7 Implementation for Subsampled YCbCr and YCrCb .....	91
<b>8 Motion Blur .....</b>	<b>95</b>
8.1 Introduction .....	95
8.2 Algorithm.....	96
8.3 Integer Implementation.....	99
8.4 Results.....	100
8.5 Comparison with Photoshop.....	100
8.6 Implementation for Subsampled YCbCr and YCrCb .....	102
<b>9 Multi-directional Blur: Gaussian and Uniform Blur .....</b>	<b>103</b>
9.1 Introduction .....	103
9.2 Algorithm .....	105
9.3 Avoiding Integer Overflow .....	107
9.4 Results .....	108
9.5 Comparison with Photoshop.....	110
9.6 Implementation for Subsampled YCbCr and YCrCb .....	110
<b>10 Water Reflection (Patent Pending).....</b>	<b>113</b>
10.1 Introduction .....	113
10.2 A New Algorithm.....	116
10.2.1 Setting a Boundary and Determining the Axis of Reflection .....	116
10.2.2 Reflecting the Image.....	120
10.2.3 Adding Ripples .....	121
10.2.4 Blending with Water.....	124

10.3	Case Studies .....	125
10.4	Integer Implementation .....	130
10.5	Results .....	131
10.6	Implementation for Subsampled YCbCr and YCrCb .....	135
<b>11</b>	<b>Alpha Blending and Related Applications .....</b>	<b>139</b>
11.1	Introduction .....	139
11.2	Alpha Blending.....	140
11.3	Fog .....	141
11.4	Shadow .....	141
11.5	Fade In, Fade Out.....	143
11.6	Integer Implementation.....	144
11.7	Comparison with Photoshop.....	144
11.8	Implementation for Subsampled YCbCr and YCrCb .....	144
<b>12</b>	<b>Arbitrary Rotation (Patent Pending) .....</b>	<b>145</b>
12.1	Introduction .....	145
12.2	Existing Equations and Algorithms.....	146
12.3	A New Integer Implementation .....	148
12.4	Avoiding Integer Overflow .....	150
12.5	Implementation for Subsampled YCbCr and YCrCb.....	152
<b>13</b>	<b>Other Affine Transforms .....</b>	<b>155</b>
13.1	Introduction: Geometric Spatial Transformations.....	155
13.2	Affine Transforms .....	156
13.3	Shear.....	157
13.4	Scaling: Overlap and Resize.....	159
13.5	Translation: Film Strip.....	160
13.6	Composites: Horizontal, Vertical, and Center .....	161
13.7	Integer Implementation.....	163
13.8	Comparison with Photoshop.....	163
13.9	Implementation for Subsampled YCbCr and YCrCb .....	164
<b>14</b>	<b>Corner Fold .....</b>	<b>167</b>
14.1	Introduction .....	167
14.2	Algorithm .....	168
14.3	A Fast Integer Implementation .....	168
14.4	Results.....	169
14.5	Comparison with Photoshop.....	172
14.6	Implementation for Subsampled YCbCr and YCrCb .....	172
<b>15</b>	<b>Color Change .....</b>	<b>175</b>
15.1	Introduction .....	175
15.2	Algorithm .....	175

15.3 Results.....	175
15.4 Comparison with Photoshop.....	177
15.5 Implementation for YCbCr and YCrCb.....	177
<b>16 Conclusion and Future Work.....</b>	<b>179</b>
16.1 Conclusion.....	179
16.2 Future Work .....	179
<b>References .....</b>	<b>181</b>
<b>Appendix: SIP API .....</b>	<b>185</b>



# List of Figures

Fig. 1.1	22 steps for creating a water reflection in Adobe Photoshop 7.0	15
Fig. 2.1	Bayer pattern	18
Fig. 2.2	H2V1 formats	19
Fig. 2.3	YCbCr 4:2:0	19
Fig. 4.1	Localized magnification	27
Fig. 4.2	Localized pinching	28
Fig. 4.3	Coordinate system for magnification and pinching	30
Fig. 4.4	Magnification results for varying $a$ and constant $k$	35
Fig. 4.5	Magnification results for varying $k$ and constant $a$	36
Fig. 4.6	Pinched results for varying $a$ and constant $k$	37
Fig. 4.7	Pinched results for varying $k$ and constant $a$	38
Fig. 4.8	Changing facial expressions and features	39
Fig. 4.9	Comparison with Photoshop for $a = 2, k = 1, m = 0$	40
Fig. 4.10	Comparison with Photoshop for $a = 2, k = 1.5, m = 0$	40
Fig. 4.11	Comparison with Photoshop for $a = 1.5, k = 1, m = 0$	41
Fig. 4.12	Comparison with Photoshop for $a = 2, k = 1, m = 1$	42
Fig. 4.13	Comparison with Photoshop for $a = 2, k = 1.5, m = 1$	42
Fig. 4.14	Comparison with Photoshop for $a = 1.5, k = 1, m = 1$	43
Fig. 4.15	H2V1 YCbCr 4:2:2	45
Fig. 4.16	Comparison of magnified images for 4 different color formats	46
Fig. 4.17	Comparison of pinched images for 4 different color formats	47
Fig. 5.1	Perspective drawings	49
Fig. 5.2	Planar geometric projections	50
Fig. 5.3	One-point perspective with x-axis vanishing point	52
Fig. 5.4	Two-point perspective with x- and z-axis vanishing points	52
Fig. 5.5	One-point perspective projection	53
Fig. 5.6	Two-dimensional perspective mapping	55
Fig. 5.7	Perspective result from Photoshop 7.0	55
Fig. 5.8	Perspective images for different vanishing point locations	56
Fig. 5.9	Coordinate system used in perspective	57
Fig. 5.10	Perspective result using the new algorithm	58
Fig. 5.11	H2V1 YCbCr 4:2:2	62
Fig. 5.12	Comparison of perspective images for 4 different color formats	63
Fig. 5.13	Perspective images with vanishing points close to camera	65
Fig. 5.14	Perspective images with vanishing points far from camera	66
Fig. 5.15	Perspective images with off-center top edge	66
Fig. 6.1	Neon sign	67
Fig. 6.2	Neon Effect	68

Fig. 6.3 Four 3 x 3 filter masks .....	68
Fig. 6.4 5 x 5 filter masks.....	69
Fig. 6.5 Results from Neon Effect algorithm .....	70
Fig. 6.6 Comparison with Photoshop 7.0.....	71
Fig. 7.1 Radial blur from Photoshop 7.0.....	73
Fig. 7.2 Radial blur in Cartesian coordinates .....	74
Fig. 7.3 Two sets of neighbors for adjacent pixels .....	75
Fig. 7.4 Region of interest.....	77
Fig. 7.5 Enlarged views of region of interest .....	78
Fig. 7.6 Regions of different rotational increments for $\alpha = 1^\circ$ .....	79
Fig. 7.7 Regions of different rotational increments for $2^\circ \leq \alpha \leq 5^\circ$ .....	80
Fig. 7.8 Regions of different rotational increments for $\alpha > 5^\circ$ .....	80
Fig. 7.9 Coordinate system used for radial blur .....	82
Fig. 7.10 Intermediate image rotated by $5^\circ$ counterclockwise .....	82
Fig. 7.11 Mapping from $\alpha$ to $\beta$ .....	83
Fig. 7.12 Original 520 x 390 image .....	87
Fig. 7.13 Radial blur results using different blur amounts .....	88
Fig. 7.14 Radial blur results using different centers of rotation .....	88
Fig. 7.15 Radial blur versus mixed blur .....	89
Fig. 7.16 Region of interest .....	89
Fig. 7.17 Radial blur comparison in region of interest.....	90
Fig. 7.18 Radial blur comparison with full-sized images .....	90
Fig. 7.19 H2V1 YCbCr 4:2:2 .....	91
Fig. 7.20 H2V1 YCrCb 4:2:2 .....	91
Fig. 7.21 Comparison of radial blur results for 4 different color formats .....	92
Fig. 8.1 Real-life motion blur .....	95
Fig. 8.2 Rotated corner positions .....	96
Fig. 8.3 Original 520 x 390 image .....	97
Fig. 8.4 Intermediate rotated result for $\alpha = 30^\circ$ .....	97
Fig. 8.5 Padding the left image boundary.....	98
Fig. 8.6 Blurred intermediate image for $\alpha = 30^\circ$ and $n = 25$ .....	99
Fig. 8.7 Final motion blur result for $\alpha = 30^\circ$ and $n = 25$ .....	99
Fig. 8.8 Motion blur results .....	100
Fig. 8.9 Comparison with Photoshop for $\alpha = 30^\circ$ and $n = 25$ .....	101
Fig. 8.10 Comparison with Photoshop for $\alpha = 0^\circ$ and $n = 35$ .....	101
Fig. 8.11 Comparison with Photoshop for $\alpha = 135^\circ$ and $n = 21$ .....	101
Fig. 8.12 Comparison of motion blur results for 4 different color formats.....	102
Fig. 9.1 Gaussian Blur from Photoshop 7.0 .....	103
Fig. 9.2 Comparison of Gaussian and uniform blur .....	104
Fig. 9.3 Cross-sectional view of Gaussian and uniform filter shapes .....	105
Fig. 9.4 Boundary padding .....	106
Fig. 9.5 Equivalent Gaussian blur results .....	107

Fig. 9.6 Gaussian blur.....	108
Fig. 9.7 Uniform blur.....	109
Fig. 9.8 Gaussian blur comparison with Photoshop.....	109
Fig. 9.9 Comparison of Gaussian blur results for 4 different color formats.....	110
Fig. 9.10 Comparison of uniform blur results for 4 different color formats.....	111
Fig. 10.1 22 steps for water reflection in Adobe Photoshop 7.0.....	113
Fig. 10.2 Water reflection in Photoshop 7.0.....	114
Fig. 10.3 Real water reflections.....	115
Fig. 10.4 Possible water boundaries.....	117
Fig. 10.5 Axes of reflection in real water reflections.....	118
Fig. 10.6 Coordinate system for Water Reflection algorithm.....	119
Fig. 10.7 Reflection images before blending.....	123
Fig. 10.8 Sample water images.....	124
Fig. 10.9 Effects of varying $\alpha$ .....	125
Fig. 10.10 Effects of varying $k$ with no perspective foreshortening.....	126
Fig. 10.11 Effects of varying $k$ with $c = H/4$ .....	127
Fig. 10.12 Effects of varying $k$ with $c = H/2$ .....	128
Fig. 10.13 Adding horizontal motion blur with a constant filter length.....	129
Fig. 10.14 Adding horizontal motion blur with a variable filter length.....	129
Fig. 10.15 Water reflections with horizontal linear boundaries.....	132
Fig. 10.16 Water reflection with downward sloping linear boundaries.....	133
Fig. 10.17 Water reflection with upward sloping linear boundaries.....	134
Fig. 10.18 H2V1 formats.....	135
Fig. 10.19 Comparison of water reflections for 4 different color formats.....	136
Fig. 11.1 Alpha blending using Photoshop 7.0.....	139
Fig. 11.2 Comparison of alpha blending results.....	140
Fig. 11.3 Fog.....	141
Fig. 11.4 Palm leaf shadow.....	141
Fig. 11.5 Four different orientations for palm leaf shadow.....	142
Fig. 11.6 Casting a palm leaf shadow.....	142
Fig. 11.7 Fade-in and fade-out.....	143
Fig. 12.1 Rotation by $45^\circ$ .....	145
Fig. 12.2 Coordinate system used for rotation equations.....	146
Fig. 12.3 Mapping $\alpha$ to $\beta$ .....	148
Fig. 12.4 H2V1 YCbCr 4:2:2.....	152
Fig. 12.5 Comparison of rotated images for 4 different color formats.....	153
Fig. 13.1 Downsizing by a factor of 3.....	155
Fig. 13.2 Comparison of results from combined shear.....	158
Fig. 13.3 Shear results.....	158
Fig. 13.4 Original 398 x 298 images.....	159
Fig. 13.5 Overlap result.....	159
Fig. 13.6 Original 398 x 298 images.....	160
Fig. 13.7 Film Strip.....	161

Fig. 13.8 Horizontal Composite .....	161
Fig. 13.9 Vertical Composite.....	162
Fig. 13.10 Center Composite .....	162
Fig. 13.11 Comparison of Shear with Photoshop 7.0.....	163
Fig. 13.12 Photoshop 7.0 parameter settings for Shear.....	164
Fig. 13.13 Shear results for 4 different color formats .....	165
Fig. 14.1 Typical corner fold .....	167
Fig. 14.2 Corner Fold effect .....	167
Fig. 14.3 Regions in Corner Fold output image .....	168
Fig. 14.4 Corner fold template.....	169
Fig. 14.5 Original 520 x 390 images .....	169
Fig. 14.6 520 x 390 Corner Fold results.....	170
Fig. 14.7 Original 398 x 298 images .....	170
Fig. 14.8 398 x 298 Corner Fold results .....	171
Fig. 14.9 Comparison of results for different values of $\alpha$ .....	171
Fig. 14.10 Comparison of Corner Fold results for 4 different color formats .....	172
Fig. 15.1 Color Change results.....	176

# List of Tables

TABLE 3.1 SIP Functions.....23  
TABLE 5.1 Y-Coordinate Mappings for  $k = 1.5$  .....60  
TABLE 5.2 Perspective Results .....64  
TABLE 7.1 Rotational Increments for Angle  $\alpha$  .....79  
TABLE 10.1 Selected Values of  $\delta$  for  $c = H/4$  .....127  
TABLE 10.2 Selected Values of  $\delta$  for  $c = H/2$  .....128



# Chapter 1

## Introduction

In recent years, there has been an explosion in the availability of digital cameras in the mass market. With the advent of everyday digital photography, ordinary consumers are enjoying the ease with which high-quality pictures can be taken and shared. Accompanying this is a new wave of innovation in image processing – the enhancement of digital photographs with powerful techniques unconstrained by chemistry, optics, and analog processes [13]. Whereas traditional cameras required the well-honed skills of master photographers to add aesthetically pleasing touches, digital cameras allow anyone to manipulate photographs expertly, given the right image processing software.

Camera phones add another dimension of complexity to digital image processing. There are now tight constraints on microprocessor speed, memory usage, and power consumption. Camera phone users expect all existing capabilities of handheld digital cameras and desktop graphics software while demanding immediate output on cell phone screens.



Fig. 1.1. The 22 steps for creating a water reflection in Adobe Photoshop 7.0 [19].

Existing software in digital cameras, both handheld and on cell phones, offer only a limited selection of image processing functions, ranging from contrast adjustment to stitching. Desktop graphics applications, such as Adobe Photoshop, offer a larger selection but require more advanced operations to be done manually. Fig. 1.1 shows an example of creating a water reflection in Photoshop 7.0, which requires 22 manual steps [19]. Only the first and last steps are shown for illustration purposes.

What is needed is a smart software library that can automatically perform advanced image processing operations with almost no human intervention and achieve real-time or near real-time performance on any hardware platform, including cell phone chips with integer microprocessors. This is one motivation for the Smart Image Processing (SIP) library, which provides sophisticated tools for automated digital image editing on camera phones.

The SIP library also features new solutions to long-standing problems in image processing and computer graphics. Five patents are pending for the new algorithms and implementations developed in this library. One problem solved by the SIP library is the detail-in-context problem, which is described in Chapter 4. A new unified approach to magnification and pinching is presented as a solution. Another example is the problem of real-time rotation of images using strictly integer arithmetic. A new solution is presented in Chapter 12. Other problems include transforming ordinary photographs to perspective view, creating realistic water reflections in varying weather conditions, creating translucent corner folds, adding high-quality radial blur, and directly processing subsampled YCbCr and YCrCb images.

This thesis is organized as follows: Chapter 2 provides background information on the SIP library, including existing image processing systems, color formats, YCbCr and YCrCb complications, and hardware constraints. A list of functions developed in the library is given in Chapter 3. Chapters 4 through 15 present algorithms, implementations, and results for each function. In particular, Chapters 4, 5, 7, 10, and 12 are self-contained and describe in detail the five pending patents. Chapter 16 concludes the thesis and provides ideas for future work. Finally, an application program interface (API) for all SIP functions is given in the Appendix.



# Chapter 2

## Background and Purpose

The purpose of this thesis is to build an advanced image processing library for easy, automated editing of digital photographs. All algorithms in the library are implemented with strictly integer arithmetic and achieve real-time or near real-time performance on actual cell phone chips. These algorithms have been optimized for the often severe hardware constraints found on camera phones, which generally have integer microprocessors. Common constraints include the unacceptable latency of integer microprocessors when performing floating-point arithmetic, the scarcity of memory space, and the limited power supply. However, all algorithms developed in this thesis should be suitable for any system – cellular, handheld, laptop, or desktop.

### 2.1 Existing System

The SIP library is an advanced upgrade for a pre-existing image processing library already used in commercial cell phones. The pre-existing library functions provide only a set of basic operations: 90° rotation, reflection, cropping, resizing, addition of frames, color format conversion, grayscale/negative/sepia conversion, posterization, solarization, merging of two images, standard histogram equalization, hue/saturation/intensity/contrast adjustment, and lowpass/sharpening filters. To maintain backward compatibility, the SIP library supports RGB, YCbCr, and YCrCb color formats.

### 2.2 Color Formats

A well known color image format is RGB, in which each pixel has three values representing the red, green, and blue components of light. However, this is not the most efficient data format. A better alternative is to use the luminance-chrominance color space, where luminance describes the intensity of light and chrominance describes both hue and saturation. This allows for much higher compression rates with less reduction in image quality. Popular luminance-chrominance representations in camera phones are YCbCr and its variant, YCrCb, in which the order of Cb and Cr components is reversed. Y is called the luma, and Cb and Cr are the chroma. YCbCr and YCrCb are used extensively in the JPEG standard and have demonstrated good performance in image compression [32], [18].

Most camera sensors do not allow photographs to be taken directly in the YCbCr domain. Instead, sensors are often configured in the Bayer pattern, with individual sensor cells capable of detecting light intensity for only one of three colors: red, green, or blue. A

sample Bayer pattern from [37] is shown in Fig. 2.1. Notice that there are more green cells than red or blue cells. This arrangement boosts image quality since the human visual system is more sensitive to information in the green color channel.

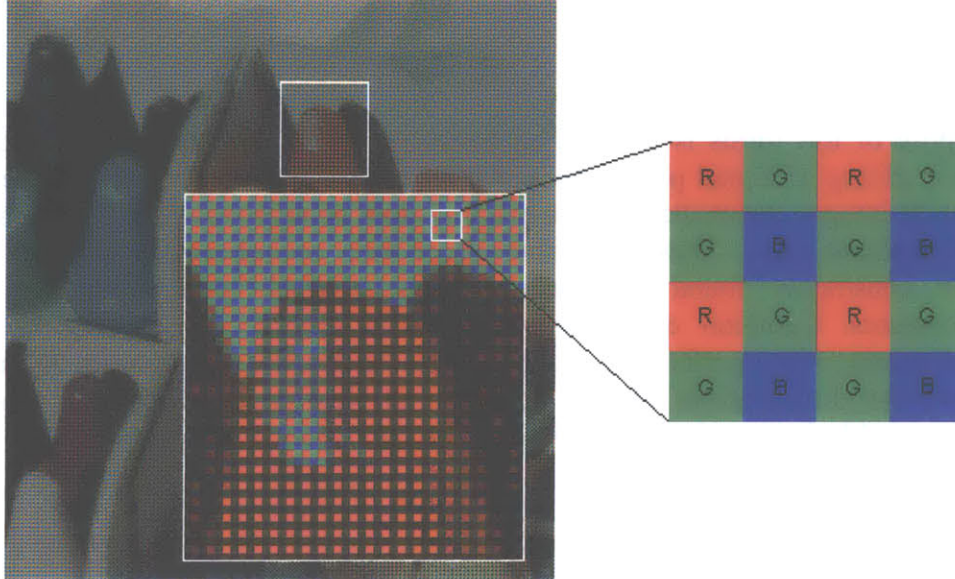


Fig. 2.1. A sample Bayer pattern for digital camera sensors with alternating rows of RGRG and GBGB [37].

Before a raw Bayer image can be transformed to YCbCr or YCrCb, data must be interpolated so that each image pixel has all three color components. The interpolated, or demosaiced, data is in RGB format, which is one reason why RGB remains widely used despite its mediocre compression properties. Equations (2.1) and (2.2) give the transformation matrices between RGB and YCbCr [34].

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \frac{1}{256} \begin{bmatrix} 298.082 & 0 & 408.583 \\ 298.082 & -100.291 & -208.120 \\ 298.082 & 516.411 & 0 \end{bmatrix} \cdot \left( \begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} - \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix} \right) \quad (2.1)$$

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix} + \frac{1}{256} \begin{bmatrix} 65.738 & 129.057 & 25.064 \\ -37.945 & -74.494 & 112.439 \\ 112.439 & -94.154 & -18.285 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix} \quad (2.2)$$

Similar equations can be found for conversion between RGB and YCrCb. Any practical image processing software should handle all three color formats. Some image processing algorithms are easier to develop for RGB. As will be explained later, YCbCr and YCrCb subsampled formats lead to complications in implementation.

Due to the small size of camera phones, additional space- and cost-saving techniques are greatly desired. One technique is to reduce the resolution of digital image data by subsampling the chrominance components in YCbCr and YCrCb [32]. For instance, subsampling by a factor of two discards every other chrominance sample, thereby halving the amount of memory space required. Visual quality is not significantly reduced since the human visual system is less sensitive to chrominance variation [27].

One common form of subsampled YCbCr is YCbCr 4:2:2, in which the Cb and Cr components of either pixel rows or pixel columns are subsampled by a factor of two. Two formats are H2V1 YCbCr 4:2:2, in which pixel columns are subsampled but pixel rows are unaffected, and H1V2 YCbCr 4:2:2, in which pixel rows are subsampled but pixel columns are unaffected.

Fig. 2.2 (a) shows the H2V1 format. For every 2 x 2 block of pixels, there are 4 Y values, 2 Cb values, and 2 Cr values, hence the name 4:2:2. If columns are numbered starting from zero, only even columns have the Cb component and only odd columns have Cr. Hereafter, YCbCr 4:2:2 refers to H2V1 YCbCr 4:2:2.

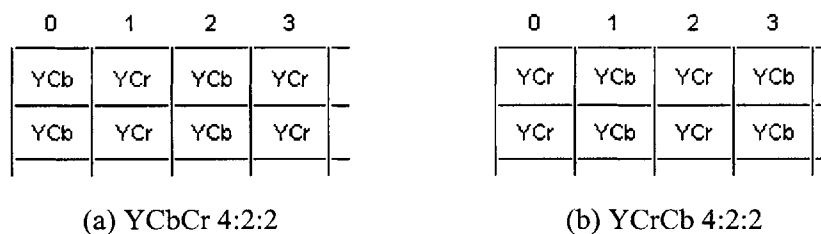


Fig. 2.2. H2V1 formats.

YCrCb formats are the same as YCbCr but with the order of Cb and Cr components switched. For instance, an H2V1 YCrCb 4:2:2 image has the pixel arrangement shown in Fig. 2.2 (b). Hereafter, YCrCb 4:2:2 refers to H2V1 YCrCb 4:2:2.

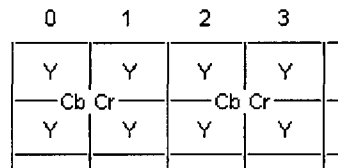


Fig. 2.3. YCbCr 4:2:0.

Another common form of subsampled YCbCr is YCbCr 4:2:0, in which each block of 2 x 2 pixels has only one Cb and one Cr component, as illustrated in Fig. 2.3. All four pixels in a block share the same two chrominance components. The corresponding YCrCb format is YCrCb 4:2:0, which has the order of Cb and Cr components switched.

## 2.3 Complications from Subsampled YCbCr and YCrCb

The loss of chrominance information in subsampled YCbCr and YCrCb images causes difficulties during implementation. Cropping, downsizing, and more advanced operations in the SIP library are difficult to perform on these images. Traditional methods of converting to RGB and processing images in the RGB domain are too computationally expensive for camera phones. Faster methods are needed that process images directly in the YCbCr domain.

### 2.3.1 Simple but Limited Solutions

For a subsampled YCbCr or YCrCb image, a simple task, such as cropping the image, becomes problematic when boundaries happen to fall on odd-numbered columns. This is because displaying the image requires reading pairs of YCb-YCr columns. If a boundary is placed so that an unpaired column remains, undesirable boundary artifacts will appear. To prevent this, a simple solution is to shift the boundary by one pixel so that no unpaired columns remain. This is a quick and simple strategy that requires very little computation. The technique has been used to successfully implement functions described in Chapter 13.

A second complication arises for downsizing. A naïve downsizing algorithm would simply find the width and height factors by taking the ratios of the original and downsized image dimensions, and use these factors to map pixels in the original image to pixels in the downsized image. However, if the downsized image is in YCbCr 4:2:2 format, pixel columns must alternate between YCb and YCr. The naïve approach might assign 2 YCb columns or 2 YCr columns in the original image to adjacent columns in the downsized result, thus creating mismatched pairs. To remedy this, one solution is to check for incorrectly paired YCb-YCb or YCr-YCr column mappings and fix the pair by substituting the nearest YCr or YCb column from the original image.

### 2.3.2 An Advanced Systematic Approach

The above ad-hoc methods of dealing with subsampled YCbCr and YCrCb are useful only for simple operations, such as cropping and downsizing. A more systematic approach is needed for more advanced operations that require the use of transformation equations or multi-step processing.

This thesis provides a new, effective solution that is broadly applicable and avoids extensive computation. The solution is to create a temporary YCbCr 4:4:4 (YCrCb 4:4:4) image from the original subsampled YCbCr (YCrCb) image. Transformation equations and intermediate processing are applied to the YCbCr 4:4:4 (YCrCb 4:4:4) image to produce a temporary YCbCr 4:4:4 (YCrCb 4:4:4) output, whose extra chrominance components are discarded to obtain the final subsampled YCbCr (YCrCb) output.

A YCbCr 4:4:4 image contains all three YCbCr components for each pixel. If the original image is YCbCr 4:2:2, the temporary YCbCr 4:4:4 image is created by taking pairs of adjacent YCb and YCr pixels and having each pixel borrow the missing chrominance component from its partner. The YCb pixel borrows a Cr component from its paired YCr pixel. The YCr pixel borrows a Cb component from the YCb pixel. Alternatively, if the original image is YCbCr 4:2:0, the Cb and Cr components are duplicated for each pixel in each 2 x 2 block.

Normally, this method of chrominance duplication does not produce an accurate, visually pleasing YCbCr 4:4:4 image from a subsampled YCbCr image. However, since the temporary YCbCr 4:4:4 image is used only for intermediate processing, the accuracy of the intermediate result does not matter as long as the final output is accurate. The final subsampled YCbCr output is obtained from the temporary YCbCr 4:4:4 output by discarding extra Cb and Cr components for each pixel.

This method has been applied successfully in all SIP functions that require special treatment of subsampled YCbCr and YCrCb images. There is no visible difference between a subsampled YCbCr or YCrCb output image processed in this manner and an equivalent RGB output image. All SIP functions that use the method offer comparisons of RGB output with subsampled YCbCr and YCrCb output to demonstrate the effectiveness and broad applicability of this method.

## **2.4 Hardware Constraints**

Although advanced image processing algorithms should be fast and effective, some can still be computationally intensive. Standard image sizes, sometimes on the order of megapixels, exacerbate the situation. Heavy responsibility therefore rests on the implementer to produce efficient, optimized code. In the cell phone industry, integer microprocessors are often used to reduce silicon cost and power consumption [2], thus hindering the use of floating-point arithmetic. Although a floating-point emulator or software library may be used, the additional latency is often unacceptable. Algorithms implemented in the SIP library must use strictly fixed-point arithmetic to minimize the number of core instruction cycles. Unfortunately, this may come at the expense of code length.

Due to the limited processing power and battery life of cell phones, additional minimization of memory access and power consumption is necessary. Careful code profiling is needed to find the right balance in the number of external memory accesses and cache misses. A decrease in memory access produces a decrease in power consumption but at the expense of increased cache misses and execution time. Some assembly code optimization may be needed to find the best tradeoff, which can depend on the specific on-chip architecture.

All algorithms developed in this project have been implemented on the ARM926EJ-S microprocessor. ARM926EJ-S, which has a 32-bit RISC CPU, is a general-purpose





integer microprocessor targeted at multi-tasking applications where full memory management, high performance, and low power are all important [3]. Code optimization has been done using the ARMulator, a software emulator for the microprocessor.


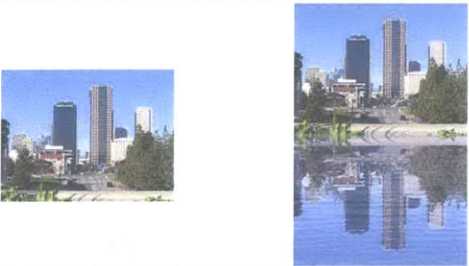


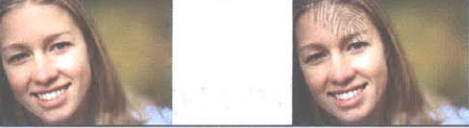






# Chapter 3

## Functions Developed




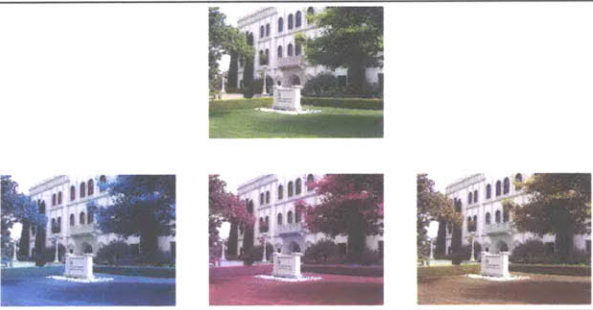
The library functions developed in this thesis are listed in TABLE 3.1 in the order presented in subsequent chapters. All functions have been incorporated in an image processing software application embedded in existing camera phone systems.

TABLE 3.1: SIP Functions.

Function	Example	Description	Chapter
Magnify	 	Magnify an arbitrary circular region.	4
Pinch	 	Pinch an arbitrary circular region.	4
Perspective	 	Transform to perspective view.	5
Neon Effect	 	Create a neon image.	6
Spin Radial Blur	 	Add spin radial blur.	7
Motion Blur	 	Add motion blur.	8
Gaussian Blur	 	Add Gaussian blur.	9

Uniform Blur		Add uniform blur.	9
Water Reflection		Create realistic water reflections in varying weather conditions.	10
Alpha Blending		Perform alpha blending.	11
Fog		Add fog.	11
Shadow		Cast a shadow.	11
Fade In, Fade Out		Make an image sequence fade to or from a still image.	11
Arbitrary Rotation		Rotate an image.	12
Shear		Shear an image.	13
Overlap		Overlap one image on top of another.	13
Film Strip		Combine parts of two images to simulate a moving film strip.	13
Horizontal Composite		Combine the left side of one image with the right side of another image.	13



Vertical Composite		Combine the top part of one image with the bottom part of another image.	13
Center		Combine the center part of one image with the sides of another image.	13
Corner Fold		Fold over a corner and display a translucent version of the folded region.	14
Color Change		Change the color mapping.	15



# Chapter 4

## Magnification and Pinching (Patent Pending)

### 4.1 Introduction

Magnification is a solution to a common problem associated with visual display terminals. This is the detail-in-context problem, which is to find the best use of limited window space in a graphical user interface so that detailed information can be displayed without losing a sense of context. One instance of the problem occurs when viewing a large image in a display terminal such as a cell phone screen. In such a situation, the user is confronted with two conflicting desires: to zoom in on an image and see fine details, or to view the entire image and maintain a sense of context. Zooming in on a large image allows only a portion of the image to be displayed and forces the viewer to lose sight of the larger picture. Viewing the entire image reduces the resolution quality and forces the viewer to forgo information hidden in finer details. This is an unavoidable tradeoff in an interface in which the user can only zoom in to obtain a more detailed and more restricted view. Evidence in [8] and [38] suggest that such an interface makes users waste significant amounts of time zooming in and out to re-establish their context in the original image.



(a) Original Image



(b) Magnified Image

Fig. 4.1. Localized magnification. Image size is 520 x 390 pixels.

Localized magnification solves the problem by displaying a local magnified region inside a global view of a full image so that both detail and context are simultaneously visible.

Fig. 4.1 provides an example. Closely related to magnification is the concept of pinching, which contracts a part of an image. Pinching is useful for image manipulation, especially for warping face images to change facial expressions and features [30]. A pinched image is shown in Fig. 4.2. Both magnification and pinching also serve the aesthetic purpose of producing interesting effects for visual entertainment.

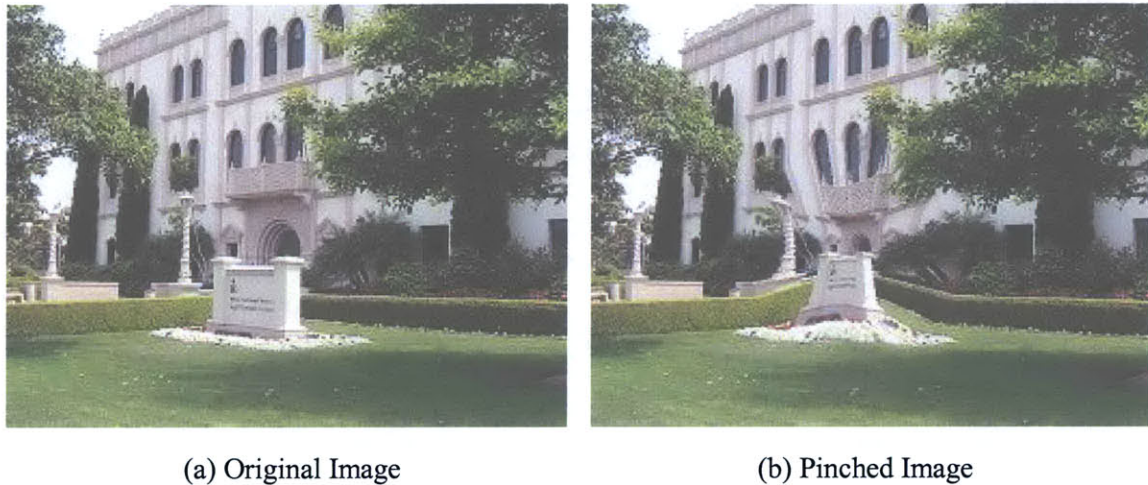


Fig. 4.2. Localized pinching. Image size is 520 x 390 pixels.

The existing literature provides many different approaches to magnification and pinching of images for effective visualization. These include polyfocal projection, bifocal display, fisheye views, rubber sheet, distortion-oriented presentation, and focus + context [20]. The variety of techniques can be grouped into two main categories, based on the type of transformation function used to create the effect.

Linear, or non-distortion based, transformations are the more familiar variety. This type of transformation produces a constant level of magnification or contraction across the image, with all regions equally enlarged or reduced. Unfortunately, there are inherent disadvantages in this approach. Uniformly magnifying a selected region fails to emphasize details in the center of the region, which is the most important area of focus for the user. Furthermore, if a magnified region is displayed on top of a normal-resolution image, a large part of the original selected region will be discarded. For instance, if the selected region is magnified by a factor of three, two thirds of it will be discarded in order to fit the magnified version within the boundaries of the original region.

Alternatively, if the magnified region is displayed in another part of the screen, such as in a separate window, to allow all of the original region to be seen, the user is forced to make abrupt transitions on two cognitive levels to reconcile the two representations. On a spatial level, the user must shift from the normal view window to the zoom window. On a resolution level, the user must create a mental mapping between the two different levels

of resolution to maintain a sense of context [21]. Even when there is a box glyph showing the location of the local zoom window within a normal view window, the user must translate scrolling movements within the zoom window to the normal window in a non-intuitive way [22]. In addition, linear transformations cannot be used to warp face images since a magnified region must be displayed on top of the original image and large parts of the face would be lost during the magnification process.

Nonlinear, or distortion-based, transformations form the second category of magnification and pinching techniques. This type of transformation changes the local resolution while preserving global context. For the special case of localized magnification, the concurrent presentation of local detail in magnified view together with global context in normal view allows the user to interactively position a focus region in a larger image without losing sight of spatial relationships. Many techniques for nonlinear transformation have been presented in the literature, including bifocal display, fisheye view, and polyfocal projection. These techniques are mainly intended for magnification of a focus region, where detailed information is displayed [36]. Outside this focus region is generally a transition region that provides a demagnified boundary area between the magnified region and the rest of the image. Demagnification of the transition region is necessary to reduce boundary discontinuities and loss of information in the magnified area. Unlike linear transformations, nonlinear transformations can be readily used to warp face images.

One deficiency in existing techniques for nonlinear transformation is that they are intended mainly for magnification, not for pinching. Both magnification and pinching are needed to warp facial expressions and features [30]. Pinching is simply the opposite of magnification. In a pinched image, there should be a focus region, where part of the original image is shown in contracted view. The pinched image should also have a transition region with a magnified boundary area connecting the pinched region and the rest of the image, so that no tears or gaps result from pinching. Ideally, because pinching and magnification are closely related, there should exist a single nonlinear transformation that provides both effects.

The image processing library developed in this thesis solves the problem by providing a simple, unified approach for creating both magnified and pinched images using a single nonlinear transformation. A simple binary parameter in the transformation functions controls whether magnification or pinching occurs.

## **4.2 A New Approach**

A nonlinear transformation is created by applying a mathematical function, called a transformation function, to an image. The transformation function defines how the original image is mapped to a desired view and uniquely identifies the nonlinear transformation. In the case of two-dimensional images, two transformation functions are needed, one for each dimension.

To present transformation functions for the new approach, we must first specify a coordinate system for the equations. Suppose the coordinate system has an origin located at the lower left corner of the image, an x axis pointing to the right, and a y axis pointing upward, as shown in Fig. 4.3. If the selected region has a center located at  $(x_o, y_o)$  and a radius  $R$ , the two-dimensional transformation functions are

$$x_{out} = \begin{cases} x_o + (x_{in} - x_o) \cdot a \left[ 1 - \frac{(x_{in} - x_o)^2 + (y_{in} - y_o)^2}{R^2} \right]^k & \text{for } (x_{in} - x_o)^2 + (y_{in} - y_o)^2 \leq R^2 \\ x_{in} & \text{otherwise} \end{cases}$$

$$y_{out} = \begin{cases} y_o + (y_{in} - y_o) \cdot a \left[ 1 - \frac{(x_{in} - x_o)^2 + (y_{in} - y_o)^2}{R^2} \right]^k & \text{for } (x_{in} - x_o)^2 + (y_{in} - y_o)^2 \leq R^2 \\ y_{in} & \text{otherwise} \end{cases}$$

where  $(x_{in}, y_{in})$  is the input pixel location,  $(x_{out}, y_{out})$  is the output pixel location, and parameters  $a$  and  $k$  control the level and type of distortion. The ranges for  $a$  and  $k$  are  $0 < a < \infty$  and  $-\infty < k < \infty$ . When  $0 < a < 1$ , pinching occurs. When  $1 < a < \infty$ , magnification occurs. When  $a = 1$ , the transformed image is identical to the original image. In these transformation equations, output location  $(x_{out}, y_{out})$  must be within the circular region  $(x_{out} - x_o)^2 + (y_{out} - y_o)^2 \leq R^2$ . For points outside this region,  $x_{out} = x_{in}$  and  $y_{out} = y_{in}$ .

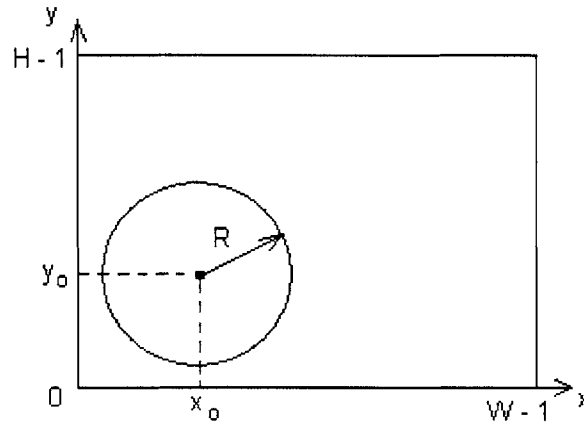


Fig. 4.3. Coordinate system for magnification and pinching in equations (4.1) and (4.2). Image size is  $W \times H$  pixels.

To better understand the effects of varying parameter  $a$ , we need a way to separate its effects on the level of distortion from its effects on the type of distortion. This can be done by restricting  $a$  to the range  $1 \leq a < \infty$  and introducing a new binary parameter  $m$  that determines whether magnification ( $m = 0$ ) or pinching ( $m = 1$ ) occurs. When limited to  $1 \leq a < \infty$ , parameter  $a$  will only affect the level of magnification or pinching, not the type of distortion. The new transformation functions are

$$x_{out} = \begin{cases} x_o + (x_{in} - x_o) \cdot a & \text{for } (x_{in} - x_o)^2 + (y_{in} - y_o)^2 \leq R^2 \\ x_{in} & \text{otherwise} \end{cases} \quad (4.1)$$

$$y_{out} = \begin{cases} y_o + (y_{in} - y_o) \cdot a & \text{for } (x_{in} - x_o)^2 + (y_{in} - y_o)^2 \leq R^2 \\ y_{in} & \text{otherwise} \end{cases} \quad (4.2)$$

These new transformation functions are identical to the original functions since

$$a \left[ 1 - \frac{(x_{in} - x_o)^2 + (y_{in} - y_o)^2}{R^2} \right]^k = \left( \frac{1}{a} \right) \left[ 1 - \frac{(x_{in} - x_o)^2 + (y_{in} - y_o)^2}{R^2} \right]^k.$$

If  $a$  varies in the range  $1 \leq a < \infty$ ,  $\frac{1}{a} = a^{-1}$  varies in the range  $0 < \frac{1}{a} < 1$ . Thus, if we restrict  $a$  to the range  $1 \leq a < \infty$  and use a negative exponent by setting  $m = 1$  in equations (4.1) and (4.2), it is equivalent to varying  $a$  in the range  $0 < a < 1$  in the original transformation functions. Alternatively, setting  $m = 0$  to get a positive exponent is equivalent to varying  $a$  in the range  $1 \leq a < \infty$  in the original transformation functions. By changing the value of  $m$ , the new transformation functions in equations (4.1) and (4.2) cover exactly the same range of  $a$  as the original transformation functions.

Equations (4.1) and (4.2) can be modified to apply the nonlinear transformation in a single dimension only. For instance, if the transformation is applied only in the horizontal direction, the equations become

$$x_{out} = \begin{cases} x_o + (x_{in} - x_o) \cdot a & \text{for } |x_{in} - x_o| \leq d \\ x_{in} & \text{otherwise} \end{cases}$$

$$y_{out} = y_{in}$$

where  $d$  is half the width of the region of interest.  $x_{out}$  must be in the range  $|x_{out} - x_o| \leq d$ . Outside this range,  $x_{out} = x_{in}$ .

To apply the transformation only in the vertical direction, the equations are

$$x_{out} = x_{in}$$

$$y_{out} = \begin{cases} y_o + (y_{in} - y_o) \cdot a^{(-1)^m \left[ 1 - \frac{(y_{in} - y_o)^2}{d^2} \right]^k} & \text{for } |y_{in} - y_o| \leq d \\ y_{in} & \text{otherwise} \end{cases}$$

where  $d$  is half the height of the region of interest.  $y_{out}$  must be in the range  $|y_{out} - y_o| \leq d$ . Outside this range,  $y_{out} = y_{in}$ .

To better understand equations (4.1) and (4.2), let us consider the example of  $a = 2$ ,  $k = 1$ , and  $m = 0$ . In this case, the transformation functions are

$$x_{out} = \begin{cases} x_o + (x_{in} - x_o) \cdot 2^{\left[ 1 - \frac{(x_{in} - x_o)^2 + (y_{in} - y_o)^2}{R^2} \right]} & \text{for } (x_{in} - x_o)^2 + (y_{in} - y_o)^2 \leq R^2 \\ x_{in} & \text{otherwise} \end{cases} \quad (4.3)$$

$$y_{out} = \begin{cases} y_o + (y_{in} - y_o) \cdot 2^{\left[ 1 - \frac{(x_{in} - x_o)^2 + (y_{in} - y_o)^2}{R^2} \right]} & \text{for } (x_{in} - x_o)^2 + (y_{in} - y_o)^2 \leq R^2 \\ y_{in} & \text{otherwise} \end{cases} \quad (4.4)$$

where  $(x_{out}, y_{out})$  must be in the circular region  $(x_{out} - x_o)^2 + (y_{out} - y_o)^2 \leq R^2$ . Outside this region,  $x_{out} = x_{in}$  and  $y_{out} = y_{in}$ .

Equations (4.3) and (4.4) produce a locally magnified image with a maximum magnification power of 2. At the center of the magnified region, where  $(x_{in}, y_{in}) = (x_o, y_o)$ , the power-of-2 term in these equations is equal to 2. The center is therefore magnified by a factor of 2. At the edge of the magnified region, where



$(x_{in} - x_o)^2 + (y_{in} - y_o)^2 = R^2$ , the power term becomes 1. Pixels on the edge are therefore unmagnified. The overall effect of equations (4.3) and (4.4) is to provide a magnification power that equals 2 at the center of the selected region and gradually decreases as the distance from the center increases.

For completeness, let us also consider an example of pinching with  $a = 2$ ,  $k = 1$ , and  $m = 1$ . In this case, the transformation functions are

$$x_{out} = \begin{cases} x_o + (x_{in} - x_o) \cdot 2 \left[ 1 - \frac{(x_{in} - x_o)^2 + (y_{in} - y_o)^2}{R^2} \right] & \text{for } (x_{in} - x_o)^2 + (y_{in} - y_o)^2 \leq R^2 \\ x_{in} & \text{otherwise} \end{cases} \quad (4.5)$$

$$y_{out} = \begin{cases} y_o + (y_{in} - y_o) \cdot 2 \left[ 1 - \frac{(x_{in} - x_o)^2 + (y_{in} - y_o)^2}{R^2} \right] & \text{for } (x_{in} - x_o)^2 + (y_{in} - y_o)^2 \leq R^2 \\ y_{in} & \text{otherwise} \end{cases} \quad (4.6)$$

where  $(x_{out}, y_{out})$  must be in the circular region  $(x_{out} - x_o)^2 + (y_{out} - y_o)^2 \leq R^2$ . Outside this region,  $x_{out} = x_{in}$  and  $y_{out} = y_{in}$ . Notice that the exponent in the power of 2 is now negative.

Equations (4.5) and (4.6) produce a locally pinched image with a maximum pinching factor of 2. At the center of the pinched region, where  $(x_{in}, y_{in}) = (x_o, y_o)$ , the power-of-2 term is equal to 1/2. The center is therefore pinched by a factor of 2. At the edge of the pinched region, where  $(x_{in} - x_o)^2 + (y_{in} - y_o)^2 = R^2$ , the power term becomes 1. Pixels on the edge are therefore unpinched. The overall effect of equations (4.5) and (4.6) is to provide a pinching factor that equals 2 at the center of the selected region and gradually decreases as the distance from the center increases.

In an actual implementation of equations (4.1) and (4.2), not every point in the output image may be assigned a corresponding point in the input image. If there are gaps in the output image, some form of interpolation must be used to fill in the missing pixels. Each missing pixel can be filled by simply duplicating the nearest neighboring pixel or by a distance-weighted average of a group of neighboring pixels.

Equations (4.1) and (4.2) can be extended to support an elliptical region of magnification or pinching, rather than a strictly circular region, as follows:

$$x_{out} = \begin{cases} x_o + (x_{in} - x_o) \cdot a & \text{for } b(x_{in} - x_o)^2 + c(y_{in} - y_o)^2 \leq R^2 \\ x_{in} \cdot (-1)^m \left[ 1 - \frac{b(x_{in} - x_o)^2 + c(y_{in} - y_o)^2}{R^2} \right]^k & \text{otherwise} \end{cases}$$

$$y_{out} = \begin{cases} y_o + (y_{in} - y_o) \cdot a & \text{for } b(x_{in} - x_o)^2 + c(y_{in} - y_o)^2 \leq R^2 \\ y_{in} \cdot (-1)^m \left[ 1 - \frac{b(x_{in} - x_o)^2 + c(y_{in} - y_o)^2}{R^2} \right]^k & \text{otherwise} \end{cases}$$

Two extra parameters  $b$  and  $c$  are needed. These determine the semimajor and semiminor axes of the ellipse, which are equal to  $b \cdot R$  and  $c \cdot R$ , respectively.

### 4.3 Case Studies

This section presents case studies on the effects of various parameter values for  $a$ ,  $k$ , and  $m$  in equations (4.1) and (4.2). The visual effects of changing these parameters are illustrated with sample images. For simplicity, all illustrated cases use only nearest-neighbor duplication to fill in missing pixels. All images are 520 x 390 pixels.

#### 4.3.1 Magnify

To magnify an image, parameter  $m$  in equations (4.1) and (4.2) must be set to 0. Parameters  $a$  and  $k$  can be varied to achieve different levels of magnification power and distortion. The effects of changing these two parameters can be best studied by varying one parameter at a time and observing changes in the magnified image.

##### 4.3.1.1 Magnification Variations in $a$

Holding  $k$  fixed at 1, we can vary  $a$  in the range  $1 \leq a < \infty$ . Fig. 4.4 shows images with a magnification region of radius 100 pixels. Fig. 4.4 (a), (b), (c), and (d) correspond to  $a = 1, 1.5, 2,$  and  $3,$  respectively.

The figure shows that as  $a$  increases, the magnification power increases. When  $a = 1$ , no magnification occurs. Hence, Fig. 4.4 (a) shows the original image. When  $a = 1.5$ , the center of the magnified region is slightly magnified by a power of 1.5. When  $a$  increases to 3, the center of the magnified region is noticeably magnified by a power of 3. Parameter  $a$  can be interpreted as controlling the power of magnification in the center of the magnified region.



(a)  $a = 1, k = 1, m = 0$   
(same as original image)



(b)  $a = 1.5, k = 1, m = 0$



(c)  $a = 2, k = 1, m = 0$



(d)  $a = 3, k = 1, m = 0$

Fig. 4.4. Magnification results for various values of  $a$  when holding  $k$  constant.

#### 4.3.1.2 Magnification Variations in $k$

Similarly, by holding  $a$  fixed at 2, we can vary  $k$  in the range  $-\infty < k < \infty$ . Fig. 4.5 shows images with a magnification region of radius 100 pixels. Fig. 4.5 (a), (b), (c), and (d) correspond to  $k = 1, 1.5, 2,$  and  $3,$  respectively.

The figure shows that as  $k$  increases, the transition area between the magnified region and the rest of the image becomes smoother and wider. A small value of  $k$ , such as  $k = 1$ , produces a very abrupt transition, with a large amount of demagnification applied to a very narrow transition region. A larger value of  $k$ , such as  $k = 3$ , produces a much more gradual transition, with a lower level of demagnification applied to a wider transition region. Parameter  $k$  can be interpreted as controlling the degree of distortion and the size of the transition region.



(a)  $a = 2, k = 1, m = 0$



(b)  $a = 2, k = 1.5, m = 0$



(c)  $a = 2, k = 2, m = 0$



(d)  $a = 2, k = 3, m = 0$

Fig. 4.5. Magnification results for various values of  $k$  when holding  $a$  constant.

### 4.3.2 Pinch

To pinch an image, parameter  $m$  in equations (4.1) and (4.2) must be set to 1. Parameters  $a$  and  $k$  can be varied to achieve different levels of pinching. The effects of changing these two parameters can be best studied by varying one parameter at a time and observing changes in the pinched image.

#### 4.3.2.1 Pinching Variations in $a$

Holding  $k$  fixed at 1, we can vary  $a$  in the range  $1 \leq a < \infty$ . Fig. 4.6 shows images with a pinched region of radius 100 pixels. Fig. 4.6 (a), (b), (c), and (d) correspond to  $a = 1, 1.5, 2,$  and  $3,$  respectively.

The figure shows that as  $a$  increases, the amount of pinching increases. When  $a = 1$ , no pinching occurs. Hence, Fig. 4.6 (a) shows the original image. When  $a = 1.5$ , the center of the pinched region is slightly pinched by a factor of 1.5. When  $a$  increases to 3, the center of the pinched region is much more noticeably pinched by a factor of 3. Parameter  $a$  can be interpreted as controlling the degree of pinching in the center of the pinched region.



(a)  $a = 1, k = 1, m = 1$   
(same as original image)



(b)  $a = 1.5, k = 1, m = 1$



(c)  $a = 2, k = 1, m = 1$



(d)  $a = 3, k = 1, m = 1$

Fig. 4.6. Pinched results for various values of  $a$  when holding  $k$  constant.

#### 4.3.2.2 Pinching Variations in $k$

Similarly, by holding  $a$  fixed at 2, we can vary  $k$  in the range  $-\infty < k < \infty$ . Fig. 4.7 shows images with a pinched region of radius 100 pixels. Fig. 4.7 (a), (b), (c), and (d) correspond to  $k = 1, 1.5, 2$ , and 3, respectively.

The figure shows that as  $k$  increases, the pinching effect decreases in the transition region around the rim of the pinched area. A small value of  $k$ , such as  $k = 1$ , produces a clearly noticeable distortion in the transition region. A larger value of  $k$ , such as  $k = 3$ , produces much less distortion. The difference can be seen by comparing the lamppost to the left of the arch and sign in Fig. 4.7 (a) with the same lamppost in Fig. 4.7 (d). In Fig. 4.7 (a), the lamppost is visibly bent and distorted. In Fig. 4.7 (d), it is straight and virtually undistorted. Parameter  $k$  can be interpreted as controlling the degree of distortion in the transition region.



(a)  $a = 2, k = 1, m = 1$



(b)  $a = 2, k = 1.5, m = 1$



(c)  $a = 2, k = 2, m = 1$



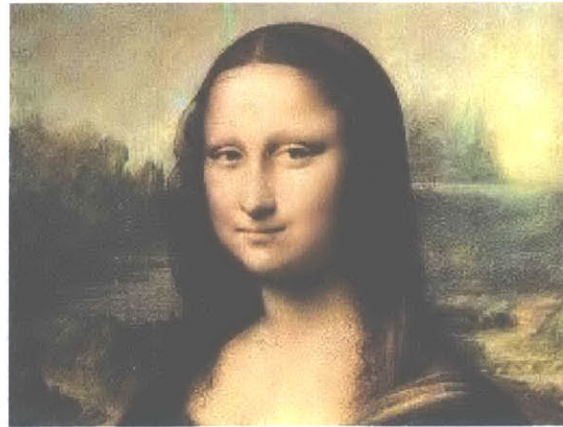
(d)  $a = 2, k = 3, m = 1$

Fig. 4.7. Pinched results for various values of  $k$  when holding  $a$  constant.

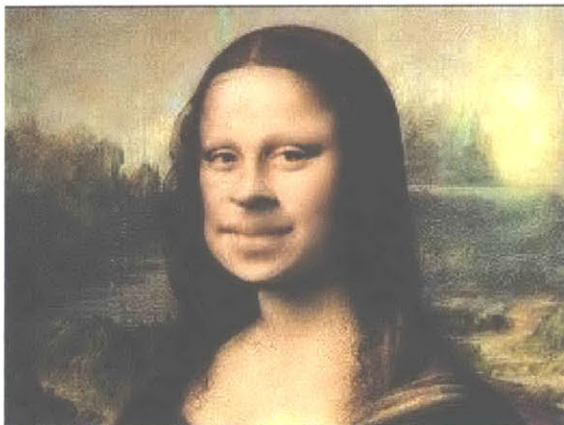
### 4.3.3 Changing Facial Expressions and Features

Both magnification and pinching can be used to change the facial expression and features of a face image. Fig. 4.8 shows two warped faces. Fig. 4.8 (a) shows the original Mona Lisa image. Fig. 4.8 (b) was obtained by magnifying a circular region of radius 60 pixels

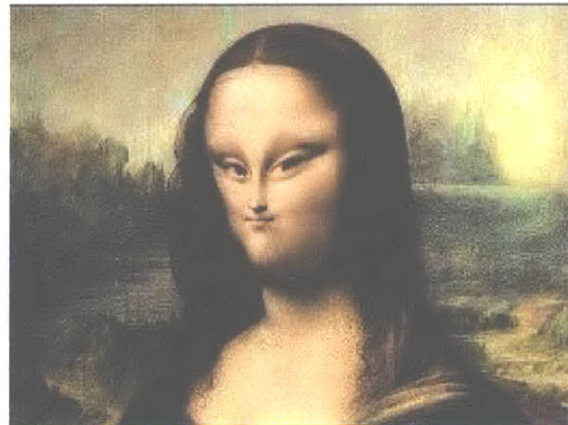
around the mouth of the Mona Lisa using parameters  $a = 2$ ,  $k = 3$ , and  $m = 0$ . Fig. 4.8 (c) was obtained by pinching a circular region of radius 70 pixels around the nose using parameters  $a = 2$ ,  $k = 1$ , and  $m = 1$ . Combinations of pinching and magnification can produce even more interesting facial changes.



(a) Original Image



(b)  $a = 2$ ,  $k = 3$ ,  $m = 0$ ,  $R = 60$



(c)  $a = 2$ ,  $k = 1$ ,  $m = 1$ ,  $R = 70$

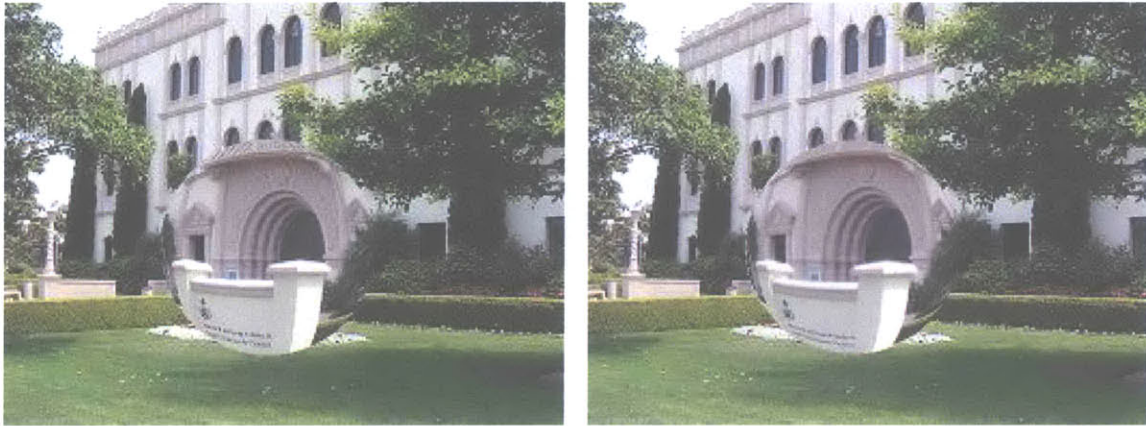
Fig. 4.8. Changing facial expressions and features.

#### 4.4 Comparison with Photoshop

In Adobe Photoshop 7.0, the closest approximations to the magnification and pinching effects produced by equations (4.1) and (4.2) are two functions called Spherize and Pinch. Neither Spherize nor Pinch can match all the results of equations (4.1) and (4.2). Attempts to duplicate the examples illustrated in Section 3 succeeded only for a few special cases described in this section. No other cases in Section 3 could be reproduced in Photoshop. All sample results from equations (4.1) and (4.2) were produced using nearest-neighbor duplication to fill in missing pixels. All images are 520 x 390 pixels.

#### 4.4.1 Magnify

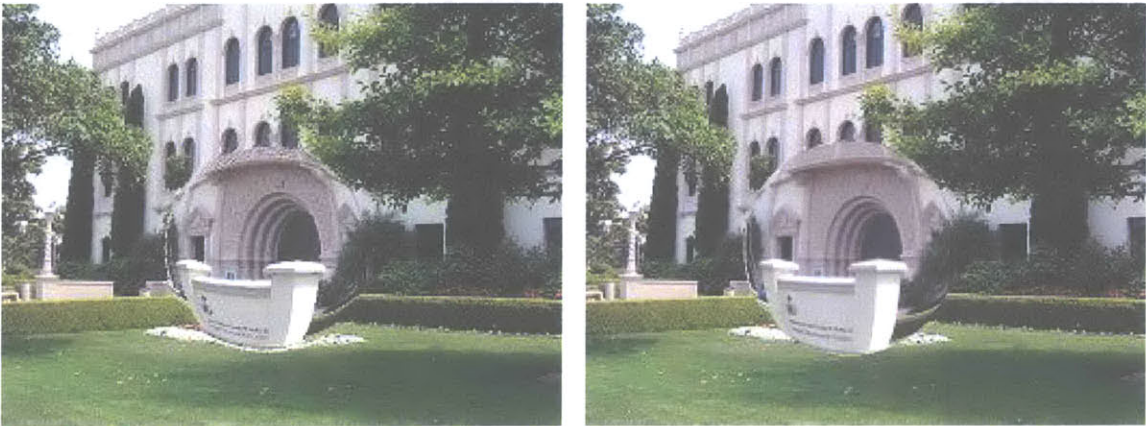
Besides the trivial case of  $a = 1$ ,  $k = 1$ ,  $m = 0$ , which produces an unmagnified image, only three cases of magnification are reproducible in Photoshop. These are shown in Figs. 4.9, 4.10, and 4.11.



(a)  $a = 2$ ,  $k = 1$ ,  $m = 0$

(b) Photoshop Spherize 100%, then 75%

Fig. 4.9. Comparison of results for  $a = 2$ ,  $k = 1$ , and  $m = 0$  with a Photoshop approximation.



(a)  $a = 2$ ,  $k = 1.5$ ,  $m = 0$

(b) Photoshop Spherize 100%, then 38%

Fig. 4.10. Comparison of results for  $a = 2$ ,  $k = 1.5$ , and  $m = 0$  with a Photoshop approximation.

In Fig. 4.9, the parameters in equations (4.1) and (4.2) are  $a = 2$ ,  $k = 1$ , and  $m = 0$ . In Photoshop, the closest approximation is obtained by applying a Spherize operation twice,



once with a setting of 100% and again with 75%. In Fig. 4.10, the parameters are  $a = 2$ ,  $k = 1.5$ , and  $m = 0$ . In Photoshop, the closest approximation is obtained by applying Spherize twice, once with 100% and a second time with 38%. In Fig. 4.11, the parameters are  $a = 2$ ,  $k = 1.5$ , and  $m = 0$ . In Photoshop, the closest approximation is obtained by applying Spherize once with 90%.



(a)  $a = 1.5$ ,  $k = 1$ ,  $m = 0$

(b) Photoshop Spherize 90%

Fig. 4.11. Comparison of results for  $a = 1.5$ ,  $k = 1$ , and  $m = 0$  with a Photoshop approximation.

In two out of the three cases, more than one Photoshop operation is needed to achieve a result that takes only one step using equations (4.1) and (4.2). This demonstrates the ease and efficiency of this new approach. Moreover, the fact that Photoshop fails to duplicate any of the other magnification examples in Sections 4.3.1.1 and 4.3.1.2 indicates the power and flexibility of the new approach.

#### 4.4.2 Pinch

Besides the trivial case of  $a = 1$ ,  $k = 1$ ,  $m = 1$ , which produces an unpinched image, only three cases of pinching are reproducible in Photoshop. These are shown in Figs. 4.12, 4.13, and 4.14.

In Fig. 4.12, the parameters in equations (4.1) and (4.2) are  $a = 2$ ,  $k = 1$ , and  $m = 1$ . In Photoshop, the closest approximation is obtained by applying a Pinch operation with a setting of 90%. In Fig. 4.13, the parameters are  $a = 2$ ,  $k = 1.5$ , and  $m = 1$ . In Photoshop, the closest approximation is obtained by applying Pinch with 70%. In Fig. 4.14, the parameters are  $a = 2$ ,  $k = 1.5$ , and  $m = 1$ . In Photoshop, the closest approximation is obtained by applying Pinch with 50%.



(a)  $a = 2, k = 1, m = 1$



(b) Photoshop Pinch 90%

Fig. 4.12. Comparison of results for  $a = 2, k = 1,$  and  $m = 1$  with a Photoshop approximation.

Photoshop's Spherize cannot be used to produce the pinched images. Instead, an entirely different Photoshop operation, Pinch, is needed to produce pinched images created by the same set of transformation functions that created magnified images. This again demonstrates the power and flexibility of the new approach. Even with the combined options of two Photoshop functions, Spherize and Pinch, the rest of the pinched examples presented in Sections 4.3.2.1 and 4.3.2.2 cannot be duplicated.



(a)  $a = 2, k = 1.5, m = 1$



(b) Photoshop Pinch 70%

Fig. 4.13. Comparison of results for  $a = 2, k = 1.5,$  and  $m = 1$  with a Photoshop approximation.



(a)  $a = 1.5, k = 1, m = 1$



(b) Photoshop Pinch 50%

Fig. 4.14. Comparison of results for  $a = 1.5, k = 1$ , and  $m = 1$  with a Photoshop approximation.

## 4.5 Integer Implementation

A problem arises if the transformation equations for magnification and pinching must be implemented on an integer microprocessor where floating-point arithmetic produces unacceptable latency. Under such circumstances, a method must be found to calculate the power functions in equations (4.1) and (4.2) using strictly integer arithmetic. The solution is to use a Taylor series expansion to approximate the power function. In general, the series expansion for an arbitrary power function is

$$a^n = 1 + (\ln a)n + \frac{(\ln a)^2}{2!}n^2 + \frac{(\ln a)^3}{3!}n^3 + \dots + \frac{(\ln a)^k}{k!}n^k + \dots \quad (4.7)$$

For the special case of  $a = 2, m = 0$ , and  $k = 1$ , as given by equations (4.3) and (4.4), the first four terms in the Taylor series are sufficient to accurately calculate the power of 2 for  $n > 0$ . Higher order terms can still be used if desired. However, these terms have little effect on accuracy and only decrease the speed of the implementation. Equations (4.3) and (4.4) were implemented with the series approximation

$$2^n \cong 1 + (\ln 2)n + \frac{(\ln 2)^2}{2!}n^2 + \frac{(\ln 2)^3}{3!}n^3 \quad (4.8)$$

Although equations (4.7) and (4.8) do not contain strictly integer terms, the non-integer terms can be converted to integers by multiplying with a suitably large integer factor. For instance,  $\ln a$  can be first computed as a real number, then multiplied by  $2^{10} = 1024$ , and finally rounded to an integer. Thus,  $\ln 2$  can be converted to  $(\ln 2) \cdot 2^{10} = (0.69315) \cdot 1024 \cong 710$ . Intermediate arithmetic operations are calculated

using this new integer representation for  $\ln 2$ . After all intermediate operations are completed, the final result is obtained by dividing by  $2^{10}$ . This technique preserves accuracy during intermediate integer arithmetic operations. Similarly, other non-integer terms such as division by  $3!$  can be converted to integers in the same manner to obtain accurate results using purely integer arithmetic. For more details on this method, a typical integer implementation is given in the next section.

## 4.6 Avoiding Integer Overflow

Equations (4.3) and (4.4) were implemented on a 32-bit RISC integer microprocessor. To reduce latency and achieve real-time performance, no integers greater than 32 bits were used. This 32-bit restriction caused problems with integer overflow. The solution was to rearrange the order of intermediate arithmetic operations to avoid overflow while maintaining the necessary degree of accuracy.

One sample 32-bit integer implementation in C for magnification using equations (4.3) and (4.4) and the Taylor series approximation in equation (4.8) is as follows:

```
int32 r, xo, yo, xin, xout, yin, yout, rSq, k1, k2, xy, factor;

rSq = r * r;
k1 = 5767168 / r;
k2 = 2048 / r;
xy = (xin - xo) * (xin - xo) + (yin - yo) * (yin - yo);

/* Calculate the power term */
factor = 8388608 + (5767168 - (xy * k1) / r) +
        (2048 - (xy * k2) / r) * (1024 - ((xy * k2) >> 1) / r) +
        (128 - (xy * 128) / rSq) * (64 - (xy * 64) / rSq) *
        (64 - (xy * 64) / rSq);

/*
** For each output pixel (xout, yout), map to an input pixel
** (xin, yin)
*/
xout = xo + ((factor * (xin - xo)) >> 23);
yout = yo + ((factor * (yin - yo)) >> 23);
```

In this implementation,  $r$  is the radius of the region to be magnified and  $(x_o, y_o)$  are the center coordinates. A similar implementation can be made in C++, Java, or any other programming language.

This method of dealing with integer overflow by rearranging the order of arithmetic operations can be applied to all integer implementations. More specifically, the method can be used for any combination of parameter values for  $a$ ,  $k$ , and  $m$  in equations (4.1) and (4.2) and for any microprocessor, including 64-bit processors, 128-bit processors, and so forth.

## 4.7 Implementation for Subsampled YCbCr and YCrCb

The current implementation supports RGB, YCbCr, and YCrCb images, including subsampled color formats such as YCbCr 4:2:2, YCbCr 4:2:0, and YCrCb 4:2:0. RGB images can be processed by directly applying equations (4.1) and (4.2) and the Taylor series approximation in equation (4.7). However, subsampled YCbCr and YCrCb images cause problems due to their subsampled chrominance components, Cb and Cr.

For instance, one common form of subsampled YCbCr is YCbCr 4:2:2, in which the Cb and Cr components of either pixel rows or pixel columns are subsampled by a factor of two. Two formats are H2V1 YCbCr 4:2:2, in which pixel columns are subsampled but pixel rows are unaffected, and H1V2 YCbCr 4:2:2, in which pixel rows are subsampled but pixel columns are unaffected. Fig. 4.15 shows the H2V1 format. For every block of 2 x 2 pixels, there are 4 Y values, 2 Cb values, and 2 Cr values, hence the name 4:2:2. If columns are numbered starting from zero, only even columns have the Cb component and only odd columns have Cr. Hereafter, YCbCr 4:2:2 refers to H2V1 YCbCr 4:2:2.

	0	1	2	3
0	YCb	YCr	YCb	YCr
1	YCb	YCr	YCb	YCr

Fig. 4.15. H2V1 YCbCr 4:2:2.

Directly applying equations (4.1) and (4.2) will fail to guarantee correctly alternating YCb-YCr pixels in the transformed image. To solve this problem, a temporary YCbCr 4:4:4 image is created from the original subsampled YCbCr 4:2:2 image. The transform equations are then applied to the YCbCr 4:4:4 image to produce a temporary YCbCr 4:4:4 output, which is subsampled to obtain the final YCbCr 4:2:2 output.

The temporary YCbCr 4:4:4 image contains all three YCbCr components for each pixel. It is created by taking pairs of adjacent YCb and YCr pixels and having each pixel borrow the missing chrominance component from its partner. The YCb pixel borrows a Cr component from its paired YCr pixel. The YCr pixel borrows a Cb component from the YCb pixel. Normally, this method does not produce an accurate, visually pleasing YCbCr 4:4:4 image from a YCbCr 4:2:2 image. However, since the temporary YCbCr 4:4:4 image is used only for intermediate processing, the accuracy of the intermediate result does not matter as long as the final output is accurate.

For the purpose of applying equations (4.1) and (4.2), this method of creating a YCbCr 4:4:4 image and performing transformations in the YCbCr 4:4:4 domain works very well. The final YCbCr 4:2:2 output is obtained from the temporary YCbCr 4:4:4 output by discarding the extra Cb or Cr component for each pixel. There is no visible difference between a YCbCr 4:2:2 transformed image produced in this manner and an

equivalent RGB transformed image. Fig. 4.16 (a) and (b) compare an RGB magnified image produced from RGB input with the corresponding YCbCr 4:2:2 magnified image produced from YCbCr 4:2:2 input. Both images were produced with parameters  $a = 2$ ,  $k = 1$ , and  $m = 0$ , using the same circular region of magnification. Similarly, Fig. 4.17 (a) and (b) compare RGB and YCbCr 4:2:2 pinched images created with parameters  $a = 2$ ,  $k = 1$ , and  $m = 1$ , using the same circular region of pinching.

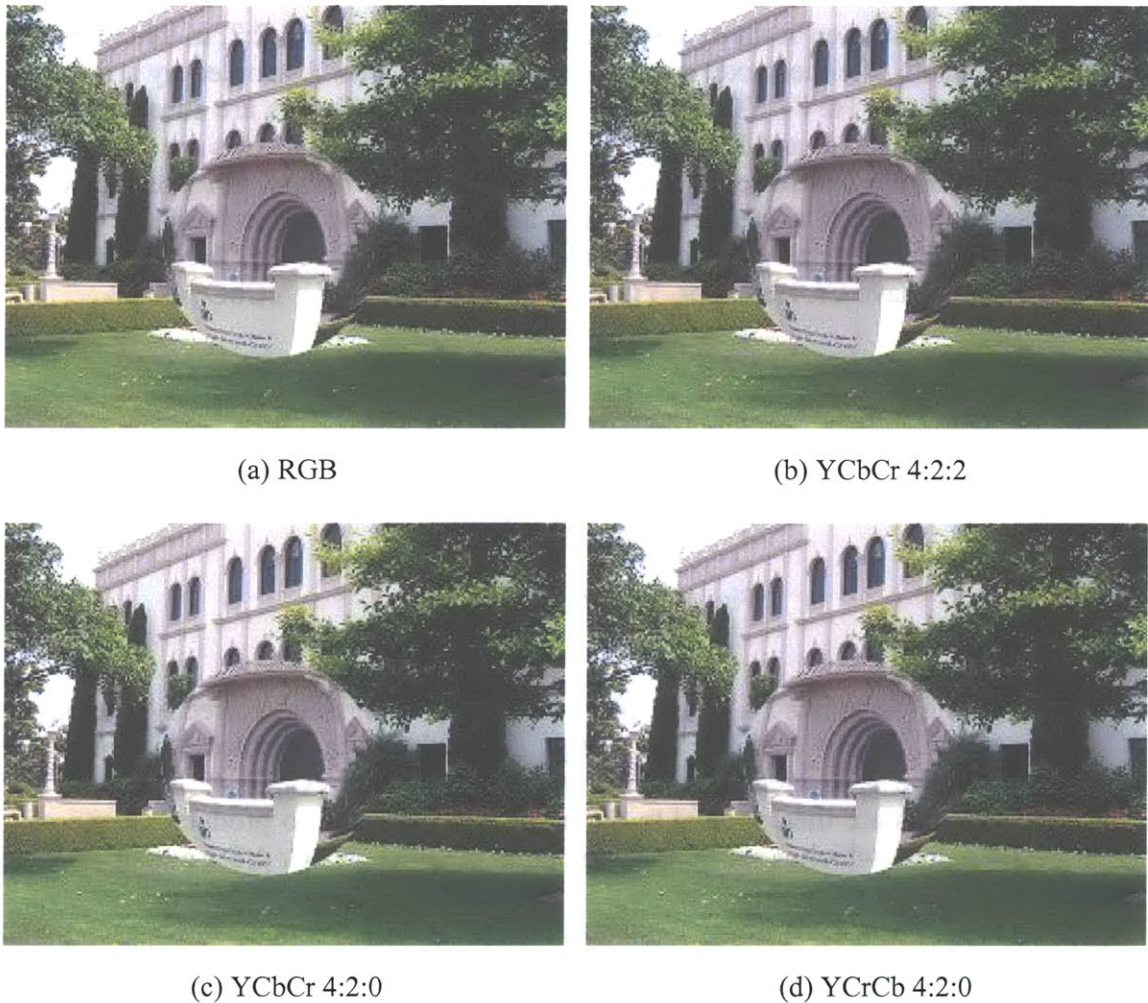


Fig. 4.16. Comparison of magnified images for 4 different color formats.

This method of creating a temporary YCbCr 4:4:4 image can be extended to handle input images in any subsampled YCbCr or YCrCb color format. For instance, another common format is YCbCr 4:2:0, in which each block of  $2 \times 2$  pixels has only one Cb and one Cr component. A temporary YCbCr 4:4:4 image is created by duplicating the Cb and Cr components for each pixel in each  $2 \times 2$  block. Transformation functions are applied to the temporary YCbCr 4:4:4 image to obtain a YCbCr 4:4:4 output, which is subsampled to YCbCr 4:2:0 by discarding the extra chrominance components. There is no visible

difference between YCbCr 4:2:0 transformed images produced in this manner and RGB transformed images produced from RGB input. This can be seen by comparing Fig. 4.16 (a) with (c) and (d). Fig. 4.16 (c) shows magnified results for YCbCr 4:2:0 and Fig. 4.16 (d) shows YCrCb 4:2:0, where the order of the Cb and Cr components has been switched. Both images were produced with parameters  $a = 2$ ,  $k = 1$ , and  $m = 0$ , using the same circular region of magnification. Similarly, Fig. 4.17 (c) and (d) show pinched results for YCbCr 4:2:0 and YCrCb 4:2:0 created with parameters  $a = 2$ ,  $k = 1$ , and  $m = 1$ , using the same circular region of pinching.



(a) RGB



(b) YCbCr 4:2:2



(c) YCbCr 4:2:0



(d) YCrCb 4:2:0

Fig. 4.17. Comparison of pinched images for 4 different color formats.





# Chapter 5

## Perspective Transformation (Patent Pending)

### 5.1 Introduction

Perspective is a mathematical transformation for creating the illusion of space and distance on a flat surface. It is used to represent three-dimensional scenes on a two-dimensional surface while simulating the natural perception of depth and distance by the human eye. Objects that are far away from the viewer or camera appear smaller in a perspective image and objects that are closer appear larger. Literary allusions indicate that ancient Greek painters and geometers knew of the laws of perspective. Isolated uses of perspective have been found on Greek vases from as early as the 6<sup>th</sup> century B.C. [9]. A systematic approach to perspective first appeared in the early 1400s in Florence, Italy, based on the writings of the architect Leon Battista Alberti [29].

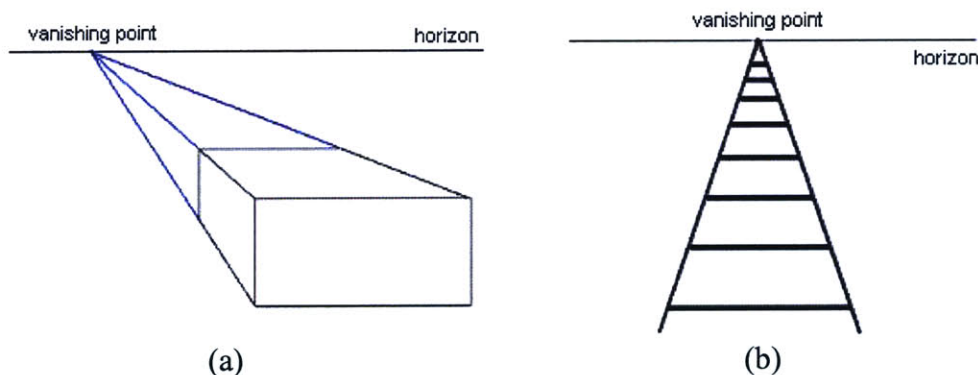


Fig. 5.1. Perspective drawings. (a) A drawing in perspective with a vanishing point and horizon line. (b) A railroad track drawn in perspective.

To use perspective, an artist must imagine the image surface as an open window through which to see the world. A straight line is drawn on the image to represent the horizon, as shown in Fig. 5.1 (a). Visual rays connect the viewer's eye to a point on the horizon called the vanishing point. For instance, if one looks along a long, straight railroad track, the parallel sides of the track appear to meet at a point in the distance. This point is the vanishing point in a perspective drawing, as shown in Fig. 5.1 (b). The farther away the vanishing point is from the viewer, the less slanted the visual rays will be, and the less distorted an image will appear when viewed in perspective.

Perspective has long been used in a variety of applications, including industrial and architectural design, computer animation in movies, games, and other entertainment applications, teaching, medicine, city planning, and many others. It is a visual effect widely used to add realism to two-dimensional displays.

## 5.2 Perspective Projections

Strictly speaking, a perspective projection should be a three-dimensional transformation that maps three-dimensional scenes onto a two-dimensional projection plane so that depth information is maintained. This can be done conceptually through a three-step process. First, a three-dimensional view volume is defined as the volume of space being viewed in the world. For ease of modeling, the view volume can be approximated by a pyramid whose apex is at the location of the viewer or camera. It is assumed that the camera is a perfect pinhole camera with known focal length. Next, objects in the three-dimensional world are clipped against the view volume and projected onto a two-dimensional projection plane called the window. Finally, the window projection is mapped onto a two-dimensional viewport and displayed on a device such as a computer screen. In practice, it is not necessary to follow these steps exactly as long as the final result agrees with the perspective model [24]. In some cases, it might even be necessary to use techniques that provide visually compelling results which may not be entirely physically correct [23].

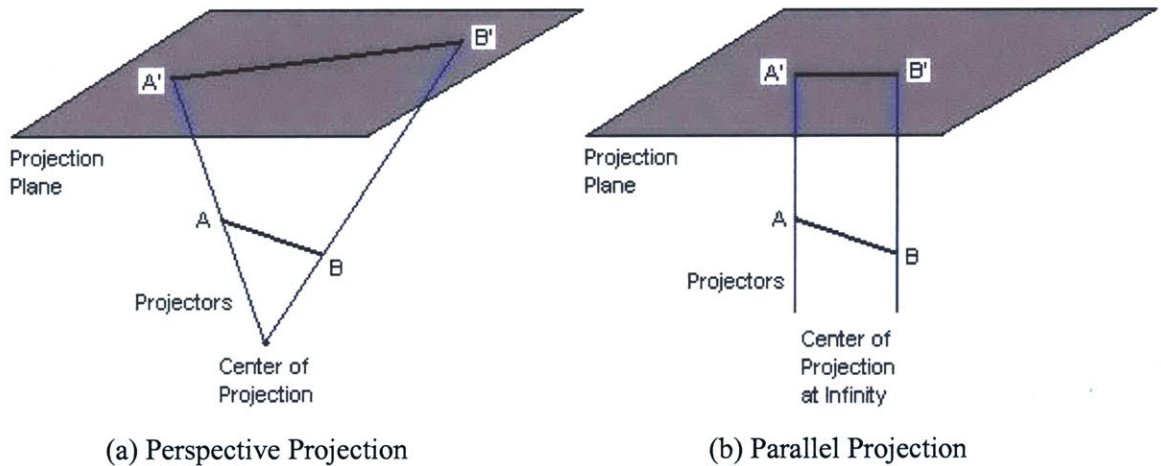


Fig. 5.2. The two types of planar geometric projections.

Perspective projections belong to a more general class of projections called planar geometric projections. Such projections are planar because they are formed on a plane, rather than a curved surface, and are geometric because they use straight, rather than

curved, projection lines called projectors. The projected two-dimensional image is found by passing a projector through each point of the three-dimensional object and finding the projector's intersection with the projection plane. All projectors extend from a single point called the center of projection. Besides planar geometric projections, nonplanar and nongeometric projections are also used in practice, most notably for cartography and for displaying the Omnimax film format [29].

Perspective projections represent only one of two types of planar geometric projections. The second type consists of parallel projections. These two projection types differ in the distance between the projection plane and the center of projection. For a perspective projection, the distance is finite. For a parallel projection, it is infinite. Both perspective and parallel projections are illustrated in Fig. 5.2, which shows the projections of a line in three-dimensional space. A parallel projection can be interpreted as the limiting case of a perspective projection in which the center of projection becomes a point at infinity and the projectors, which normally converge at the center of projection, become parallel and nonconvergent.

A perspective projection produces a visual effect called perspective foreshortening, which mimics the perception of depth and distance by the human visual system and photographic camera systems. In perspective foreshortening, the size of the perspective projection of an object is inversely proportional to the distance between that object and the center of projection, which represents the location of the viewer. The farther away an object is from the viewer, the smaller it will appear in a perspective projection. Although foreshortening allows a perspective projection to look realistic, the amount of foreshortening is nonuniform, and both distances and angles of objects are distorted. The nonuniformity renders the resulting perspective image unusable for measuring exact distances and angles. Only faces of an object that are parallel to the projection plane have angles that are preserved by a perspective projection. Similarly, only lines parallel to the projection plane remain parallel in a perspective image.

A set of parallel lines that are no longer parallel in a perspective image must converge at the vanishing point. This is the point of intersection between the projection plane and a line that passes through the center of projection and is parallel to the set of parallel lines in the original image. Since parallel lines in the real three-dimensional world meet only at infinity, a vanishing point can be interpreted as the perspective projection of a point at infinity. Vanishing points for lines parallel to a plane always lie along a straight line in the projection plane. When this line is horizontal to the viewer, it is the horizon line. The vanishing point and horizon line were both illustrated in Fig. 5.1.

If the set of parallel lines is parallel to one of three principal axes,  $x$ ,  $y$ , or  $z$ , in the original image, the vanishing point at which they converge in the perspective image is called an axis vanishing point. The number of axis vanishing points is equal to the number of principal axes intersected by the projection plane. No more than three such points can exist. For instance, if the projection plane intersects only the  $x$  axis and is therefore normal to it, only the  $x$  axis has a principal vanishing point. In this case, lines

parallel to the y or z axis are also parallel to the projection plane and do not converge at a vanishing point. This case is called one-point perspective, which is shown in Fig. 5.3.

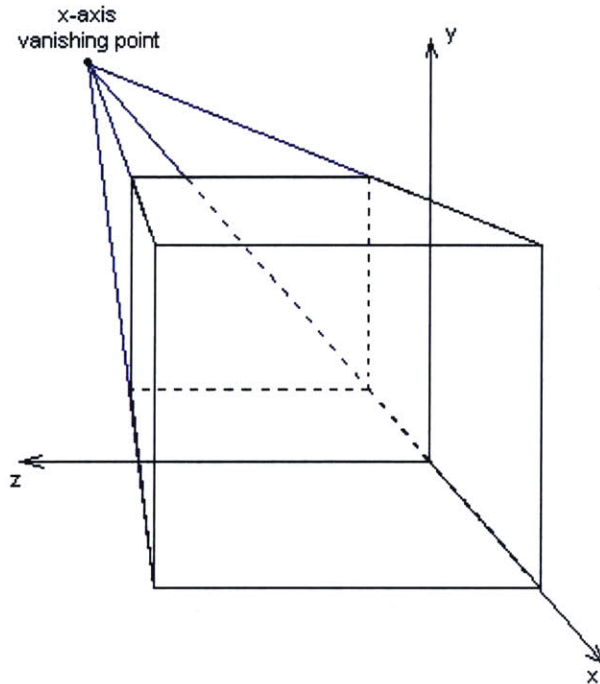


Fig. 5.3. One-point perspective with an x-axis vanishing point.

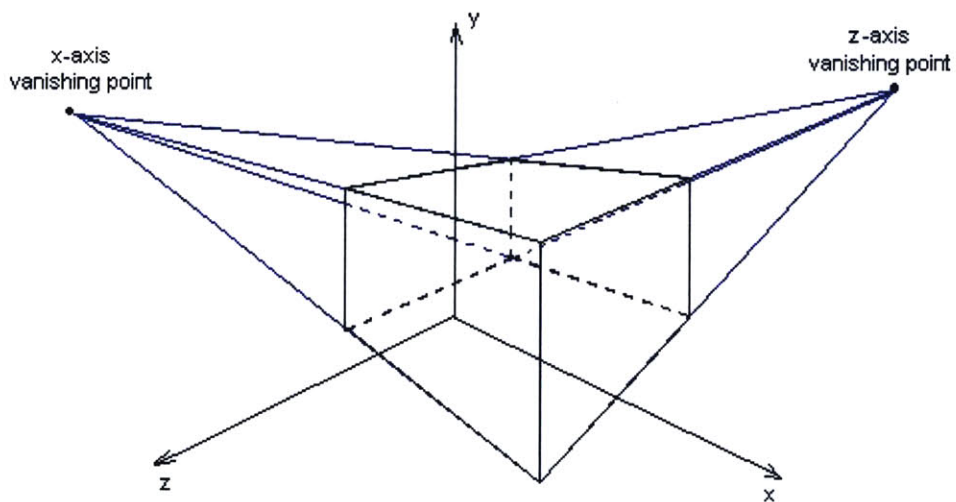


Fig. 5.4. Two-point perspective with x- and z-axis vanishing points.

Perspective projections are classified by the number of principal vanishing points they contain. Fig. 5.4 shows an example of two-point perspective in which the x and z axes

have principal vanishing points, but the y axis does not. Lines parallel to the y axis do not converge at a single point. Two-point perspective is often used in architectural, engineering, and industrial design drawings. Three-point perspective also exists. However, it is not used frequently since it does not add much realism beyond that offered by two-point perspective [15]. Our main focus here will be on one-point perspective.

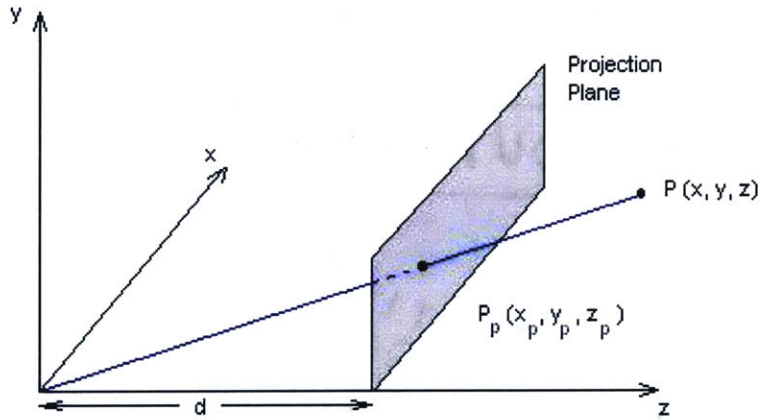


Fig. 5.5. One-point perspective projection [15].

Suppose that in a one-point perspective projection, the projection plane is normal to the z axis at  $z = d$  and that the center of projection, where the viewer is located, is at the origin, as illustrated in Fig. 5.5. Let  $P = (x, y, z)$  be a point in three-dimensional space to be projected onto the projection plane at  $z = d$ . To find  $P_p = (x_p, y_p, z_p)$ , the projection of  $P$ , similar triangles are used to obtain the equations

$$\frac{x_p}{d} = \frac{x}{z} \quad \text{and} \quad \frac{y_p}{d} = \frac{y}{z},$$

which yield

$$x_p = \frac{x}{z/d} \quad \text{and} \quad y_p = \frac{y}{z/d}. \quad (5.1)$$

Dividing by  $z$  produces the effect of perspective foreshortening by making objects appear smaller when they are farther away from the center of projection. The value of the  $z$ -coordinate can be any nonzero number. Points behind the center of projection have negative  $z$ -coordinates and points in front have positive  $z$ -coordinates.

The equations in (5.1) can be summarized by a  $4 \times 4$  transformation matrix in homogeneous coordinates:

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$$

Point  $P = (x, y, z)$  in homogeneous coordinates is  $P' = (x, y, z, 1)$ . Multiplying  $P'$  by matrix  $M$  produces the perspective projection point  $P'_p = (x'_p, y'_p, z'_p, w'_p)$ .

$$P'_p = \begin{bmatrix} x'_p \\ y'_p \\ z'_p \\ w'_p \end{bmatrix} = M \cdot P' = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix}$$

Dividing all four coordinates in  $P'_p$  by  $w'_p = z/d$  yields

$$\left( \frac{x'_p}{w'_p}, \frac{y'_p}{w'_p}, \frac{z'_p}{w'_p}, \frac{w'_p}{w'_p} \right) = \left( \frac{x}{z/d}, \frac{y}{z/d}, \frac{z}{z/d}, 1 \right) = \left( \frac{x}{z/d}, \frac{y}{z/d}, d, 1 \right).$$

Converting back to three-dimensional coordinates from homogeneous coordinates, we get

$$(x_p, y_p, z_p) = \left( \frac{x}{z/d}, \frac{y}{z/d}, d \right)$$

which confirms the equations in (5.1).

### 5.3 Two-Dimensional Perspective

The principles of three-dimensional perspective mapping can be adapted to transform a two-dimensional photographic image into an appropriate two-dimensional perspective image. Suppose that the farthest distance from the viewer that is captured in the photograph is  $z_{\max}$  and that the closest distance is  $z_{\min}$ . Let  $f$  be the focal length of the camera. For a point  $P = (x, y)$  in the original photograph, the corresponding point  $P_p = (x_p, y_p)$  in the perspective image is given by

$$x_p = \frac{x}{z/f}, \tag{5.2}$$

$$y_p = \frac{y}{z/f}, \tag{5.3}$$

where  $z$  varies between  $z_{\min}$  and  $z_{\max}$ , depending on the original distance from the viewer to point  $P$ . A two-dimensional perspective mapping should produce a result similar to Fig. 5.6. The coordinate system used in equations (5.2) and (5.3) has an origin located at the center of the bottom edge of the image, an  $x$  axis pointing to the right, and a  $y$  axis pointing upward. Points farther away from the camera have a larger  $y$ -coordinate and points nearby have a smaller  $y$ -coordinate.

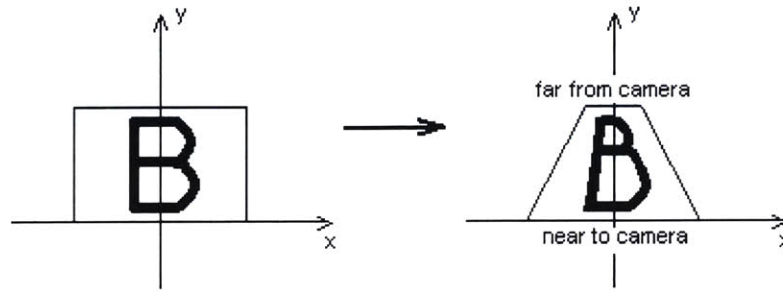


Fig. 5.6. A two-dimensional perspective mapping.



(a) Original Image

(b) Perspective Image from Photoshop

Fig. 5.7. Perspective result from Photoshop 7.0.

The Perspective function in Adobe Photoshop 7.0 produces results shown in Fig. 5.7. The original image is 520 x 390 pixels. The perspective image has a top width that is 260 pixels with a starting  $x$ -coordinate of -130 using the coordinate system defined in Fig. 5.6. Objects in the original image that are close to the camera are expected to be seen with more detail and are therefore stretched vertically in the perspective image. This is why pixels at the bottom of the image are interpolated to produce an approximate 1-to-2 mapping. Hence, the lawn at the bottom of the image is stretched out in the perspective image. Likewise, objects that are far from the camera should be seen with less detail and

are compressed in the vertical direction. Pixels at the top of the image are downsampled to produce an approximate 2-to-1 mapping. As seen in Fig. 5.7, the tops of the trees and buildings are appropriately compressed in the perspective image. This is the same technique used in computer gaming to give a realistic view of three-dimensional game worlds [23].

The amount of expansion and compression in a perspective image depends on the location of the vanishing point. Fig. 5.8 (a) shows that a vanishing point infinitely far from the camera produces a perspective image identical to the original image. A vanishing point close to the camera produces a perspective image with very slanted edges. In Fig. 5.8 (b), an arrow that was originally vertical becomes slanted in the perspective image. This phenomenon explains why the vertical tree trunks in Fig. 5.7 are slanted in the perspective image.

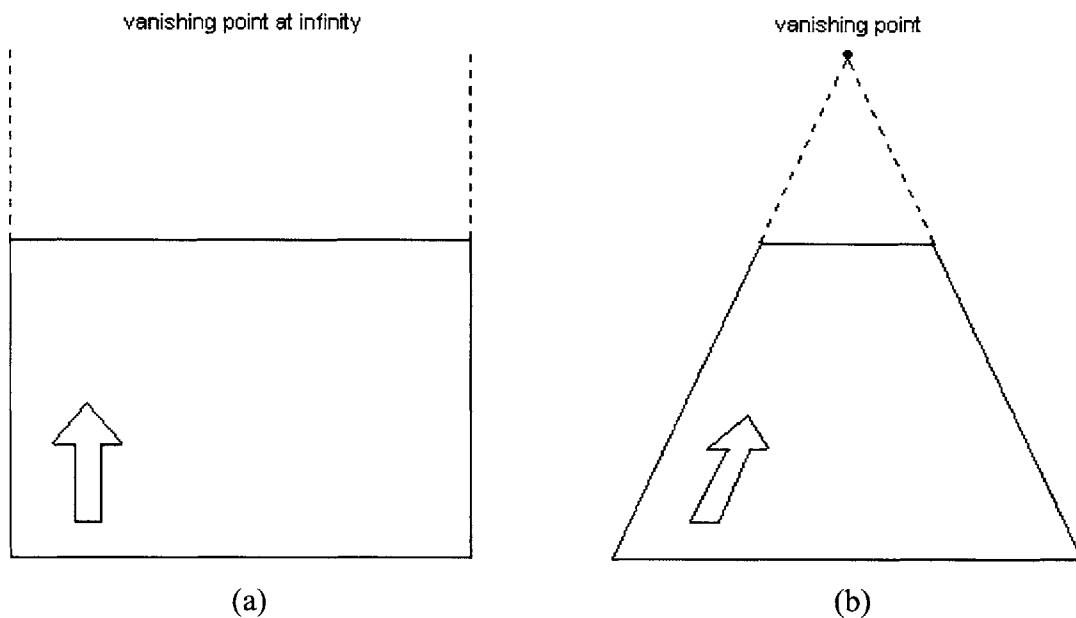


Fig. 5.8. Perspective images for different vanishing point locations: (a) A vanishing point at infinity. This can also be viewed as a parallel projection. (b) A vanishing point near the camera.

## 5.4 A New Algorithm

Creating a perspective image from an ordinary two-dimensional image requires finding a mapping equation between input and output pixel coordinates. To do so, let us define a coordinate system with the origin located at the lower left corner of the input image, the x axis pointing to the right, and the y axis pointing upward, as shown in Fig. 5.9.



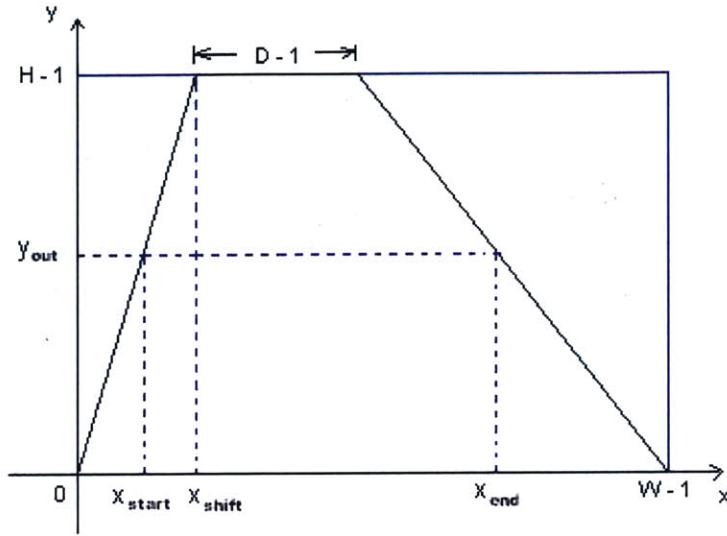


Fig. 5.9. Coordinate system used in the new perspective algorithm.

A one-point perspective image is generally shaped like a trapezoid. Suppose the desired perspective image has height  $H$ , top width  $D$ , and bottom width  $W$ . The transformation equations mapping input pixels in the original image to output pixels in the perspective image are

$$x_{out} = \frac{W-1}{\Delta x} \cdot x_{in} + x_{start}, \quad (5.4)$$

$$y_{out} = y_{in} \cdot 2 \left( \frac{H-1-y_{in}}{H-1} \right)^k, \quad (5.5)$$

where

$$x_{start} = \frac{x_{shift} \cdot y_{out}}{H-1},$$

$$x_{end} = W-1 - \frac{(W-D-x_{shift}) \cdot y_{out}}{H-1},$$

$$\Delta x = x_{end} - x_{start},$$

$$k = a \cdot \left( \frac{D}{W} \right) + b \cdot \left( \frac{D^2}{W^2} \right) + c \cdot \left( \frac{H}{y_{in} + \frac{H}{2}} \right) + d, \quad (5.6)$$

$(x_{in}, y_{in})$  is the input pixel location,  $(x_{out}, y_{out})$  is the output pixel location, and  $x_{shift}$  is the x-coordinate of the left endpoint of the top edge of the trapezoid. The top edge is defined as the smaller of the two parallel edges of the trapezoid. The width of the top edge is called the top width. The variables in these equations are labeled in Fig. 5.9. In equation (5.6), constants  $a$ ,  $b$ ,  $c$ , and  $d$  are empirically determined. A single set of values for these constants can be used for all images, regardless of size.

The key equations that determine the amount of expansion and compression in the perspective image are equations (5.5) and (5.6). Parameter  $k$  in the exponent of equation (5.5) controls the amount of expansion and compression. As given in equation (5.6),  $k$  itself depends on the distance between the camera and the vanishing point, which also determines the ratio  $D/W$ . Such dependency is appropriate since the amount of distortion in a perspective image should vary according to the location of the vanishing point relative to the viewer. Equation (5.6) gives only one possible specification for  $k$ . The equation can be altered to change the way  $k$  varies as a function of  $H$ ,  $D$ , and  $W$ . In particular, it is possible to change the equation so that results from this new perspective algorithm are identical to perspective results from Photoshop.

When the vanishing point is at infinity,  $k = 0$  and equation (5.5) reduces to  $y_{out} = y_{in}$  for every pixel. This produces a perspective image that is identical to the original image, as expected.



(a) Original Image

(b) Perspective Image from New Algorithm

Fig. 5.10. Perspective result using the new algorithm in equations (5.4) and (5.5) with  $k = 1$ ,  $W = 520$ ,  $H = 390$ ,  $D = 260$ , and  $x_{shift} = 130$ .

When  $k = 1$ , the result is the image shown in Fig. 5.10. The original input image is  $520 \times 390$  pixels. The top edge of the perspective image has a width of 260 pixels and a starting x-coordinate of 130. Here, equation (5.5) reduces to

$$y_{out} = y_{in} \cdot 2^{\left(\frac{H-1-y_{in}}{H-1}\right)}. \quad (5.7)$$

To understand equation (5.7), let us consider what happens to the bottom of the original image when it is transformed to perspective view. Conceptually, the bottom region should be expanded since it is close to the camera. Fig. 5.10 shows that an approximate 1-to-2 mapping is needed in this region. That is, each input pixel should be mapped to two output pixels. Setting  $y_{in} = 1$  in equation (5.7) and truncating the result to an integer produces

$$y_{out} = y_{in} \cdot 2^{\left(\frac{H-1-y_{in}}{H-1}\right)} = 1 \cdot 2^{\left(\frac{390-1-1}{390-1}\right)} \cong 1.$$

Similarly, for  $y_{in} = 2$ ,

$$y_{out} = y_{in} \cdot 2^{\left(\frac{H-1-y_{in}}{H-1}\right)} = 2 \cdot 2^{\left(\frac{390-1-2}{390-1}\right)} \cong 3.$$

The gap between the  $y_{out}$  values for  $y_{in} = 1$  and  $y_{in} = 2$  can be filled by pixel interpolation or duplication. Notice that the desired 1 to 2 mapping is achieved.

Now let us consider the top of the image. Substituting  $y_{in} = H - 3 = 387$  produces

$$y_{out} = y_{in} \cdot 2^{\left(\frac{H-1-y_{in}}{H-1}\right)} = 387 \cdot 2^{\left(\frac{390-1-387}{390-1}\right)} \cong 388.$$

Similarly, for  $y_{in} = H - 2 = 388$ ,

$$y_{out} = y_{in} \cdot 2^{\left(\frac{H-1-y_{in}}{H-1}\right)} = 388 \cdot 2^{\left(\frac{390-1-388}{390-1}\right)} \cong 388.$$

Both  $y_{in}$  values are mapped to the same  $y_{out}$  value, thus producing the approximate 2-to-1 mapping seen at the top of Fig. 5.10. This is conceptually correct since the top of the image is farther away from the camera and should be compressed. These results confirm that equation (5.5) produces an appropriate perspective image.

When  $k > 1$ , the vanishing point is closer to the camera and more extreme distortion will appear in the perspective image. For instance, when  $k = 1.5$ , TABLE 5.1 shows the y-coordinate mappings that would result for a 520 x 390 input image. For small y-coordinates, which represent points close to the camera, there is a high degree of expansion. For large y-coordinates representing points far from the camera, there is a high degree of compression. TABLE 5.1 shows that there is an approximate 1-to-3 mapping for  $1 \leq y_{in} \leq 3$  and an approximate 5-to-1 mapping for  $385 \leq y_{in} \leq 389$ . This is the desired result.

TABLE 5.1: Y-Coordinate Mappings for  $k = 1.5$  in equation (5.5).

$y_{in}$	$y_{out}$
1	2
2	5
3	8
4	11
5	13

$y_{in}$	$y_{out}$
385	389
386	389
387	389
388	389
389	389

## 5.5 Integer Implementation

A problem arises if the transformation equations for perspective must be implemented on an integer microprocessor where floating-point arithmetic produces unacceptable latency. Under such circumstances, a method must be found to calculate the power of 2 in equation (5.5) using strictly integer arithmetic. The solution is to use a Taylor series expansion to approximate the power function:

$$2^n = 1 + (\ln 2)n + \frac{(\ln 2)^2}{2!}n^2 + \frac{(\ln 2)^3}{3!}n^3 + \dots + \frac{(\ln 2)^k}{k!}n^k + \dots \quad (5.8)$$

The first four terms in the Taylor series are sufficient to accurately calculate the power of 2. Higher order terms can still be used if desired. However, these terms will have little effect on accuracy and will only decrease the speed of the implementation. Equation (5.5) was implemented with the series approximation

$$2^n \cong 1 + (\ln 2)n + \frac{(\ln 2)^2}{2!}n^2 + \frac{(\ln 2)^3}{3!}n^3 \quad (5.9)$$

Although equations (5.8) and (5.9) do not contain strictly integer arithmetic, the non-integer terms can be converted to integers by multiplying with a suitably large integer factor. For instance,  $\ln a$  can be first computed as a real number, then multiplied by  $2^{10} =$

1024, and finally rounded to an integer. Thus,  $\ln 2$  can be converted to  $(\ln 2) \cdot 2^{10} = (0.69315) \cdot 1024 \cong 710$ . Intermediate arithmetic operations are calculated using the new integer representation for  $\ln 2$ . After all intermediate operations are completed, the final result is obtained by dividing by  $2^{10}$ . This technique preserves accuracy during intermediate integer arithmetic operations. Similarly, other non-integer terms such as division by  $3!$  can be converted to integers in the same manner to obtain accurate results using purely integer arithmetic. For more details on this method, a typical integer implementation is given in the next section.

## 5.6 Avoiding Integer Overflow

The perspective transformation in equations (5.4) and (5.5) was implemented on a 32-bit RISC integer microprocessor. To reduce latency and achieve real-time performance, no integers greater than 32 bits were used. This 32-bit restriction caused problems with integer overflow. The solution was to rearrange the order of intermediate arithmetic operations to avoid overflow while maintaining the necessary degree of accuracy.

One sample 32-bit integer implementation in C for equations (5.4) and (5.5) using the approximation in (5.9) is the following:

```
int32 yin, yout, c1, c2, c3, a, k, k2;

a = h - 1;
k2 = ((1054 * (d << 8)) / w + 58777298) -
      (((1922 * (d << 8)) / w) << 8) / w * d);
k = (k2 + (int32)(1038 * ((a << 16) / ((yin << 1) + a)))) >> 16;

c1 = ((k * (a - yin)) >> 1) / a;
c2 = (((c1 * (a - yin)) * k) / a) >> 10;
c3 = (((c2 * 7 * (a - yin)) >> 6) * k) / a >> 4;

/*
** For each input y-coordinate yin, map to the corresponding
** output y-coordinate yout
*/
yout = (yin * (65536 + 89*c1 + 31*c2 + c3)) >> 16;
```

In this implementation,  $h$  is the height of the trapezoidal perspective image,  $w$  is the bottom width, and  $d$  is the top width. A similar implementation can be made in C++, Java, or any other programming language.

This method of dealing with integer overflow by rearranging the order of arithmetic operations can be applied to all implementations and for all microprocessor, including 64-bit processors, 128-bit processors, and so forth.

## 5.7 Implementation for Subsampled YCbCr and YCrCb

The current implementation supports RGB, YCbCr, and YCrCb images, including subsampled color formats such as YCbCr 4:2:2, YCbCr 4:2:0, and YCrCb 4:2:0. RGB images can be processed by directly applying transformation equations (5.4) and (5.5) and the Taylor series approximation in equation (5.8). However, images in a subsampled YCbCr or YCrCb color format cause problems due to their subsampled chrominance components, Cb and Cr.

For instance, one common form of subsampled YCbCr is YCbCr 4:2:2, in which the Cb and Cr components of either pixel rows or pixel columns are subsampled by a factor of two. Two formats are H2V1 YCbCr 4:2:2, in which pixel columns are subsampled but pixel rows are unaffected, and H1V2 YCbCr 4:2:2, in which pixel rows are subsampled but pixel columns are unaffected. Fig. 5.11 shows the H2V1 format. For every block of 2 x 2 pixels, there are 4 Y values, 2 Cb values, and 2 Cr values, hence the name 4:2:2. If columns are numbered starting from zero, only even columns have the Cb component and only odd columns have Cr. Hereafter, YCbCr 4:2:2 refers to H2V1 YCbCr 4:2:2.

0	1	2	3
YCb	YCr	YCb	YCr
YCb	YCr	YCb	YCr

Fig. 5.11. H2V1 YCbCr 4:2:2.

Directly applying transformation equations (5.4) and (5.5) will fail to guarantee correctly alternating YCb-YCr pixels in the output image. To solve this problem, a temporary YCbCr 4:4:4 image is created from the subsampled YCbCr 4:2:2 input. The perspective mapping is performed on the YCbCr 4:4:4 image to obtain a temporary YCbCr 4:4:4 output, which is subsampled to produce the final YCbCr 4:2:2 output.

The temporary YCbCr 4:4:4 image contains all three YCbCr components for each pixel. It is created by taking pairs of adjacent YCb and YCr pixels and having each pixel borrow the missing chrominance component from its partner. The YCb pixel borrows a Cr component from its paired YCr pixel. The YCr pixel borrows a Cb component from the YCb pixel. Normally, this method does not produce an accurate, visually pleasing YCbCr 4:4:4 image from a YCbCr 4:2:2 image. However, since the temporary YCbCr 4:4:4 image is used only for intermediate processing, the accuracy of the intermediate result does not matter as long as the final output is accurate.

For the purpose of applying transformation equations (5.4) and (5.5), the method of creating a YCbCr 4:4:4 image and performing transformations in the YCbCr 4:4:4 domain works very well. The final YCbCr 4:2:2 output is obtained from the temporary YCbCr 4:4:4 output by discarding the extra Cb or Cr component for each pixel. There is

no visible difference between a YCbCr 4:2:2 perspective image produced in this manner and an equivalent RGB perspective image. Fig. 5.12 (a) and (b) compare an RGB perspective image produced from RGB input with the corresponding YCbCr 4:2:2 image produced from YCbCr 4:2:2 input. For both images,  $k=1$ ,  $W=520$ ,  $H=390$ ,  $D=130$ , and  $x_{shift}=196$ .

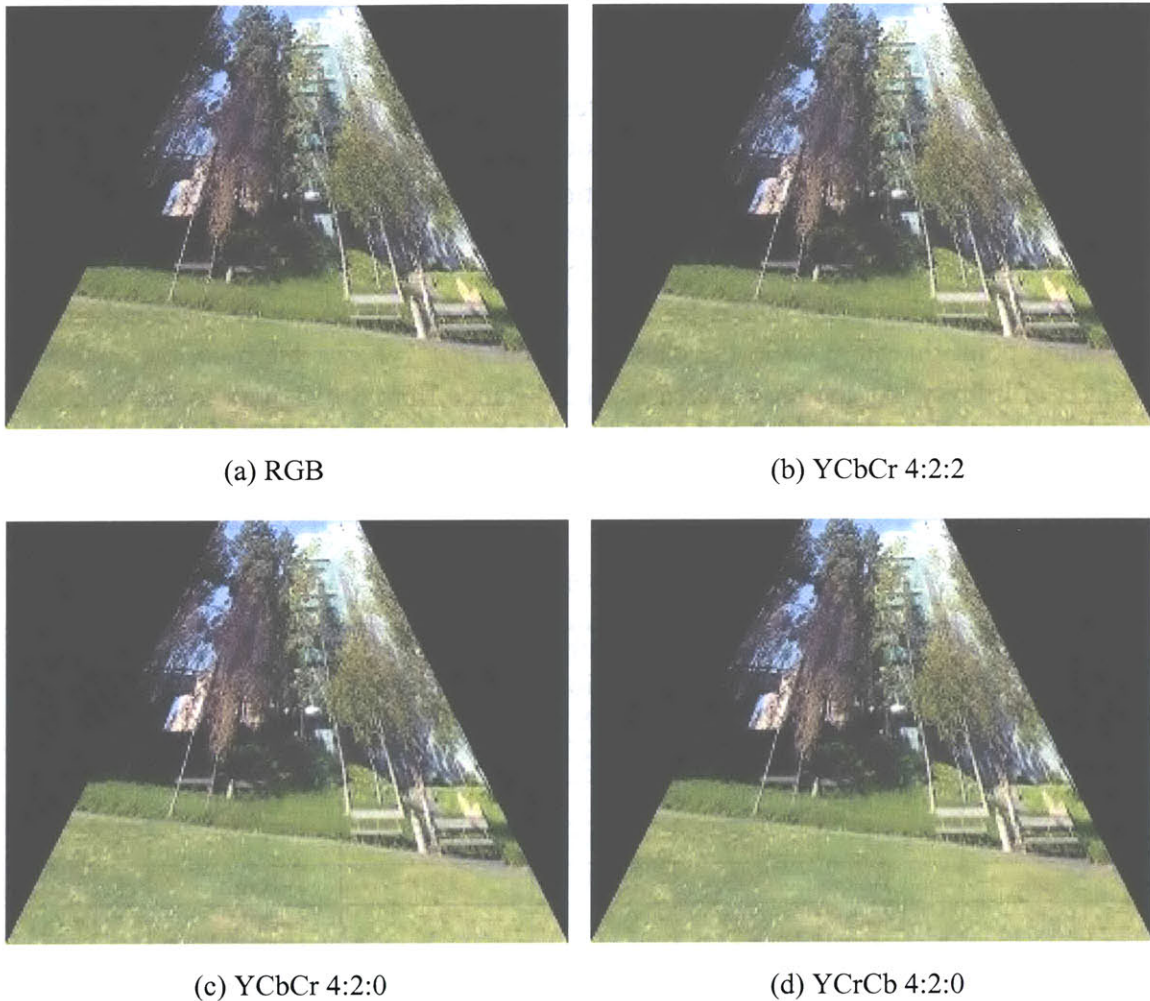


Fig. 5.12. Comparison of perspective images for 4 different color formats. For all images,  $k=1$ ,  $W=520$ ,  $H=390$ ,  $D=130$ , and  $x_{shift}=196$ .

The method of creating a temporary YCbCr 4:4:4 image can be extended to handle input images in any subsampled YCbCr or YCrCb color format. For instance, another common format is YCbCr 4:2:0, in which each block of  $2 \times 2$  pixels has only one Cb and one Cr component. A temporary YCbCr 4:4:4 image is created by duplicating the Cb and Cr components for each pixel in each  $2 \times 2$  block. Transformation equations are applied to the temporary YCbCr 4:4:4 image to obtain a YCbCr 4:4:4 output, which is subsampled to YCbCr 4:2:0 by discarding the extra chrominance components. There is no visible

difference between YCbCr 4:2:0 perspective images produced in this manner and RGB perspective images produced from RGB input. This can be seen by comparing Fig. 5.12 (a) with (c) and (d). Fig. 5.12 (c) shows perspective results for YCbCr 4:2:0 and Fig. 5.12 (d) shows YCrCb 4:2:0, where the order of Cb and Cr components has been switched. For all images,  $k = 1$ ,  $W = 520$ ,  $H = 390$ ,  $D = 130$ , and  $x_{shift} = 196$ .

## 5.8 Results

TABLE 5.2 summarizes various perspective results for a 520 x 390 test image. Eight cases are shown for different combinations of width ratio  $D/W$ , starting coordinate  $x_{shift}$ , and trapezoid orientation. Regardless of orientation, the top width is always the width of the top edge, which is the smaller of the two parallel edges of the trapezoid. The different values of  $D/W$  correspond to different locations for the vanishing point. A value indicates a vanishing point that is far away from the camera. A small value indicates a vanishing point that is close to the camera. When  $D/W = 1$ , for instance, the vanishing point is at infinity. As the ratio increases, the distance from the camera to the vanishing point decreases. Figs. 5.13, 5.14, and 5.15 show the output images obtained.

TABLE 5.2: Perspective Results.

Vanishing Point Location	Top Width (D)	Bottom Width (W)	Width Ratio (D/W)	Starting Coordinate ( $x_{shift}$ )	Orientation	Figure
Near	130	520	0.25	195	Up	5.13 (a)
Near	130	520	0.25	195	Down	5.13 (b)
Near	130	390	0.333	130	Left	5.13 (c)
Near	130	390	0.333	130	Right	5.13 (d)
Far	450	520	0.865	35	Up	5.14 (a)
Far	338	390	0.867	26	Left	5.14 (b)
Near	65	520	0.125	0	Up	5.15 (a)
Near	65	390	0.167	0	Left	5.15 (b)

Perspective images produced from equations (5.4) and (5.5) do not exactly match the theoretical mathematical formulas for perspective projection. This is intentional because if images did match the theoretical formulas, there would be too much compression in the far regions of each image and too much expansion in the near regions. The distortion produced by a theoretical perspective transformation would be so severe that very little information would be left in an output image. The compressed parts would have very few details and the expanded parts would have very coarse details stretched out in a visually



unpleasant manner. In fact, this extreme distortion is the main reason why the computer gaming industry also avoids the use of a mathematically accurate perspective transformation when depicting three-dimensional game worlds on a two-dimensional computer screen [23].

The mathematically accurate version of perspective appropriately models what the human visual system is able to perceive when viewing a three-dimensional scene. However, when perspective is applied to two-dimensional digital images already recorded by cameras, there is not enough pixel information available to support the necessary expansions and compressions. Applying the theoretical perspective equations would result in too much information being compressed and too much detail being lost. Therefore in this thesis, equations (5.4) and (5.5) were used instead of the theoretical equations to produce aesthetically pleasing perspective images.

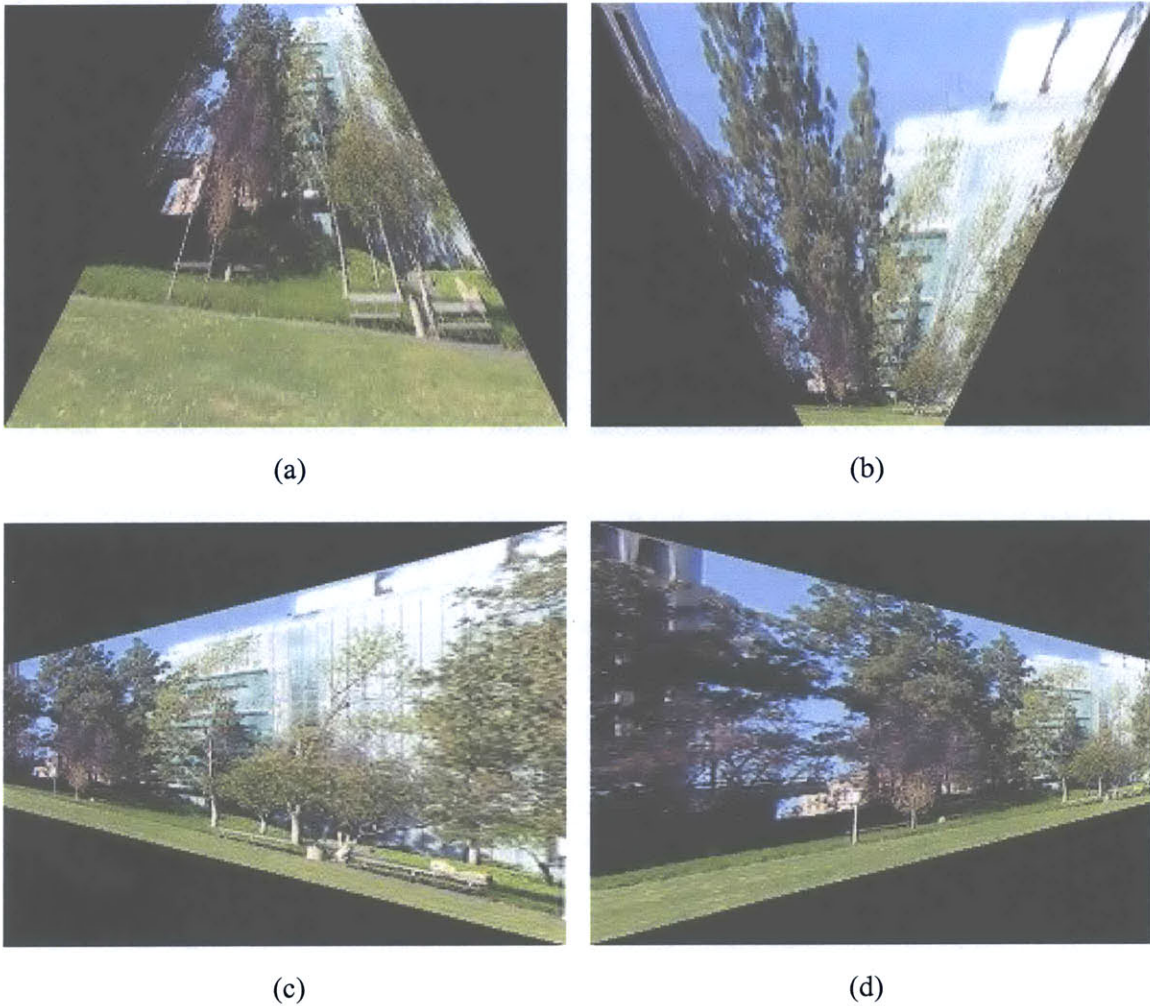


Fig. 5.13. Perspective images with a centered top edge and a vanishing point relatively close to the camera.



(a)



(b)

Fig. 5.14. Perspective images with a centered top edge and a vanishing point relatively far from the camera.



(a)



(b)

Fig. 5.15. Perspective images with a top edge that is off-center.

# Chapter 6

## Neon Effect

### 6.1 Introduction

Neon signs are often used in real life to outline advertisements, provide decoration, and light up storefronts. Fig. 6.1 shows a sample neon sign. In general, neon lighting can cast an otherworldly glow and add a subtle atmospheric effect to an otherwise boring night spot [19]. Through digital image processing, ordinary photographs can be transformed to neon images by accentuating edges with bright, glowing colors. Although such images are not true neon signs, the overall visual effect can be just as compelling.

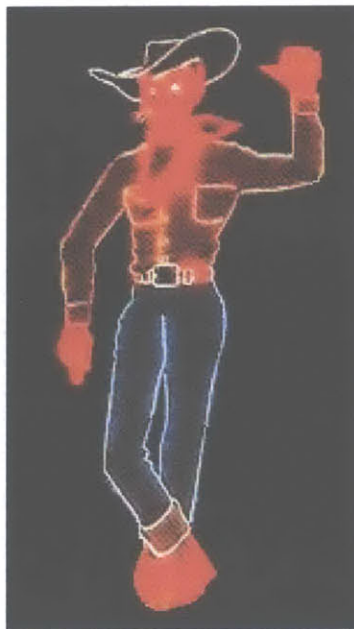


Fig. 6.1. Neon sign.

Neon Effect is a function that transforms photographs to neon images. Fig. 6.2 shows a sample result from this function. Based on the observation that edges must be found before they can be highlighted with neon colors, the algorithm for Neon Effect begins with edge detection.

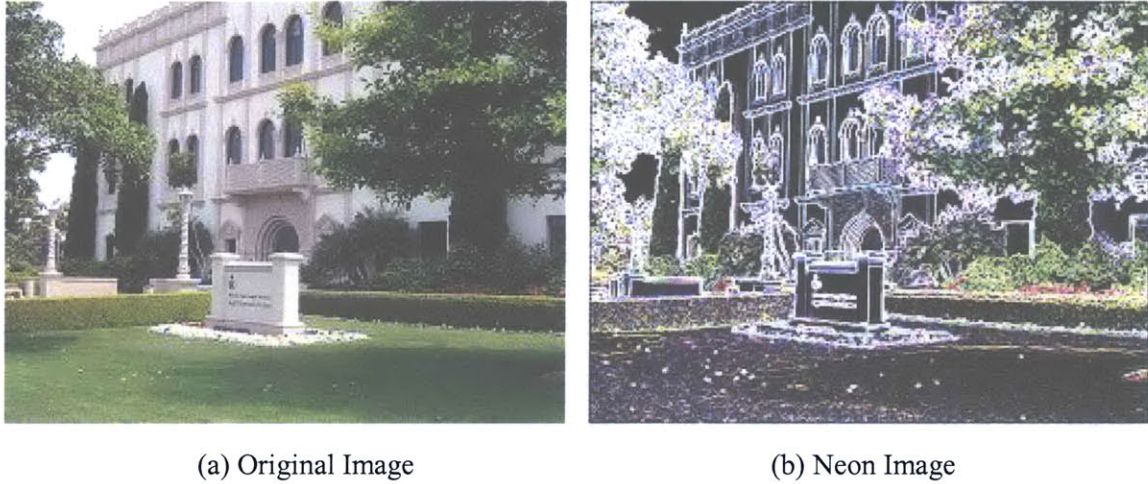


Fig. 6.2. Neon Effect.

## 6.2 Algorithm

To find the edges of an image, the Neon Effect algorithm uses a Prewitt gradient edge detector, one of the fastest, simplest edge detectors available. For each pixel, four directional gradients are calculated using the 3 x 3 filter masks shown in Fig. 6.3.

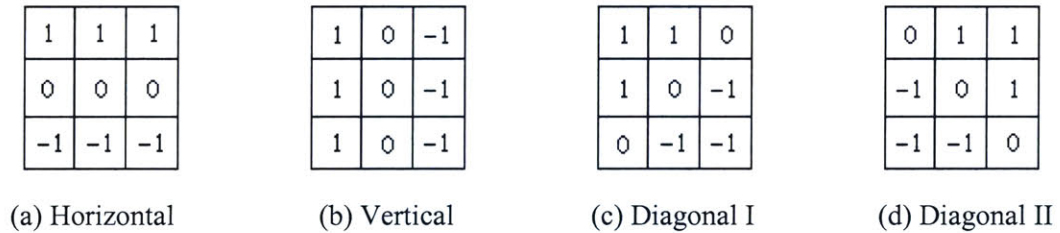


Fig. 6.3. Four 3 x 3 filter masks for calculating directional gradients.

If  $f(x, y)$  is the pixel intensity in the input image, the four gradients are

$$\nabla_1(x, y) = [f(x-1, y-1) - f(x-1, y+1)] + [f(x, y-1) - f(x, y+1)] + [f(x+1, y-1) - f(x+1, y+1)],$$

$$\nabla_2(x, y) = [f(x-1, y-1) - f(x+1, y-1)] + [f(x-1, y) - f(x+1, y)] + [f(x-1, y+1) - f(x+1, y+1)],$$

$$\nabla_3(x, y) = [f(x-1, y-1) - f(x+1, y+1)] + [f(x, y-1) - f(x+1, y)] + [f(x-1, y) - f(x, y+1)],$$

$$\nabla_4(x, y) = [f(x+1, y-1) - f(x-1, y+1)] + [f(x, y-1) - f(x-1, y)] + [f(x+1, y) - f(x, y+1)],$$

The maximum magnitude of the four gradients is found using

$$M(x, y) = \max(|\nabla_1(x, y)|, |\nabla_2(x, y)|, |\nabla_3(x, y)|, |\nabla_4(x, y)|) \quad (6.1)$$

To produce a neon image,  $M(x, y)$  is multiplied by a constant  $k$ , clipped to 255, and assigned as the output pixel intensity  $g(x, y)$ :

$$g(x, y) = \begin{cases} k \cdot M(x, y) & \text{for } M(x, y) \leq \frac{255}{k} \\ 255 & \text{otherwise} \end{cases} \quad (6.2)$$

Parameter  $k$  can be any value in the range  $k > 1$ . In this thesis, the best empirical results were obtained with  $k = 2$ .

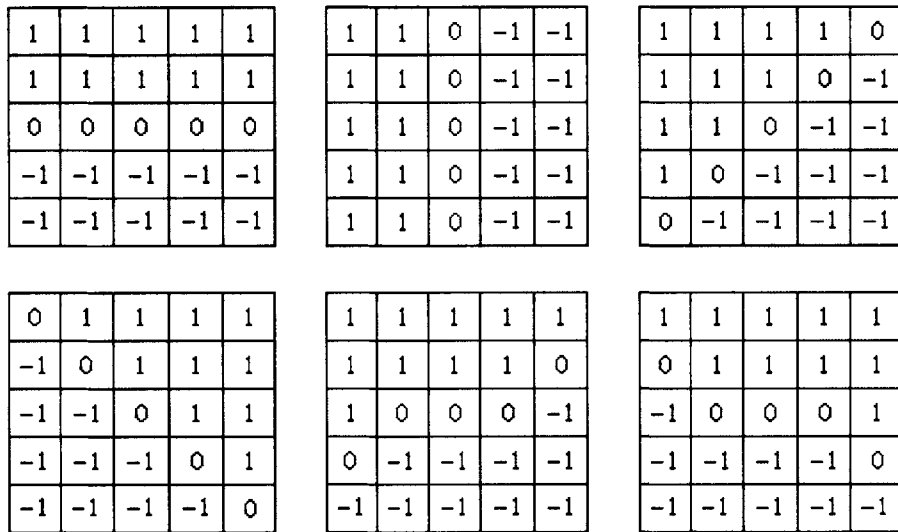


Fig. 6.4. 5 x 5 filter masks.

The Neon Effect algorithm can be extended in a variety of ways. First, any  $n \times n$  filter mask can be used, as long as  $n$  is an odd positive integer no less than 3. The basic algorithm given above uses a minimum filter size of 3 x 3 to optimize for speed. Second, instead of only four directional gradients, a larger number of gradients can be used. Larger values of  $n$  result in larger filter masks that allow more directional gradients to be calculated. Fig. 6.4 gives 5 x 5 filter masks for six different directional gradients. Third, any value of  $k$  greater than 1 can be used in equation (6.2). A larger  $k$  results in

brighter edges and more frequent clipping of pixel intensity values. Finally, a threshold  $\tau$  can be used for  $M(x, y)$  so that pixels with  $M(x, y) \leq \tau$  are colored black. That is, equation (6.2) becomes

$$g(x, y) = \begin{cases} 0 & \text{for } M(x, y) \leq \tau \\ k \cdot M(x, y) & \text{for } \tau < M(x, y) \leq \frac{255}{k} \\ 255 & \text{otherwise} \end{cases}$$

Adding an extra threshold weeds out weak edges from the neon image.

### 6.3 Results

Fig. 6.5 shows two sample neon images produced by the Neon Effect algorithm. Notice that edges are highlighted in neon colors, whereas relatively homogeneous regions are black.

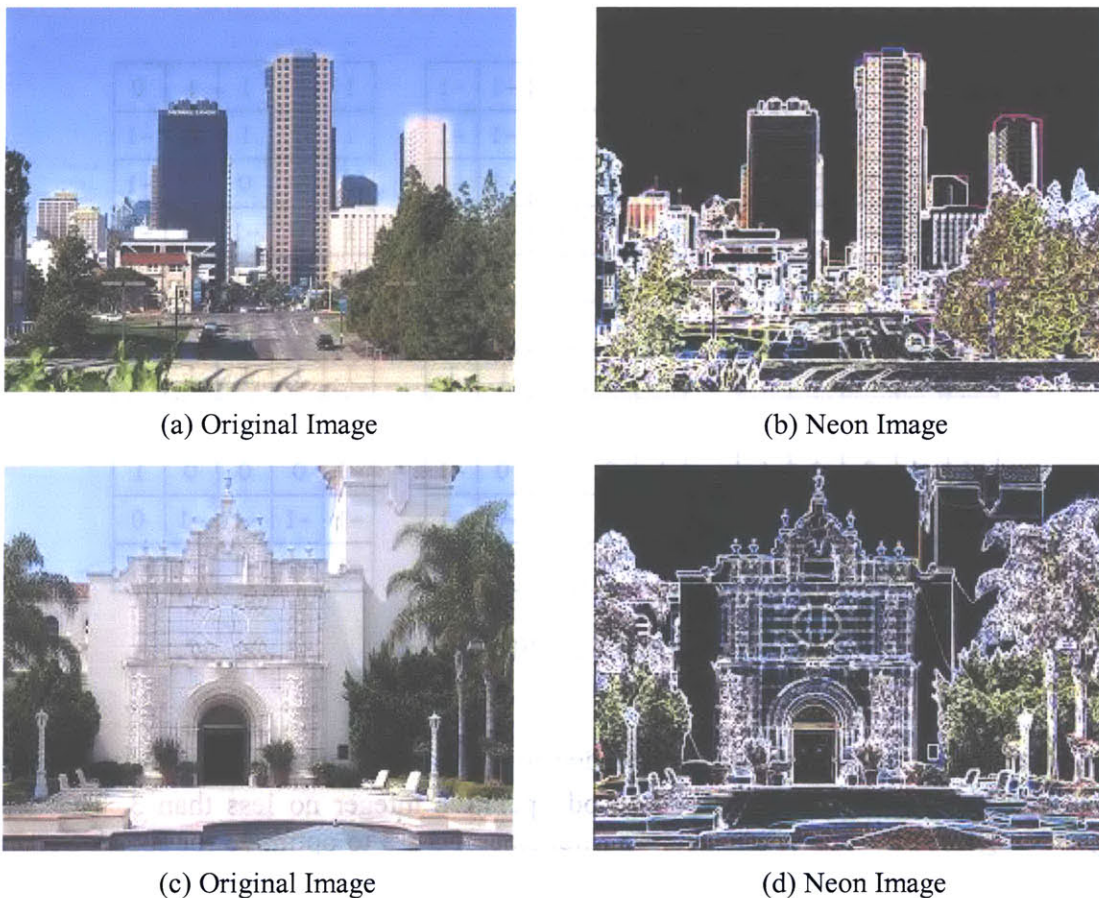


Fig. 6.5. Results from the Neon Effect algorithm.

## 6.4 Comparison with Photoshop

Adobe Photoshop 7.0 provides a function called Glowing Edges, which corresponds to the Neon Effect function in this thesis. Unlike the Neon Effect function, which automatically generates neon images, the Photoshop function relies on a user to manually adjust three different parameters to get desired results. Fig. 6.6 compares Neon Effect results with neon images from Photoshop, which were obtained with the following settings: Edge Width = 2, Edge Brightness = 10, and Smoothness = 2. These settings produced neon results most similar to those produced by the Neon Effect algorithm.



(a) Neon Effect Algorithm



(b) Photoshop 7.0



(c) Neon Effect Algorithm



(d) Photoshop 7.0

Fig. 6.6. Comparison with Photoshop 7.0.

## 6.5 Implementation for YCbCr and YCrCb

For an RGB image, the Neon Effect algorithm is simply repeated for each of the three color channels: red, green, and blue. However, a YCbCr or YCrCb image must be converted to the RGB domain to apply the algorithm. This is because the nonlinearity of the absolute value operation in equation (6.1) and the clipping operation in equation (6.2)

prevents the algorithm from being adapted to operate directly in the YCbCr domain, despite the fact that conversion between RGB and YCbCr is a linear operation. After the algorithm is applied in the RGB domain, the final output is obtained by converting back to YCbCr or YCrCb. Conversion between RGB and YCbCr is performed using equations (2.1) and (2.2).



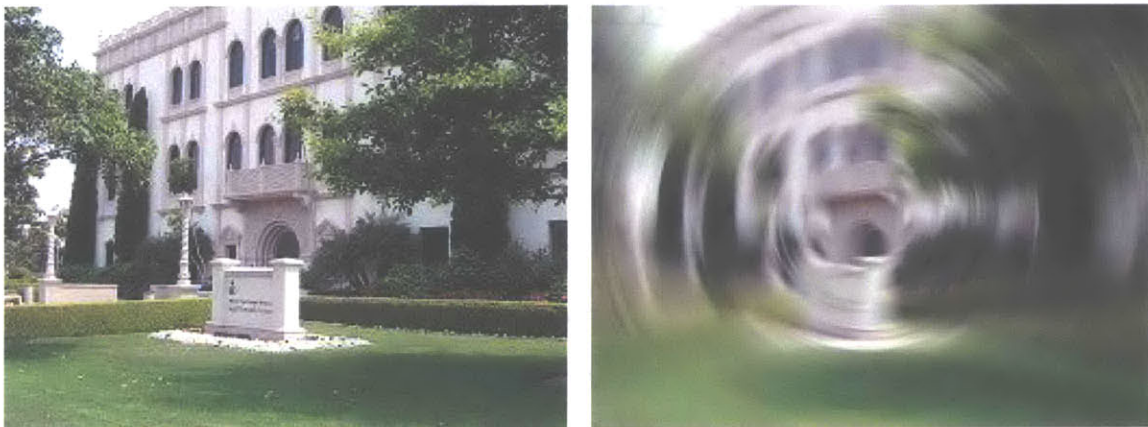
# Chapter 7

## Spin Radial Blur (Patent Pending)

### 7.1 Introduction

Radial blur simulates the blur of an image created by a rotating or zooming camera. There are two varieties of radial blur. The spin variety blurs along concentric circular lines. The zoom variety blurs along radial lines, as if zooming in or out of the image [1]. Hereafter, radial blur refers to spin radial blur unless otherwise noted.

Radial blur is one of a diverse array of special effects used by photographers to create artistic photographs that evoke emotion. Before the advent of digital image processing, only master photographers could create radial blur using special camera equipment. With modern desktop image processing applications such as Adobe Photoshop, ordinary users can create the same effects. Fig. 7.1 shows an example of radial blur from Photoshop 7.0 for a 520 x 390 image using best quality blur and a blur amount of 20. The center of rotation is the center of the image.



(a) Original Image

(b) Photoshop Radial Blur

Fig. 7.1. Radial blur from Photoshop 7.0.

On camera phones, radial blur is difficult to achieve under the often severe constraints on microprocessor speed, memory space, and power consumption. Even on a desktop computer with fast floating-point support, radial blur qualifies as one of Photoshop's

most time-consuming operations [30]. Traditional methods for radial blur perform blurring in either the Cartesian coordinate system, which requires computationally intensive convolutions, or the polar coordinate system, which requires convolution and conversion between Cartesian and polar coordinates.

If blurring is done in Cartesian coordinates, each input pixel must be averaged with neighboring pixels along a circular arc, as shown in Fig. 7.2, where  $(x, y)$  is the coordinate location of the pixel being blurred,  $(x_o, y_o)$  is the center of rotation,  $r$  is the distance between the pixel and the center of rotation, and  $n$  is the blur amount.

Suppose the original image is  $f(x, y)$  and the blurred image is  $g(x, y)$ . A Cartesian approach calculates the following average for each pixel:

$$g(x, y) = \frac{1}{M} \sum_{(x_i, y_j) \in W} f(x - x_i, y - y_j), \quad (7.1)$$

where  $W$  is a window centered on the input pixel and  $M$  is the size of  $W$ , which equals the total number of pixels being averaged. The value of  $M$  varies from pixel to pixel, depending on the blur amount  $n$  and the distance  $r$  between the pixel and the center of rotation.

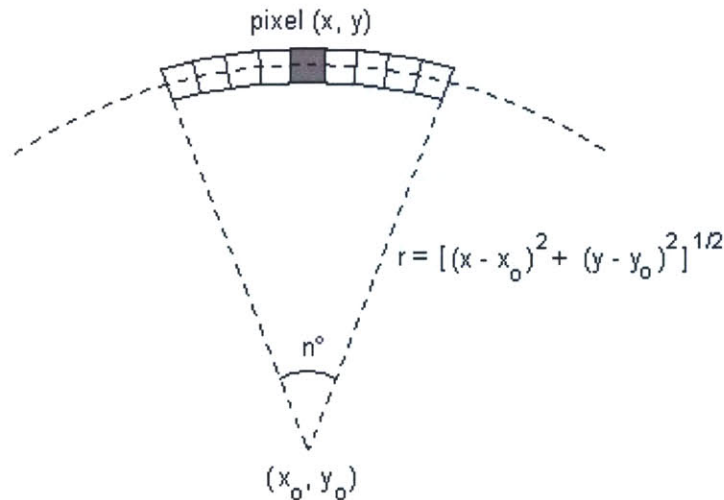


Fig. 7.2. Radial blur in Cartesian coordinates.

Calculating the average in equation (7.1) requires a significant amount of computation. Unlike ordinary motion blur, which can find neighboring pixels by simply incrementing or decrementing the x- and y-coordinates of the input pixel, radial blur must calculate a fresh set of neighbors for every input pixel due to the curved shape and changing size of

the blurring window. Fig. 7.3 shows that even adjacent pixels in a radial blur image have entirely different sets of neighbors.

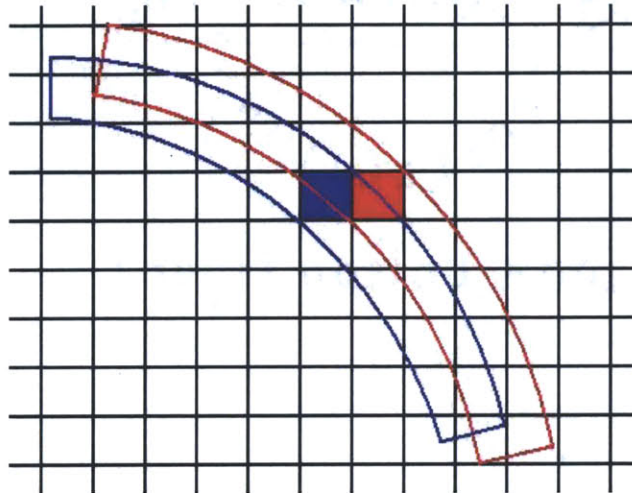


Fig. 7.3. Two sets of neighbors for adjacent pixels in a radial blur image.

Even if blurring is done in polar coordinates, a large amount of computation is still required. One major problem when performing radial blur in polar coordinates is that when integer Cartesian coordinates  $(x, y)$  are converted to polar coordinates  $(r, \theta)$ , neither  $r$  nor  $\theta$  is guaranteed to be an integer. As a result, computationally intensive floating-point arithmetic is unavoidable. In addition, neighboring pixels cannot be found by simply incrementing or decrementing integer polar coordinates. A new set of neighbors must be calculated explicitly for every pixel.

Whether blurring is done in Cartesian or polar coordinates, significant computational power and floating-point arithmetic are required. Most camera phones have chips with integer microprocessors. On such chips, floating-point arithmetic requires the use of additional floating-point emulators or software libraries, which lead to considerable increases in latency. Real-time or near real-time camera phone implementations must use strictly integer, or fixed-point, arithmetic. What is needed is a fast radial blur algorithm that can be implemented with only fixed-point arithmetic while producing high-quality results.

This thesis presents a new radial blur algorithm that is faster and simpler than traditional methods and can be used on both integer and floating-point microprocessors. The algorithm has been implemented in a radial blur function that allows users to specify the blur amount and the center of rotation, just as in Photoshop. The radial blur function also provides a new, strictly integer implementation needed for camera phones with integer microprocessors. The implementation achieves real-time or near real-time performance on camera phones, depending on the amount of blurring.

## 7.2 A New Algorithm

A key concept behind the new algorithm is that a radial blur image is the result of averaging different rotated versions of the original image. The algorithm creates high-quality results based on a new computational scheme that avoids explicitly finding neighboring pixels along circular arcs. It also avoids conversion to polar coordinates and therefore leads to a simpler, faster, and more efficient implementation.

Although the number of neighboring pixels  $M$  is never explicitly calculated or used in the algorithm, it can be found by

$$M = \frac{n}{360} \cdot 2\pi r, \quad (7.2)$$

where  $n$  is the blur amount and

$$r = \sqrt{(x - x_o)^2 + (y - y_o)^2}$$

for a pixel located at  $(x, y)$  with a center of rotation at  $(x_o, y_o)$ . By avoiding explicit calculation of neighbors for every pixel, this new algorithm reduces the number of computations by a factor of  $M$ . Such an improvement is substantial, particularly when  $M$  is large, as it is for pixels far from the center of rotation. Consider a typical case in which  $r = 300$  and  $n = 20$ . From equation (7.2),

$$M = \frac{20}{360} \cdot 2\pi \cdot 300 = 104,$$

which means that the number of computations has been reduced by more than 100 times.

To optimize for speed, the algorithm has three alternate paths, depending on the blur amount  $n$ . For each path, the original image is always rotated incrementally up to  $n$  degrees. The resulting rotated images are summed with the original image and then averaged to obtain the radial blur output. To balance out the blurring, half of the intermediate images are rotated clockwise and half are rotated counterclockwise. The different rotational increments used in the three paths of the algorithm are as follows:

- (1) If  $n = 1$ , rotate the original image by a maximum of  $1^\circ$  in increments of  $1/8^\circ$ . The eight rotations are  $1/8^\circ$ ,  $1/4^\circ$ ,  $3/8^\circ$ , and  $1/2^\circ$ , clockwise and counterclockwise. Sum up these eight intermediate images.
- (2) If  $2 \leq n \leq 5$ , do path (1) first. Do rotations for  $2^\circ$  to  $n^\circ$  in increments of  $1/4^\circ$  and multiply their sum by 2 (i.e. shift 1 bit to the left). Add this sum to the sum from path (1).

- (3) If  $n > 5$ , do path (2) first. For rotations greater than  $5^\circ$ , use an increment of  $1/2^\circ$  and multiply their sum by 4 (i.e. shift 2 bits to the left). Add this sum to the sum from path (2).

The final average is equal to the sum of rotated images and the original image, divided by  $8n + 1$ . Notice that the partial sums in paths (2) and (3) are multiplied by extra factors of 2 and 4, respectively. The extra factors compensate for the larger rotational increments used in these paths, as compared to path (1), and ensure that all degrees of rotation are equally weighted in the average.

Notice that the first  $1^\circ$  of rotation is performed with the smallest rotational increment. This ensures that all nearest-neighbor pixels are included in the final average. If radial blur must be performed on a megapixel image, smaller rotational increments are required, as explained below. Larger increments are used for larger degrees of rotation to minimize the amount of computation required.

To see why smaller blurring levels require smaller rotational increments, let us compare results for  $n = 1$  using different increment sizes. To get a more detailed view, a small region in each radial blur image has been enlarged in Fig. 7.5. Fig. 7.4 shows the location of this region within a full-sized 520 x 390 image. In Fig. 7.5, notice that large increments produce poor-quality blurring. This is because for each input pixel, some of the nearest-neighbor pixels have been skipped and are excluded from the pixel sum. Nearest neighbors are the most highly correlated with a given pixel. Excluding them from a pixel sum leads to incorrect blurring. For this image size, only an increment of  $1/8^\circ$  allows all nearest neighbors to be summed and therefore produces the best result.

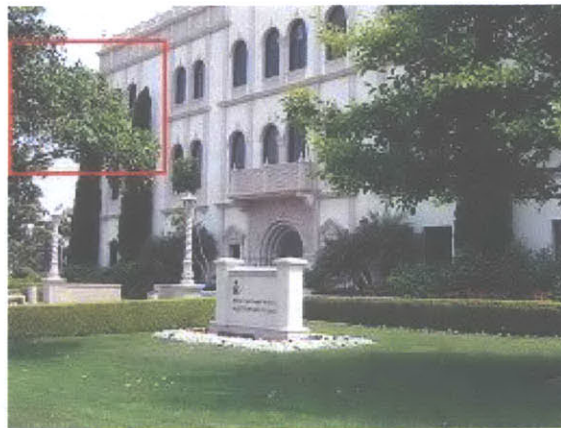


Fig. 7.4. Region of interest in the original 520 x 390 image.

For even faster execution speed, the algorithm can be streamlined by using larger rotational increments for pixels that are close to the center of rotation. For instance, pixels

located within a radius of 170 pixels from the center need only be rotated by increments of  $1/2^\circ$ , instead of  $1/8^\circ$ . Thus, when  $n = 1$ , the number of computations is reduced by a factor of four for each pixel in this region. TABLE 7.1 lists the optimal rotational increments for different distances from the center. The maximum image size supported by the table is 256 megapixels, which is sufficient for practical purposes. However, the table can be easily extended for infinitely large image sizes. In most camera phones, the image size is no more than a few megapixels. This means that in real life, the rotational increment will be no smaller than  $1/32^\circ$ . Figs. 7.6, 7.7, and 7.8 illustrate the different regions of rotational increments given in the table. During implementation, an index table is used to indicate the region to which each pixel belongs.

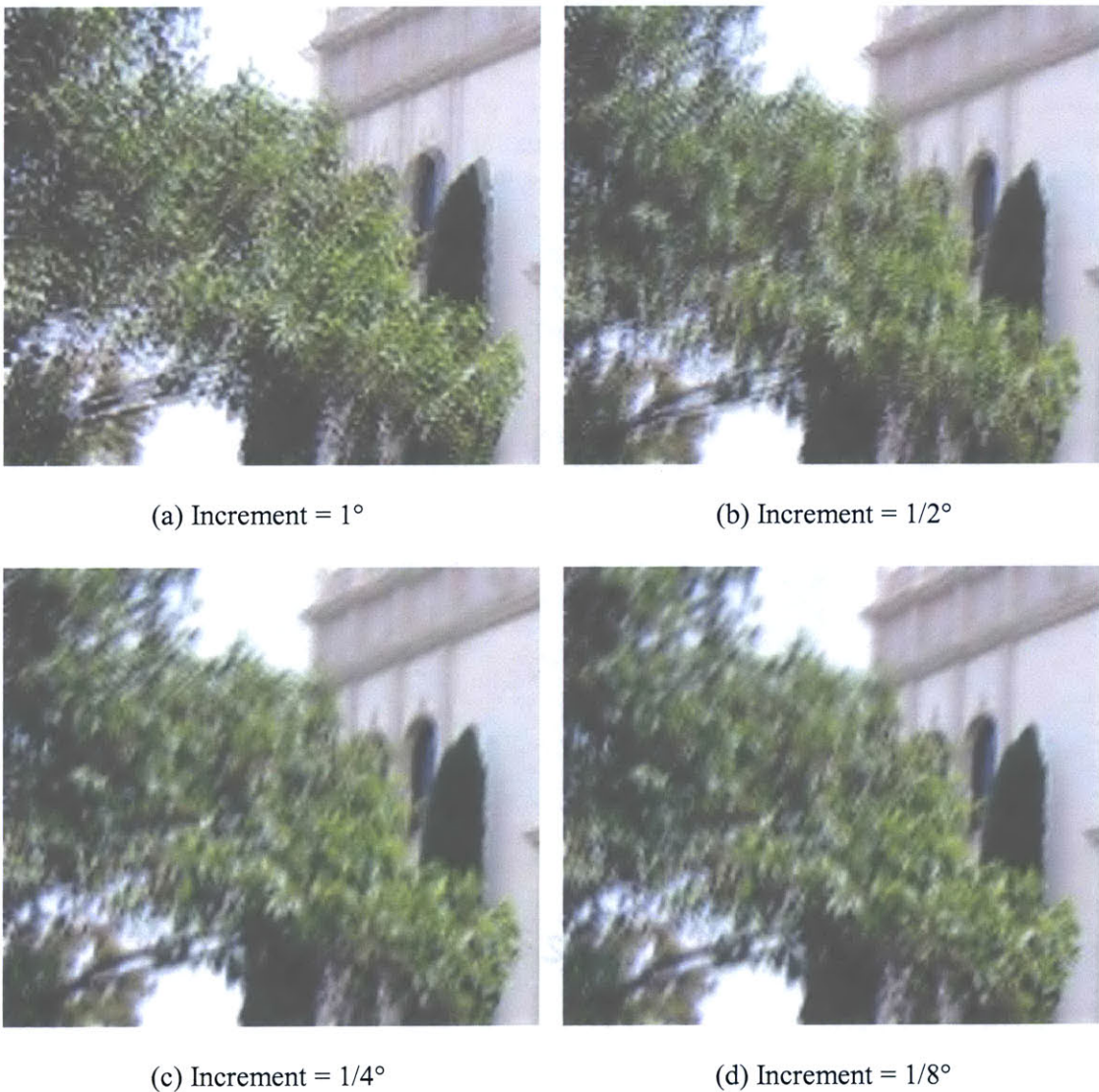


Fig. 7.5. Enlarged views of region of interest for different rotational increments.

TABLE 7.1: Rotational Increments for Angle  $\alpha$

Distance from Center of Rotation (pixels)	Rotational Increment for $\alpha = 1^\circ$	Rotational Increment for $2^\circ \leq \alpha \leq 5^\circ$	Rotational Increment for $\alpha > 5^\circ$
0 – 170	$1/2^\circ$	$1^\circ$	$1^\circ$
171 – 284	$1/4^\circ$	$1/2^\circ$	$1^\circ$
285 – 512	$1/8^\circ$	$1/4^\circ$	$1/2^\circ$
513 – 967	$1/16^\circ$	$1/4^\circ$	$1/2^\circ$
967 – 3698	$1/32^\circ$	$1/4^\circ$	$1/2^\circ$
3699 – 7339	$1/64^\circ$	$1/4^\circ$	$1/2^\circ$
> 7339	$1/128^\circ$	$1/4^\circ$	$1/2^\circ$

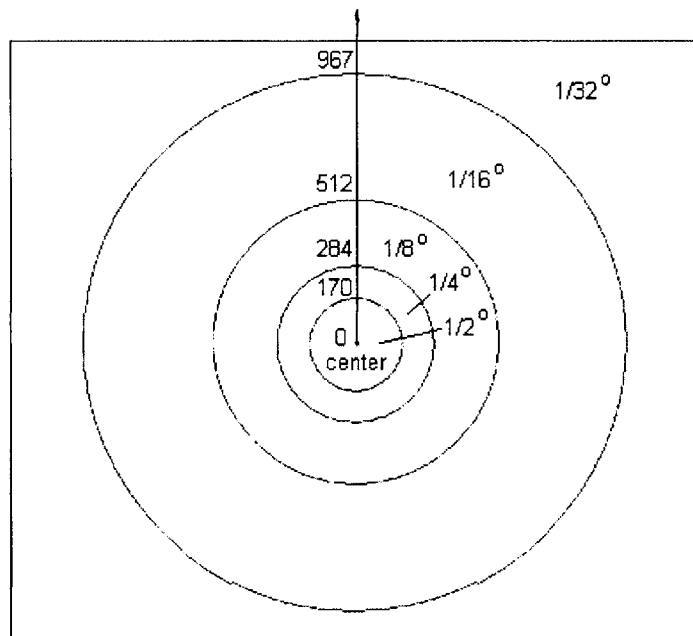


Fig. 7.6. Regions of different rotational increments for  $\alpha = 1^\circ$ . Regions for the two smallest increments have been omitted.

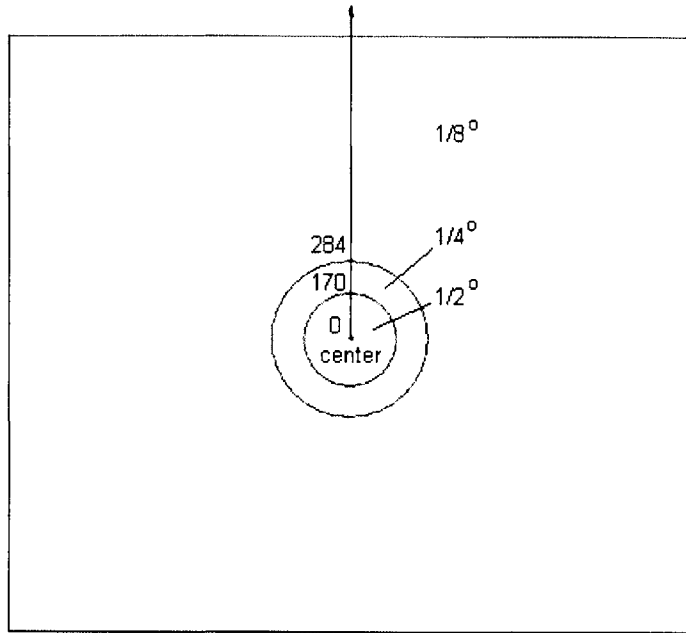


Fig. 7.7. Regions of different rotational increments for  $2^\circ \leq \alpha \leq 5^\circ$ .

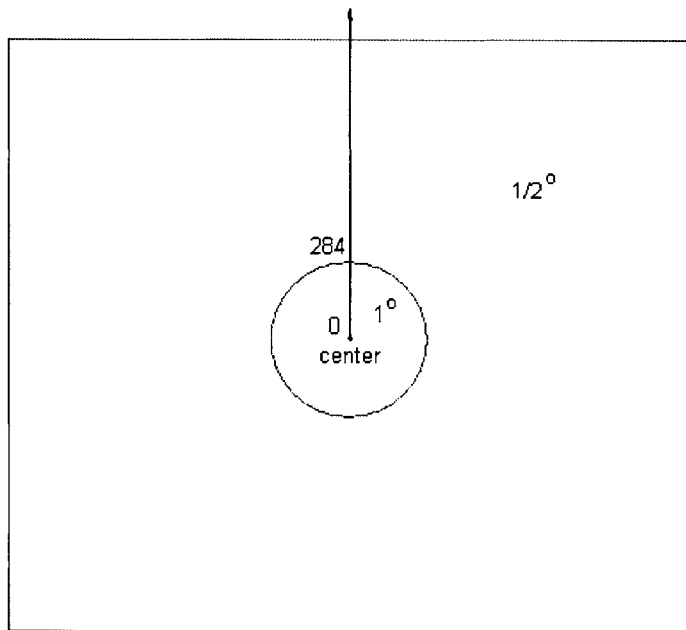


Fig. 7.8. Regions of different rotational increments for  $\alpha > 5^\circ$ .



The radial blur algorithm can be streamlined by using the optimized rotational increments in TABLE 7.1:

- (1) If  $n = 1$ , use the increments for  $\alpha = 1^\circ$ .
- (2) If  $2 \leq n \leq 5$ , perform the first  $1^\circ$  of rotation in the same way as (1). For rotations greater than  $2^\circ$ , use increments for  $2^\circ \leq \alpha \leq 5^\circ$ .
- (3) If  $n > 5$ , perform the first  $5^\circ$  of rotation in the same way as (2). For rotations greater than  $5^\circ$ , use increments for  $\alpha > 5^\circ$ .

To clarify the above algorithm, consider the case of a 1200 x 1200 image. Suppose  $n = 1$  and the center of rotation is the center of the image. Fig. 7.6 shows that there are four different regions of rotational increments, ranging from  $1/16^\circ$  to  $1/2^\circ$ . Rather than rotating each region separately, the entire image is rotated 16 times using the smallest increment of  $1/16^\circ$ . Regions with larger increments simply exclude irrelevant rotated images from their pixel sums. The four regions are handled as follows: In the region beyond 512 pixels from the center, all 16 rotated images are summed. In the region between 284 and 512 pixels from the center, only every other rotated image is summed. This means only 8 rotated images are summed. Their sum is multiplied by 2 to ensure that all regions are equally weighted in the final average. In the region between 170 and 284 pixels, only every fourth rotated image is summed, giving a total of 4 images. Their sum is multiplied by 4. In the region between 0 and 170 pixels, only every eighth rotated image is summed, giving a total of 2 images, whose sum is multiplied by 8. For all regions, the final average is equal to the weighted sum of rotated images, plus the original image, divided by  $16n + 1$ .

If  $2 \leq n \leq 5$ , the first  $1^\circ$  of rotation is done in the same way as  $n = 1$ . Rotations of  $2^\circ$  through  $n^\circ$  are done according to Fig. 7.7. The entire image is rotated by the smallest increment, which is  $1/8^\circ$ . The sum for each region includes only relevant rotated images and is multiplied by an appropriate factor to ensure equal weighting for all regions. This factor can be found in the same way as  $n = 1$ . For instance, the region beyond 284 pixels from the center sums all 8 rotated images, whereas the region between 170 and 284 pixels only sums every other rotated image and multiplies the sum by 2. In addition, an extra factor of 2 is needed for the sum in every region. This is because the smallest rotational increment for  $2^\circ$  through  $n^\circ$  is twice the smallest increment used for  $1^\circ$ . Multiplying by the additional factor of 2 ensures equal weighting for all degrees of rotation. The final average equals the sum of rotated images and the original image, divided by  $16n + 1$ .

The same approach is extended for  $n > 5$ . The first  $5^\circ$  of rotation are done in the same way as  $2 \leq n \leq 5$ . Rotations greater than  $5^\circ$  are done according to Fig. 7.8. Again, the entire image is rotated by the smallest increment, which is  $1/2^\circ$ . Each region sums only the relevant rotated images and multiplies the sum by a suitable factor to ensure equal weighting for all regions. The sum for  $6^\circ$  through  $n^\circ$  for every region is multiplied by an additional factor of 8 to ensure equal weighting for all degrees of rotation. The final

average is equal to the three weighted rotation sums for  $1^\circ$ ,  $2^\circ$  through  $5^\circ$ , and  $6^\circ$  through  $n^\circ$ , plus the original image, divided by  $16n + 1$ .

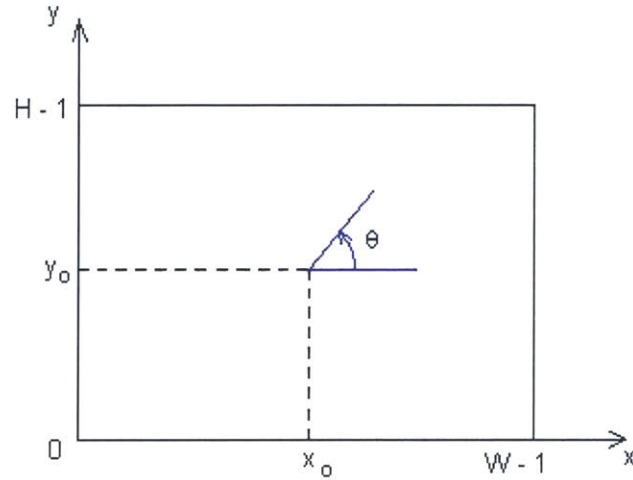


Fig. 7.9. Coordinate system used in equations (7.2) and (7.3) for radial blur.



Fig. 7.10. Intermediate image rotated by  $5^\circ$  counterclockwise.

Each intermediate rotation is performed by using a standard affine transform to map input pixels in the original image to output pixels in the rotated image. The equations are

$$x_{out} = x_o + (x_{in} - x_o) \cdot \cos \theta - (y_{in} - y_o) \cdot \sin \theta \quad (7.2)$$

$$y_{out} = y_o + (x_{in} - x_o) \cdot \sin \theta + (y_{in} - y_o) \cdot \cos \theta \quad (7.3)$$

where  $\theta$  is the rotation angle in radians,  $(x_o, y_o)$  is the center of rotation,  $(x_{in}, y_{in})$  is the input pixel location, and  $(x_{out}, y_{out})$  is the output pixel location. The coordinate

system used in these equations has an origin located at the lower left corner of the image, an x axis pointing to the right, and a y axis pointing upward, as shown in Fig. 7.9.

There is one caveat for the algorithm. Each intermediate rotated image will not overlap all parts of the original image. Fig. 7.10 shows that an intermediate image rotated by  $5^\circ$  does not overlap the corners of the original image. As a result, not every pixel in the radial blur image will have the same number of pixel values to average. A counter must be used for each pixel in the original image to keep track of the number of missing rotated pixels. The key point here is that the counter must be incremented differently depending on the rotational increment used in a particular pixel region. For instance, in the  $1200 \times 1200$  image discussed previously, where the smallest rotational increment is  $1/16^\circ$ , pixels in a region using a rotational increment of  $1/16^\circ$  should increase the counter by 1 for every missing rotated pixel, whereas pixels in a region with a rotational increment of  $1/8^\circ$  should increase the counter by 2. Each region increases the counter differently based on the region's rotational increment. This counter is subtracted from  $16n + 1$  to obtain the correct divisor for calculating the pixel average for that region.

### 7.3 Integer Implementation

If the radial blur algorithm must be implemented on an integer microprocessor, the sine and cosine values in equations (7.2) and (7.3) must be calculated using strictly integer, or fixed-point, arithmetic. This section describes how to achieve an integer implementation.

Suppose the angle of rotation is  $\alpha$  degrees. For simplicity, it is assumed that  $\alpha$  is in the range  $-50^\circ \leq \alpha \leq 50^\circ$ . Such an assumption makes it easier to compare results with Photoshop, which only supports blur amounts between 0 and 100, thus limiting all rotations to a maximum of  $50^\circ$  clockwise or counterclockwise. The new radial blur algorithm presented in the previous section remains applicable for any blur amount and any angle  $\alpha$ . To handle angles of rotation outside the range  $-50^\circ \leq \alpha \leq 50^\circ$ , refer to Chapter 12.

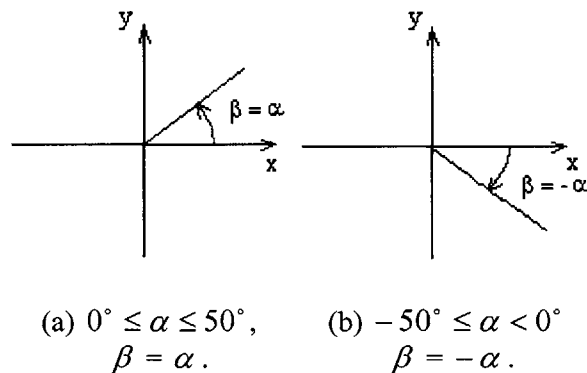


Fig. 7.11. Mapping from  $\alpha$  to  $\beta$ .

If  $\alpha < 0^\circ$ , it is mapped to an equivalent angle  $\beta$  in the range  $0^\circ \leq \beta \leq 50^\circ$  before equations (7.2) and (7.3) are applied. This mapping is shown in Fig. 7.11.

The sine and cosine relationships between  $\alpha$  and  $\beta$  are as follows:

$$\text{If } 0^\circ \leq \alpha \leq 50^\circ, \quad \beta = \alpha \quad \text{and} \quad \begin{cases} \sin \alpha = \sin \beta \\ \cos \alpha = \cos \beta \end{cases}$$

$$\text{If } -50^\circ \leq \alpha < 0^\circ, \quad \beta = -\alpha \quad \text{and} \quad \begin{cases} \sin \alpha = -\sin \beta \\ \cos \alpha = \cos \beta \end{cases}$$

To apply transformation equations (7.2) and (7.3) using fixed-point arithmetic, Taylor series expansions are used to approximate the sine and cosine functions. In general, the series expansions for the sine and cosine of an arbitrary radian angle are

$$\sin \theta = \theta - \frac{1}{3!}\theta^3 + \frac{1}{5!}\theta^5 - \frac{1}{7!}\theta^7 + \dots \quad (7.4)$$

$$\cos \theta = 1 - \frac{1}{2!}\theta^2 + \frac{1}{4!}\theta^4 - \frac{1}{6!}\theta^6 + \dots \quad (7.5)$$

where  $\theta$  is the equivalent of  $\beta$  in radians. That is,

$$\theta = \beta \cdot \frac{\pi}{180^\circ} \quad (7.6)$$

If  $\beta$  is a fractional angle, such as  $1/8^\circ$ ,  $\beta$  is first converted to fixed-point format by multiplying with a suitably large integer factor  $f$ , such as  $f = 2^7 = 128$ . All intermediate arithmetic operations are performed using this fixed-point representation. The final result is obtained by dividing by  $f$ .

When  $0^\circ \leq \beta \leq 40^\circ$ , only the first two terms of equations (7.4) and (7.5) are needed to produce accurate results. When  $40^\circ < \beta \leq 50^\circ$ , the first four terms in the series are needed for an accurate approximation. Higher order terms can still be used if desired. However, these terms have little effect on accuracy and only decrease the speed of the implementation. The radial blur function was implemented with the following approximations:

$$\sin \theta \cong \begin{cases} \theta - \frac{1}{3!}\theta^3 & \text{if } 0^\circ \leq \beta \leq 40^\circ \\ \theta - \frac{1}{3!}\theta^3 + \frac{1}{5!}\theta^5 - \frac{1}{7!}\theta^7 & \text{if } 40^\circ < \beta \leq 50^\circ \end{cases} \quad (7.7)$$

$$\cos \theta \cong \begin{cases} 1 - \frac{1}{2!}\theta^2 & \text{if } 0^\circ \leq \beta \leq 40^\circ \\ 1 - \frac{1}{2!}\theta^2 + \frac{1}{4!}\theta^4 - \frac{1}{6!}\theta^6 & \text{if } 40^\circ < \beta \leq 50^\circ \end{cases} \quad (7.8)$$

In equations (7.7) and (7.8), the relationship between  $\theta$  and  $\beta$  is given by equation (7.6). Although equations (7.6), (7.7), and (7.8) do not contain strictly integer terms, the non-integer terms can be converted to fixed-point numbers through multiplication by a suitably large integer factor. For instance, in equation (7.6),  $\frac{\pi}{180^\circ}$  is converted to fixed-point format by computing it as a real number, multiplying by  $2^9 = 512$ , and rounding to the nearest integer. This changes  $\frac{\pi}{180^\circ}$  to 9.

Likewise, in equation (7.7),  $\frac{1}{3!}$  is converted to fixed-point format by computing it as a real number, multiplying by  $2^{10} = 1024$ , and rounding to the nearest integer. Thus,  $\frac{1}{3!}$  becomes  $\frac{1}{3!} \cdot 2^{10} = (0.16667) \cdot 1024 \cong 171$ . Other non-integer terms such as  $\frac{1}{5!}$  can be converted to fixed-point numbers in the same manner. All intermediate arithmetic operations are performed using the new fixed-point representations. The final result is obtained by dividing by the same integer factor, e.g.  $2^{10}$  for equation (7.7). This technique preserves accuracy during intermediate integer arithmetic operations. For more details on the method, a typical integer implementation is given in the next section.

## 7.4 Avoiding Integer Overflow

The new radial blur algorithm was implemented on a 32-bit RISC integer microprocessor. To reduce latency and achieve real-time performance, no integers greater than 32 bits were used. This 32-bit restriction caused problems with integer overflow when calculating sine and cosine values for intermediate rotations. The solution was to split up arithmetic operations needed to calculate the trigonometric values and rearrange the order of operations to avoid overflow while maintaining the necessary degree of accuracy. This method of dealing with integer overflow can be applied to any implementation on any microprocessor, including 64-bit processors, 128-bit processors, and so forth.

The following is a sample 32-bit integer implementation in C for equations (7.2) and (7.3) using the Taylor series approximations in equations (7.7) and (7.8). This example assumes that the smallest angle of rotation is  $1/8^\circ$ , clockwise or counterclockwise. The input variable `angle` equals the real angle of rotation multiplied by  $2^3 = 8$  to ensure that all fractional angles are converted to fixed-point format. A larger multiplicative factor

can be used if an implementation is needed for fractional angles smaller than  $1/8^\circ$ , such as those listed in TABLE 7.1. For instance, an angle of  $1/16^\circ$  can be multiplied by  $2^4 = 16$  to become a fixed-point number.

```

uint32 beta, radian;
int8 sinSign;
int32 xin, yin, xout, yout, sintheta, costheta;

/*
** "angle" is a signed 32-bit integer (int32) variable indicating
** the angle of rotation multiplied by  $2^3 = 8$ . Positive angles
** produce clockwise rotation. Negative angles produce
** counterclockwise rotation.
*/
sinSign = 1;
if (angle < 0) {
    beta = -angle;
    sinSign = -1;
}

/*
** Convert beta from degrees to radians using fixed-point
** arithmetic. Divide by 8 using ">> 3" to compensate for the
** extra factor of 8 in "angle". The valid range for beta is
**  $0 \leq \beta \leq 50$ .
*/
radian = (beta * 9) >> 3;

/* Calculate sin and cos */
if (radian <= 360) {
    /* beta <= 40 degrees: use first 2 terms of Taylor series */
    sintheta = ((radian*(262144-(radian*radian)/6))>>8)*sinSign;
    costheta = 524288-radian*radian;
} else {
    /* 40 < beta <= 50: use first 4 terms of Taylor series */
    sintheta = (((radian*(262144-(radian*radian)/6))>>8) +
        (((((((radian*radian*radian)/120)>>3)*radian)>>10)*
            radian)/42)*((11010048-radian*radian)>>10))>>21))*
        sinSign;
    costheta = (524288-radian*radian) +
        (((((((radian*radian*radian)/24)>>3)*radian/30)>>10)*
            ((7864320-radian*radian)>>10))>>12);
}

/*
** For each output pixel (xout, yout), map to the corresponding
** input pixel (xin, yin) using these equations. The center of
** rotation is (xo, yo), where xo and yo are 32-bit integers.
*/
xin = xo + (((xout-xo)*costheta + (yout-yo)*sintheta)>>19);
yin = yo + (((xo-xout)*sintheta + (yout-yo)*costheta)>>19);

```

In this implementation,  $w$  is the width of the image and  $h$  is the height, both of which are 32-bit integers. A similar implementation can be made in C++, Java, or any other programming language.

## 7.5 Results

The new radial blur algorithm was implemented and tested for the full range of blur amounts from 0 to 100, which is the same range supported by Photoshop 7.0. All blur amounts must be integers. A blur amount of 0 produces no change in the original image.



Fig. 7.12. Original 520 x 390 image.

For convenience, Fig. 7.12 gives the original 520 x 390 image previously shown in Fig. 7.1 (a). Fig. 7.13 shows radial blur results for a variety of blur amounts. All results were produced with a center of rotation located at the center of the image. Fig. 7.13 (a) shows the result for a blur amount of 1. This was obtained with a rotational increment of  $1/8^\circ$ . An enlarged region of the image was shown previously in Fig. 7.5 (d) as part of a comparison between different rotational increments.

Fig. 7.13 (b), (c), and (d) show results for blur amounts of 10, 50, and 100, respectively. To obtain these results, the first  $1^\circ$  of rotation was done in increments of  $1/8^\circ$ , rotations of  $2^\circ$  through  $5^\circ$  were done in increments of  $1/4^\circ$ , and the remainder were done in increments of  $1/2^\circ$ . Fig. 7.13 (d) corresponds to the maximum blur amount supported by Photoshop 7.0. The new radial blur algorithm in this thesis supports arbitrarily large blur amounts.

Fig. 7.14 shows radial blur results from the new algorithm using different centers of rotation. In Fig. 7.14 (a) and (b), the centers are at (168, 207) and (345, 171), respectively. Both images were produced with a blur amount of 30.



(a) Blur Amount = 1



(b) Blur Amount = 10



(c) Blur Amount = 50



(d) Blur Amount = 100

Fig. 7.13. Radial blur results from the new algorithm using different blur amounts.



(a) Center = (168, 207)



(b) Center = (345, 171)

Fig. 7.14. Radial blur results from the new algorithm using different centers of rotation. Blur amount is 30.



## 7.6 Comparison with Photoshop

Photoshop 7.0 offers a radial blur function whose results are slightly different from those of the new algorithm proposed here. In fact, the radial blur images produced by Photoshop do not have strictly one-directional blur along concentric circles. However, by definition, radial blur is a position-dependent directional blur, where each pixel is blurred in only one direction and this direction varies according to the pixel's location relative to the center of rotation.

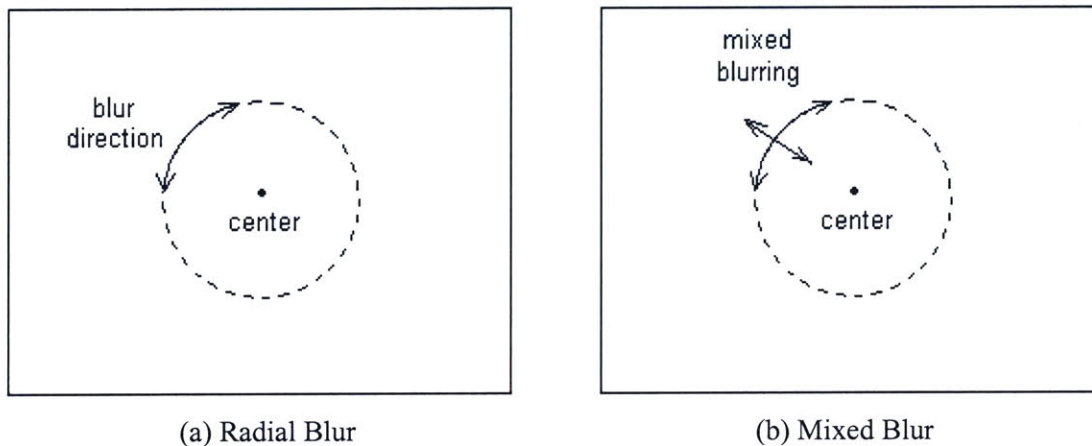


Fig. 7.15. Radial blur versus mixed blur.

To ensure a smooth radial blur, Photoshop uses bilinear or bicubic interpolation [19]. The result is a mixture of radial and centripetal blurring. The correct radial blur direction is shown in Fig. 7.15 (a), where blurring occurs only along a circular arc. Fig. 7.15 (b) shows a mixed blur where blurring occurs not only along the circular arc but also perpendicular to it. Photoshop's interpolation might cause this type of mixed blur.



Fig. 7.16. Region of interest in the original 520 x 390 image.

Fig. 7.17 compares Photoshop's best quality radial blur with results from the new algorithm. Both results used a blur amount of 30. For ease of comparison, a small region of interest from each full-size image has been cropped and enlarged. Fig. 7.16 shows the location of this region relative to the rest of the image. Fig. 7.18 shows the full-sized 520 x 390 radial blur results.

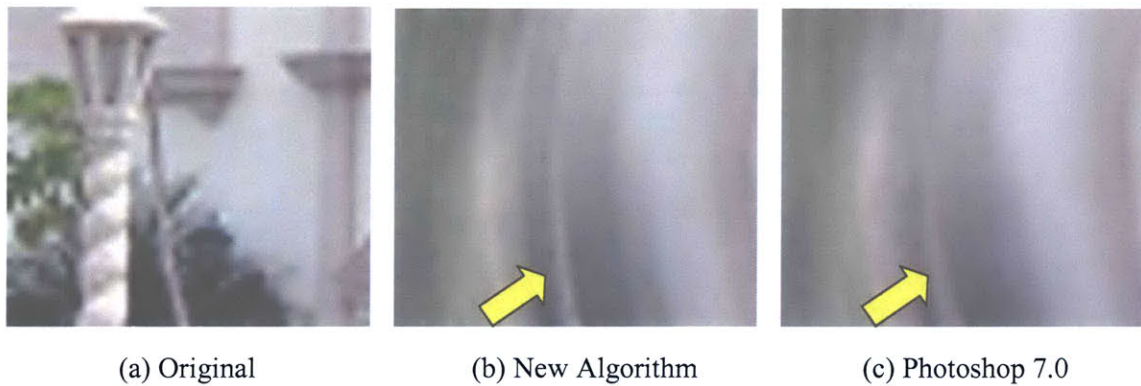


Fig. 7.17. Radial blur comparison in the region of interest.

In Fig. 7.17, notice that the white streak indicated by the arrow is wider in the Photoshop result than in the new algorithm's result. The concentric circular streaks of blurring are also less noticeable in the Photoshop image. This shows that Photoshop produces a slightly mixed blur.

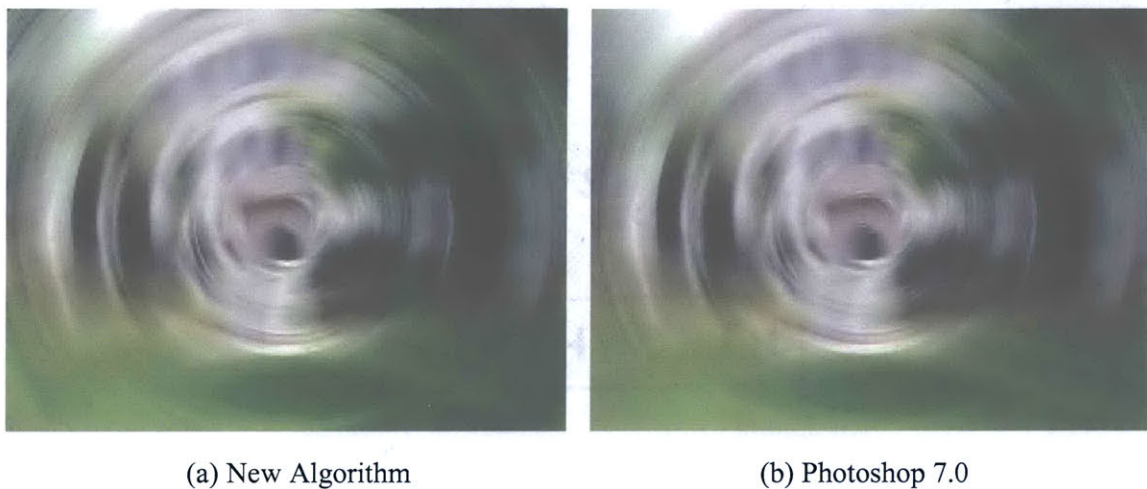


Fig. 7.18. Radial blur comparison with full-sized 520 x 390 images.

## 7.7 Implementation for Subsampled YCbCr and YCrCb

The current implementation supports RGB, YCbCr, and YCrCb color formats, including subsampled formats such as YCbCr 4:2:2, YCbCr 4:2:0, and YCrCb 4:2:0. RGB images can be processed by directly applying equations (7.2) and (7.3) along with the Taylor series approximation in equations (7.7) and (7.8). However, images in a subsampled YCbCr or YCrCb color format cause problems due to their subsampled chrominance components, Cb and Cr.

0	1	2	3
YCb	YCr	YCb	YCr
YCb	YCr	YCb	YCr

Fig. 7.19. H2V1 YCbCr 4:2:2.

For instance, one common form of subsampled YCbCr is YCbCr 4:2:2, in which the Cb and Cr components of either pixel rows or pixel columns are subsampled by a factor of two. Two formats are H2V1 YCbCr 4:2:2, in which pixel columns are subsampled but pixel rows are unaffected, and H1V2 YCbCr 4:2:2, in which pixel rows are subsampled but pixel columns are unaffected. Fig. 7.19 shows the H2V1 format. For every 2 x 2 block of pixels, there are 4 Y values, 2 Cb values, and 2 Cr values, hence the name 4:2:2. If columns are numbered starting from zero, only even columns have the Cb component and only odd columns have Cr. Hereafter, YCbCr 4:2:2 refers to H2V1 YCbCr 4:2:2.

YCrCb formats are the same as YCbCr but with the order of Cb and Cr components switched. For instance, an H2V1 YCrCb 4:2:2 image has the pixel arrangement shown in Fig. 7.20. Hereafter, YCrCb 4:2:2 refers to H2V1 YCrCb 4:2:2.

0	1	2	3
YCr	YCb	YCr	YCb
YCr	YCb	YCr	YCb

Fig. 7.20. H2V1 YCrCb 4:2:2.

Directly applying equations (7.2) and (7.3) on a YCbCr 4:2:2 image will fail to guarantee correctly alternating YCb-YCr pixels in the transformed image. To solve this problem, a temporary YCbCr 4:4:4 image can be created from the original subsampled YCbCr 4:2:2 image. The equations are then applied to the YCbCr 4:4:4 image to obtain a temporary YCbCr 4:4:4 output, which is subsampled to produce the final YCbCr 4:2:2 output.

The temporary YCbCr 4:4:4 image contains all three YCbCr components for each pixel. It is created by taking pairs of adjacent YCb and YCr pixels and having each pixel borrow the missing chrominance component from its partner. The YCb pixel borrows a Cr component from its paired YCr pixel. The YCr pixel borrows a Cb component from the YCb pixel. Normally, this method does not produce an accurate, visually pleasing YCbCr 4:4:4 image from a YCbCr 4:2:2 image. However, since the temporary YCbCr 4:4:4 image is used only for intermediate processing, the accuracy of the intermediate result does not matter as long as the final output is accurate.

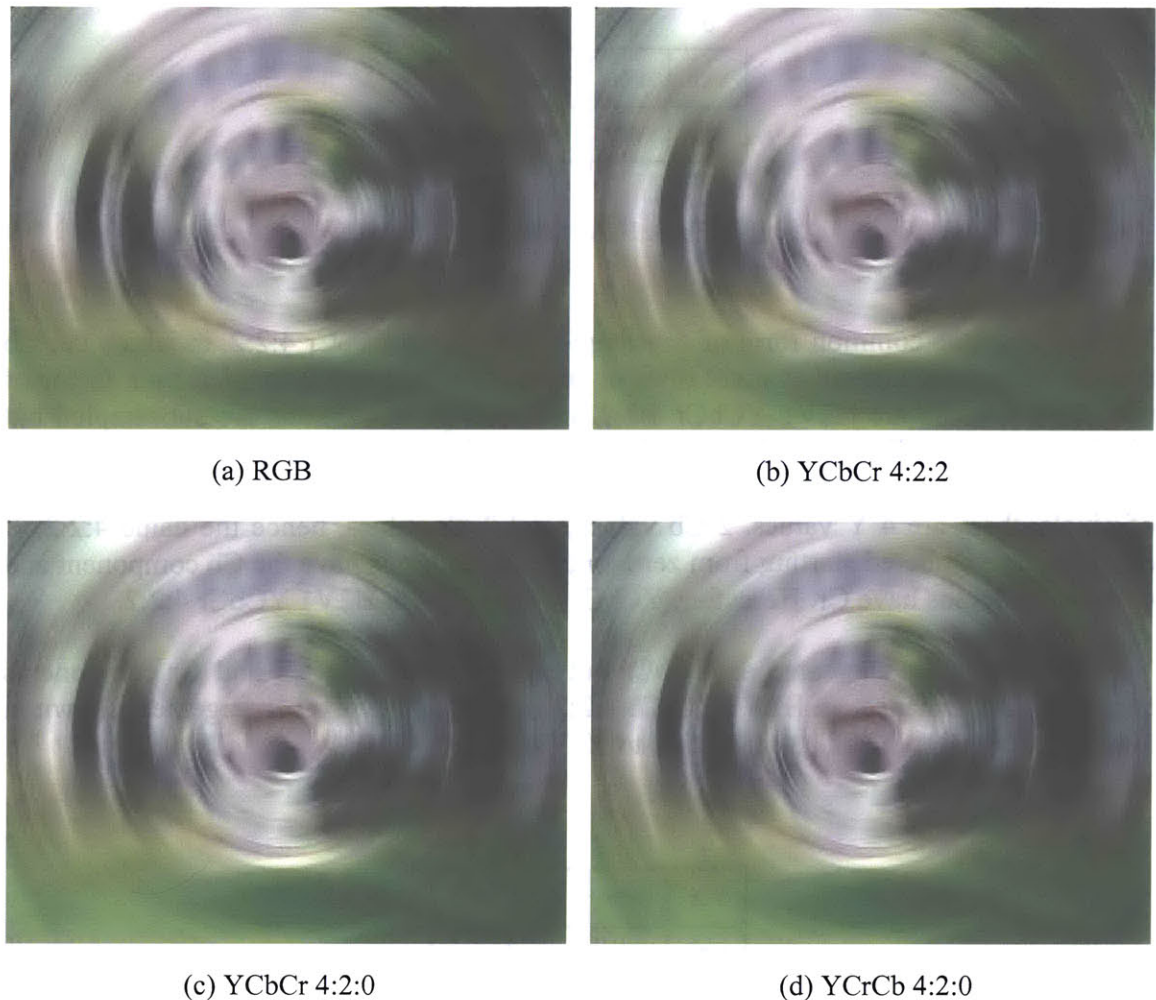


Fig. 7.21. Comparison of radial blur results for 4 different color formats. Blur amount is 30.

For the purpose of applying equations (7.2) and (7.3), this method of creating a YCbCr 4:4:4 image and performing transformations in the YCbCr 4:4:4 domain works very well. The final YCbCr 4:2:2 output is obtained from the temporary YCbCr 4:4:4 output by discarding the extra Cb or Cr component for each pixel. There is no visible difference between a YCbCr 4:2:2 transformed image produced in this manner and an equivalent RGB transformed image. Fig. 7.21 (a) and (b) compare an RGB radial blur

result produced from RGB input with the corresponding YCbCr 4:2:2 result produced from YCbCr 4:2:2 input. Both images were produced with a blur amount of 30 and a center of rotation located at the center of the image.

This method of creating a temporary YCbCr 4:4:4 image can be extended to handle input images in any subsampled YCbCr or YCrCb color format. For instance, another common format is YCbCr 4:2:0, in which each block of 2 x 2 pixels has only one Cb and one Cr component. A temporary YCbCr 4:4:4 image is created by duplicating the Cb and Cr components for each pixel in each 2 x 2 block. Transformations are applied to the temporary YCbCr 4:4:4 image to obtain a YCbCr 4:4:4 output, which is subsampled to YCbCr 4:2:0 by discarding extra chrominance components. There is no visible difference between YCbCr 4:2:0 radial blur results produced in this manner and RGB results produced from RGB input. This can be seen by comparing Fig. 7.21 (a) with (c), which shows results for YCbCr 4:2:0. Fig. 7.21 (d) shows results for YCrCb 4:2:0, where the order of Cb and Cr components has been switched. All images were produced with a blur amount of 30 and a center of rotation located at the center of the image.



# Chapter 8

## Motion Blur

### 8.1 Introduction

In real life, motion blur occurs when there are objects moving during the exposure time of a camera. Fig. 8.1 gives an example of real-life motion blur. The cyclist in the foreground is blurred because of his fast movement relative to the camera's shutter speed. In general, motion blur can occur for a variety of reasons. The primary cause is relative movement between an object and a camera. Either the object or the camera, or both, are moving during exposure time. Another cause is movement of the camera shutter. The opening and closing time of the shutter, the direction of movement of the shutter, and changes in the shape of the shutter's aperture can all affect the appearance of motion blur in a photograph [33].



Fig. 8.1. Real-life motion blur (Microsoft Word 2000 SP3 clip art).

In image processing, the problem of characterizing the degradation caused by motion blur in an optical system has long been an area of research [4]. In image restoration, an appropriate degradation function is generated to model the motion blur and used to recover the original unblurred image. The inverse to this problem is the problem of synthesizing and adding artificial motion blur.

Motion blur can be added to a static image to create the illusion of speed or movement. It can add realism to computer generated images by incorporating the optical effects of an

actual camera. Motion blur is also useful in animation sequences for the removal of temporal aliasing effects [33]. Without motion blur, the animation would be choppy and would simulate an instantaneous camera shutter. Animated scenes would appear to be illuminated by quick repeated flashes of a strobe light. Adding motion blur alleviates the strobing effect [28].

## 8.2 Algorithm

The motion blur function produces one-dimensional uniform motion blur in an arbitrary direction for an arbitrarily large blur amount. To create motion blur, one straightforward approach is to rotate the image by the desired angle of motion, apply horizontal motion blur, and rotate the image back to its original orientation.

Suppose the angle of motion is  $\alpha$  and the blur amount is  $n$ . The value of  $n$  is limited to odd positive integers since it determines the length of the motion blur window, which must be odd, as explained below. The first step in the algorithm rotates the image clockwise by  $\alpha$ . Rotation is done using the algorithm in Chapter 12.

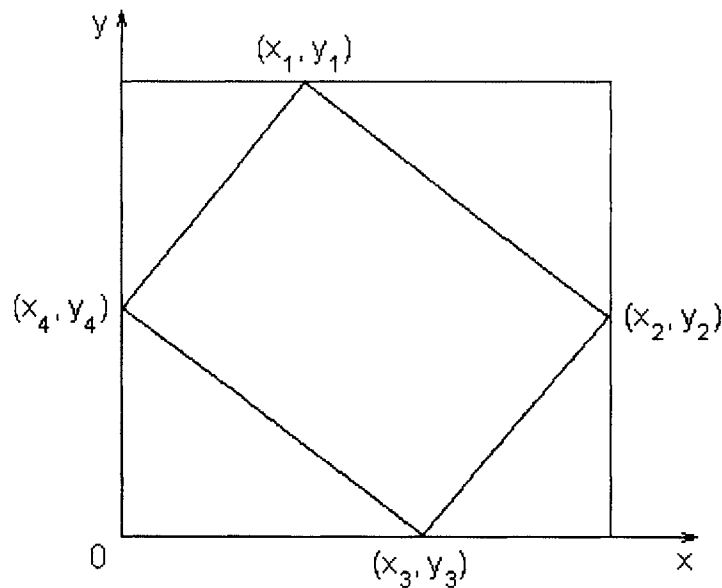


Fig. 8.2. Rotated corner positions.

One caveat is that the intermediate image must be large enough to fit the entire rotated result. Otherwise, parts of the original image would be truncated during rotation. To determine the necessary size of the intermediate image, the coordinate locations of the four corners of the rotated image must be calculated using the rotation equations given in Chapter 12. The coordinate system has an origin at the lower left corner of the image, an x axis pointing to the right, and a y axis pointing upward, as shown in Fig. 8.2.





Fig. 8.3. Original 520 x 390 image.



Fig. 8.4. Intermediate rotated result for  $\alpha = 30^\circ$ .

Based on the rotated corner positions, the minimum and maximum x- and y-coordinates in the rotated image can be found as follows:

$$\begin{aligned}x_{\min} &= \min(x_1, x_2, x_3, x_4), \\x_{\max} &= \max(x_1, x_2, x_3, x_4), \\y_{\min} &= \min(y_1, y_2, y_3, y_4), \\y_{\max} &= \max(y_1, y_2, y_3, y_4),\end{aligned}$$

where  $(x_1, y_1)$ ,  $(x_2, y_2)$ ,  $(x_3, y_3)$ , and  $(x_4, y_4)$  are the four corners shown in Fig. 8.2. The width  $w$  and height  $h$  of the intermediate image must be

$$w = x_{\max} - x_{\min} + 1,$$

$$h = y_{\max} - y_{\min} + 1.$$

Typically, the rotated result will not occupy all the space within the intermediate image. Fig. 8.3 shows an original  $520 \times 390$  image. Fig. 8.4 shows the intermediate rotated image, where  $\alpha = 30^\circ$  and blank parts are black, i.e. with pixel values set to  $(R, G, B) = (0, 0, 0)$ . A temporary Boolean array is used to keep track of which parts in the intermediate image are occupied by the actual rotated result and which parts are blank.

The second step in the algorithm applies horizontal motion blur to the intermediate rotated image. This is done by convolving the image with a one-dimensional motion blur window of length  $n$ . For each pixel in the intermediate image, the window is centered on the pixel, the intensity values of all pixels within the window are summed, and the arithmetic average is assigned as the output pixel value.

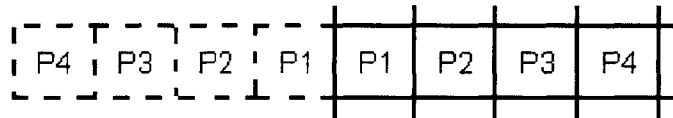


Fig. 8.5. Padding the left image boundary by “reflecting” pixels.

There are three key issues that need to be specially noted in this step. First, the symmetry of the motion blur window and the fact that the window must be centered on a pixel require the window length  $n$  to be an odd positive integer. Recall that  $n$  is also the blur amount. This is why the blur amount must always be an odd positive integer.

Second, only non-blank pixels are summed and averaged. The pre-computed Boolean array from step one is used to check whether each pixel within the window belongs to the rotated image or to a blank region. All blank pixels are ignored and the divisor in the average is adjusted accordingly. If there are  $\beta$  blank pixels, the adjusted divisor is  $n - \beta$ .

Third, pixels within a distance of  $(n - 1)/2$  from the left or right edge of the image must deal with the problem of having part of the window fall outside the image. One solution is to pad the left and right boundaries using pixel duplication. There are many duplication schemes. A common scheme is to “reflect” pixels across the boundary. Fig. 8.5 shows how to pad the left image boundary by “reflecting” pixels. Missing pixels outside the image are essentially substituted by pixels within the image. The final result from step two of the algorithm is a blurred intermediate image, as shown in Fig. 8.6, where  $\alpha = 30^\circ$  and  $n = 25$ .



Fig. 8.6. Blurred intermediate image for  $\alpha = 30^\circ$  and  $n = 25$ .



Fig. 8.7. Final motion blur result for  $\alpha = 30^\circ$  and  $n = 25$ .

The third and final step rotates the blurred intermediate image counterclockwise by  $\alpha$  and obtains an output image that is the same size as the original input image. As in step one, rotation is done according to the algorithm in Chapter 12. Fig. 8.7 shows a sample result for  $\alpha = 30^\circ$  and  $n = 25$ .

### 8.3 Integer Implementation

The motion blur algorithm was implemented on an integer microprocessor using strictly fixed-point arithmetic. Although the pixel sums and averages can be calculated easily

using fixed-point arithmetic, the rotations in the first and third steps of the algorithm require a special integer implementation which is described in Chapter 12. This is due to the sine and cosine terms that must be calculated when rotating an image.

## 8.4 Results

Fig. 8.8 shows two motion blur results. In Fig. 8.8 (a), the parameters are  $\alpha = 135^\circ$  and  $n = 21$ . In Fig. 8.8 (b),  $\alpha = 0^\circ$  and  $n = 35$ , which simply produce horizontal motion blur. Note that the blur amount  $n$  must always be a positive odd integer. If  $n = 1$ , no blurring occurs.



(a)  $\alpha = 135^\circ$ ,  $n = 21$

(b)  $\alpha = 0^\circ$ ,  $n = 35$

Fig. 8.8. Motion blur results.

## 8.5 Comparison with Photoshop

Adobe Photoshop 7.0 provides a motion blur function that produces results similar to those produced by the motion blur algorithm described in Section 8.2. Figs. 8.9, 8.10, and 8.11 compare motion blur results from this algorithm with those produced by Photoshop. Fig. 8.9 shows results for  $\alpha = 30^\circ$  and  $n = 25$ . The corresponding Photoshop parameters are Angle =  $30^\circ$  and Distance = 25 pixels. Fig. 8.10 shows results for  $\alpha = 0^\circ$  and  $n = 35$ . The corresponding Photoshop parameters are Angle =  $0^\circ$  and Distance = 35 pixels. Fig. 8.11 shows results for  $\alpha = 135^\circ$  and  $n = 21$ . The Photoshop parameters are Angle =  $135^\circ$  and Distance = 21 pixels.



(a) Motion Blur Algorithm



(b) Photoshop

Fig. 8.9. Comparison of motion blur results with Photoshop 7.0 for  $\alpha = 30^\circ$  and  $n = 25$ .



(a) Motion Blur Algorithm



(b) Photoshop

Fig. 8.10. Comparison of motion blur results with Photoshop 7.0 for  $\alpha = 0^\circ$  and  $n = 35$ .



(a) Motion Blur Algorithm



(b) Photoshop

Fig. 8.11. Comparison of motion blur results with Photoshop 7.0 for  $\alpha = 135^\circ$  and  $n = 21$ .

## 8.6 Implementation for Subsampled YCbCr and YCrCb

The current implementation supports RGB, YCbCr, and YCrCb images, including subsampled color formats such as YCbCr 4:2:2, YCbCr 4:2:0, and YCrCb 4:2:0. For intermediate rotations required by the motion blur algorithm, RGB images can be processed by directly applying rotation equations from Chapter 12. However, images in a subsampled YCbCr or YCrCb color format cause problems due to their subsampled chrominance components, Cb and Cr.

The method described in Chapter 2 can be used to handle subsampled YCbCr and YCrCb color formats by using an intermediate YCbCr 4:4:4 or YCrCb 4:4:4 image. There is no visible difference between a subsampled YCbCr or YCrCb image blurred in this manner and an equivalent RGB motion blur image. For comparison, Fig. 8.12 (a), (b), (c), and (d) give motion blur results from RGB, YCbCr 4:2:2, YCbCr 4:2:0, and YCrCb 4:2:0 inputs. All four images were obtained with  $\alpha = 60^\circ$  and  $n = 15$ .



Fig. 8.12. Comparison of motion blur results for 4 different color formats with  $\alpha = 60^\circ$  and  $n = 15$ .

## Chapter 9

# Multi-directional Blur: Gaussian and Uniform Blur

### 9.1 Introduction

Multi-directional blur can be used to soften an image without introducing directional bias. Two of the most widely used multi-directional blurs are Gaussian blur and uniform blur. Both require a square filter mask, also known as a convolution kernel. Gaussian and uniform blur help smooth an image through a spatial-domain filter operation. Two-dimensional convolution with a Gaussian or uniform filter mask is equivalent to a low-pass filter operation in the spectral domain.



(a) Original

(b) Photoshop Result

Fig. 9.1. Gaussian Blur from Photoshop 7.0. Image size is 520 x 390. Blur size is 7.0 pixels.

Fig. 9.1 (a) shows an original 520 x 390 image. Fig. 9.1 (b) shows a Gaussian Blur result from Adobe Photoshop 7.0 for a blur size of 7.0 pixels. Photoshop supports blur sizes from 0.1 to 250.0 pixels. Uniform blur is not explicitly provided in Photoshop.

Gaussian blur uses an  $n \times n$  filter mask whose coefficients are a discrete approximation to a continuous, bell-shaped Gaussian distribution. The following are examples of 3 x 3 and 5 x 5 Gaussian filters from [35]:

$$G_3 = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \quad \text{and} \quad G_5 = \frac{1}{256} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix} \quad (9.1)$$

In a Gaussian filter mask, the largest coefficient is at the center. Coefficient values taper off symmetrically on all sides. Convolution with a Gaussian kernel results in a weighted average that emphasizes the center pixels. Notice that factors of  $1/16$  and  $1/256$  are needed to ensure that the sum of coefficients is always 1. A sum of 1 avoids incorrectly distorting pixel intensities in the blurred image.

Uniform blur uses an  $n \times n$  filter mask whose coefficients are all equal and sum to 1. The  $3 \times 3$  and  $5 \times 5$  uniform filters are

$$H_3 = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad \text{and} \quad H_5 = \frac{1}{25} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$



(a) Gaussian Blur



(b) Uniform Blur

Fig. 9.2. Comparison of Gaussian and uniform blur for  $n = 11$ .

A rule of thumb for using uniform filters to smooth out noise in an image is to set the filter length to  $n \geq 2d_N + 1$ , where  $d_N$  is the maximum diameter of objects in the image [35]. For instance, an image with speckle noise no larger than 5 pixels in diameter should be blurred with a filter length of  $n \geq 2 \cdot 5 + 1 = 11$ . Conversely, if image features with a



diameter of  $d_p$  need to be preserved, a filter length of  $n \leq 2d_p + 1$  is recommended [35]. For example, if the minimum diameter of features to be preserved is 10 pixels, a filter length of  $n \leq 2 \cdot 10 + 1 = 21$  should be used.

While uniform blur is simpler to compute than Gaussian blur, there are at least two drawbacks to the former. First, the only adjustable parameter in uniform blur is the filter length, whereas in Gaussian blur, filter coefficients can be adjusted to achieve an arbitrary roll-off effect. The relative inability to control uniform blur exacerbates the problem of undesired blurring of important features, such as edges and textures. This problem exists for all low-pass filters but is especially severe for uniform blur.

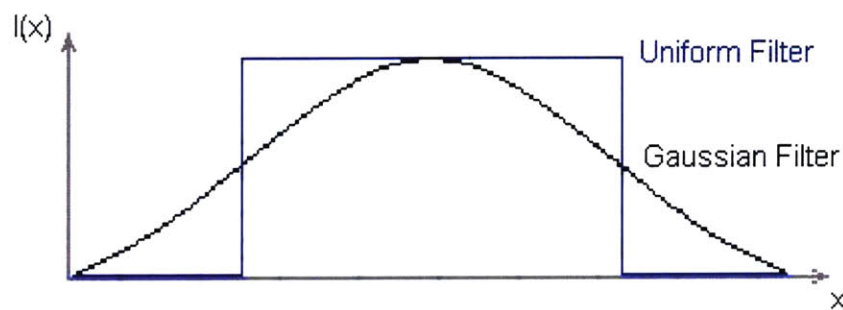


Fig. 9.3. Cross-sectional view of Gaussian and uniform filter shapes.

Second, a uniform filter introduces distortion due to ringing in the frequency domain. The tapering of Gaussian filter coefficients reduces ringing. The tradeoff is that Gaussian blur achieves less noise smoothing. In Fig. 9.2, notice that the result from uniform blur looks much more blurry than the result from Gaussian blur although both used a filter length of  $n = 11$ . The cross-sectional shapes of Gaussian and uniform filters are compared in Fig. 9.3.

## 9.2 Algorithm

The Gaussian blur and uniform blur functions use the same underlying algorithm but with different coefficients in their filter masks. Since uniform blur has equal filter coefficients, its implementation in this thesis has been optimized to avoid multiplication with filter coefficients by simply summing and averaging pixel values. Both Gaussian and uniform blur functions support arbitrarily large amounts of blurring. For a blur amount of  $n$ , the filter mask is  $n \times n$ . Since masks are symmetric and must be centered on a pixel,  $n$  must be an odd positive integer.

The blurring algorithm is essentially a convolution operation between an input image and an  $n \times n$  filter mask. Suppose the input image is  $f(x, y)$ , the output image is  $g(x, y)$ , and the filter coefficients are  $h(m, k)$ . The convolution equation is given by

$$g(x, y) = \sum_{m=-(n-1)/2}^{(n-1)/2} \sum_{k=-(n-1)/2}^{(n-1)/2} f(m, k) h(x - m, y - k)$$

Blurring is done on a pixel-by-pixel basis. The filter mask is centered on an input pixel. Each pixel in the  $n \times n$  neighborhood is multiplied by the corresponding filter coefficient. The products are summed, and the resulting weighted average is assigned as the output pixel value.

One problem that arises during implementation is the processing of pixels along the edges of an image. For these pixels, part of the filter mask will fall outside the image, where there are no pixel values to be summed. One approach is to pad the boundaries by duplicating pixels. There is a variety of pixel duplication techniques. One common technique is to “reflect” pixels across the boundary, as shown in Fig. 9.4.

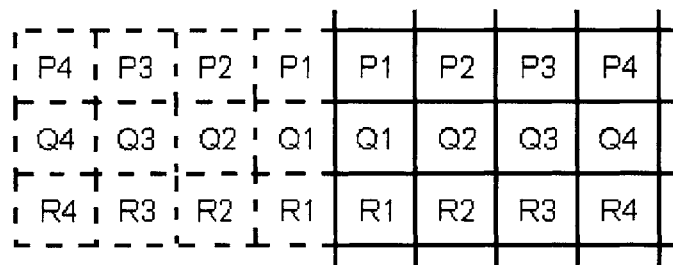


Fig. 9.4. Boundary padding for a left image boundary.

Boundary padding can be interpreted as filling in missing pixels outside an image with existing pixels inside the image. Padding allows a straightforward implementation for using a filter mask. Without padding, each pixel along a boundary would require extensive computations to calculate appropriate values for missing pixels inside the filter mask.

Although the same algorithm is used for Gaussian and uniform blur, there is one crucial difference between these two functions. The Gaussian blur function can be called repeatedly to produce output images that correspond to increasingly large Gaussian filter masks. Repeated application of a Gaussian filter mask produces blur that is still Gaussian. However, repeated application of a uniform filter mask produces blur that is no longer uniform. In fact, as the number of repeated applications of uniform blur increases, the results approach Gaussian blur.

Calling the Gaussian blur function once with a large filter mask is equivalent to calling it repeatedly with smaller filter masks. This is because a large filter mask can be calculated by convolving smaller filter masks. For instance, a  $5 \times 5$  Gaussian filter mask is equal to the convolution of two  $3 \times 3$  Gaussian filter masks. A  $7 \times 7$  filter mask is equal to the convolution of three  $3 \times 3$  filter masks or the convolution of a  $5 \times 5$  filter mask with a

3 x 3 filter mask. In general, an  $n \times n$  Gaussian filter mask  $G_n$  is obtained by convolving  $(n-1)/2$  filter masks of size 3 x 3. That is,

$$G_n = \underbrace{G_3 * G_3 * \dots * G_3}_{(n-1)/2 \text{ filter masks}}$$

where  $G_3$  is the 3 x 3 Gaussian filter mask given in equation (9.1). Fig. 9.5 compares two equivalent Gaussian blur results obtained with different filter masks. Fig. 9.5 (a) is the result of convolving an image with a 3 x 3 filter mask fifteen times. Fig. 9.5 (b) is the result of convolving the same original image with a 31 x 31 filter mask once. There is no visible difference between the two blurred images.



(a) 15 convolutions with 3 x 3 filter mask

(b) 1 convolution with 31 x 31 filter mask

Fig. 9.5. Equivalent Gaussian blur results using different filter masks.

The fact that larger Gaussian filter masks result from convolving smaller ones has been used to simplify the implementation of the Gaussian blur function and minimize the use of camera phone memory space. Instead of storing filter coefficients for all possible Gaussian filter masks, only a 3 x 3 filter mask was stored and any larger sized filter mask was generated through repeated convolutions with the 3 x 3 filter. This allows the Gaussian blur function to support any blur amount  $n$  by producing an  $n \times n$  filter mask on the fly. Note that this method only applies to Gaussian blur. Large uniform filter masks cannot be obtained by convolving smaller uniform filter masks.

### 9.3 Avoiding Integer Overflow

The functions for Gaussian and uniform blur were implemented on a 32-bit RISC integer microprocessor. To conserve memory space and speed up performance, no integers larger than 32 bits were used. For Gaussian blur, a naïve implementation, where an appropriate

filter mask is calculated and convolved once with an input image, would have caused problems with integer overflow if the blur amount  $n$  is greater than 11. To avoid overflow, repeated convolutions with smaller filter masks are used when  $n > 11$ . Instead of a single convolution with an  $n \times n$  filter mask,  $m_1$  convolutions with an  $11 \times 11$  filter mask are performed, followed by  $m_2$  convolutions with a  $3 \times 3$  filter mask. The values of  $m_1$  and  $m_2$  are calculated as follows:

$$m_1 = k/5 \quad \text{and} \quad m_2 = k \bmod 5,$$

where

$$k = (n - 1)/2.$$

The above equations use integer division, where floating-point results are truncated to integers. As explained in the previous section, multiple convolutions with smaller Gaussian filter masks are equivalent to a single convolution with a large Gaussian filter mask.

The same technique for avoiding integer overflow cannot be applied for uniform blur. Multiple convolutions with smaller uniform filter masks are not equivalent to a single convolution with a large uniform filter mask. An implementation that uses integers no more than 32 bits can only support blur amounts up to  $n = 63$  without causing overflow.

## 9.4 Results

Fig. 9.6 shows Gaussian blur results for  $3 \times 3$  and  $35 \times 35$  filter masks. The  $3 \times 3$  Gaussian blur result represents the minimum possible amount of blurring. Theoretically,  $n$  can be as small as 1, but this simply produces no blur. For comparison, Fig. 9.7 shows uniform blur results for  $3 \times 3$  and  $35 \times 35$  filter masks. Although the  $3 \times 3$  uniform and Gaussian blur results look very similar, the  $35 \times 35$  uniform blur result is much more blurry than the Gaussian blur result. This difference is expected and was explained in Section 9.1.



(a)  $n = 3$

(b)  $n = 35$

Fig. 9.6. Gaussian blur. Filter mask is  $n \times n$  pixels.



(a)  $n = 3$



(b)  $n = 35$

Fig. 9.7. Uniform blur. Filter mask is  $n \times n$  pixels.



(a) Gaussian Blur Function,  $n = 21$



(b) Photoshop 7.0, size = 2.5 pixels



(c) Gaussian Blur Function,  $n = 131$



(d) Photoshop 7.0, size = 5.0 pixels

Fig. 9.8. Gaussian blur comparison with Photoshop.

## 9.5 Comparison with Photoshop

Photoshop 7.0 offers Gaussian blur but not uniform blur. Although the Gaussian blur function in this thesis allows the size of the Gaussian filter mask to be directly specified, Photoshop uses a different parameter setting to determine the size. The Photoshop size parameter ranges from 0.1 to 250.0 pixels. Fig. 9.8 compares two pairs of results from the Gaussian blur function and Photoshop. For each pair, the Photoshop parameter was chosen to obtain the closest approximation to the result from the Gaussian blur function. Fig. 9.8 (a) and (b) were produced with  $n = 21$  in the Gaussian blur function and a size of 2.5 pixels in Photoshop. Fig. 9.8 (c) and (d) were produced with  $n = 131$  in the Gaussian blur function and a size of 5.0 pixels in Photoshop.



Fig. 9.9. Comparison of Gaussian blur results for 4 different color formats with  $n = 25$ .

## 9.6 Implementation for Subsampled YCbCr and YCrCb

The current implementation supports RGB, YCbCr, and YCrCb color formats, including subsampled formats such as YCbCr 4:2:2, YCbCr 4:2:0, and YCrCb 4:2:0. For Gaussian

and uniform blur, RGB images can be processed by directly convolving with the appropriate filter masks. However, images in a subsampled YCbCr or YCrCb color format cause problems due to their subsampled chrominance components, Cb and Cr.

The method described in Chapter 2 was used to handle subsampled YCbCr and YCrCb color formats by creating an intermediate YCbCr 4:4:4 or YCrCb 4:4:4 image. There is no visible difference between a subsampled YCbCr or YCrCb image blurred in this manner and an equivalent RGB blurred image.

For comparison, Fig. 9.9 (a), (b), (c), and (d) give Gaussian blur results produced from RGB, YCbCr 4:2:2, YCbCr 4:2:0, and YCrCb 4:2:0 inputs. All four images were obtained with a filter length of  $n = 25$ .

Fig. 9.10 provides a similar comparison of uniform blur results for different color formats. Fig. 9.10 (a), (b), (c), and (d) show results for RGB, YCbCr 4:2:2, YCbCr 4:2:0, and YCrCb 4:2:0. All four images were obtained with a filter length of  $n = 25$ .

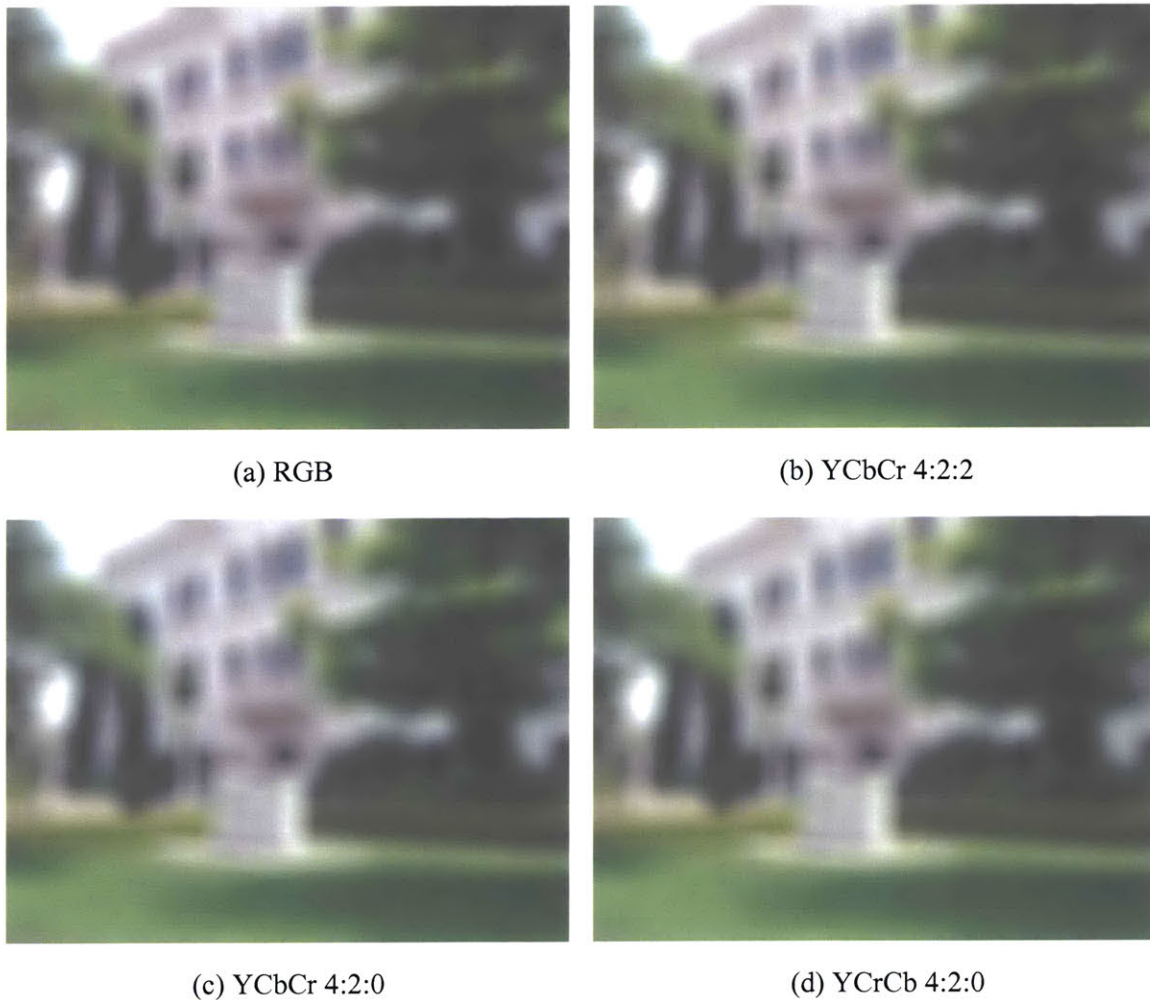


Fig. 9.10. Comparison of uniform blur results for 4 different color formats with  $n = 25$ .





# Chapter 10

## Water Reflection (Patent Pending)

### 10.1 Introduction

Adding a water reflection is a relatively popular effect in computer graphics. Art editors of magazines and other publications love to add water and submerge objects, especially if the water is reflective and looks realistic [7]. Sometimes, when a photograph is taken of a natural scene that includes a body of water, the camera frame might not be able to capture as much of the water as desired. At other times, a photographed water reflection might not look reflective enough or visually pleasing. In both situations, it would be preferable to replace the photographed water reflection with an artificial one that looks pleasing and realistic.



Fig. 10.1. The 22 steps for creating a water reflection in Adobe Photoshop 7.0 [19]. Only the first and last steps are shown for illustration purposes.

In general, water is a difficult material to simulate. Creating a realistic water reflection is labor-intensive and time-consuming, even when aided by sophisticated graphics software, such as Adobe Photoshop. No existing graphics applications can create water reflections automatically. In Photoshop 7.0, for instance, there is no standard Water Reflection function. The effect must be created manually. One Photoshop guidebook took 22 different steps to create the water reflection shown in Fig. 10.1 [19]. Another Photoshop book took 9 steps to create the water reflection shown in Fig. 10.2 [7].

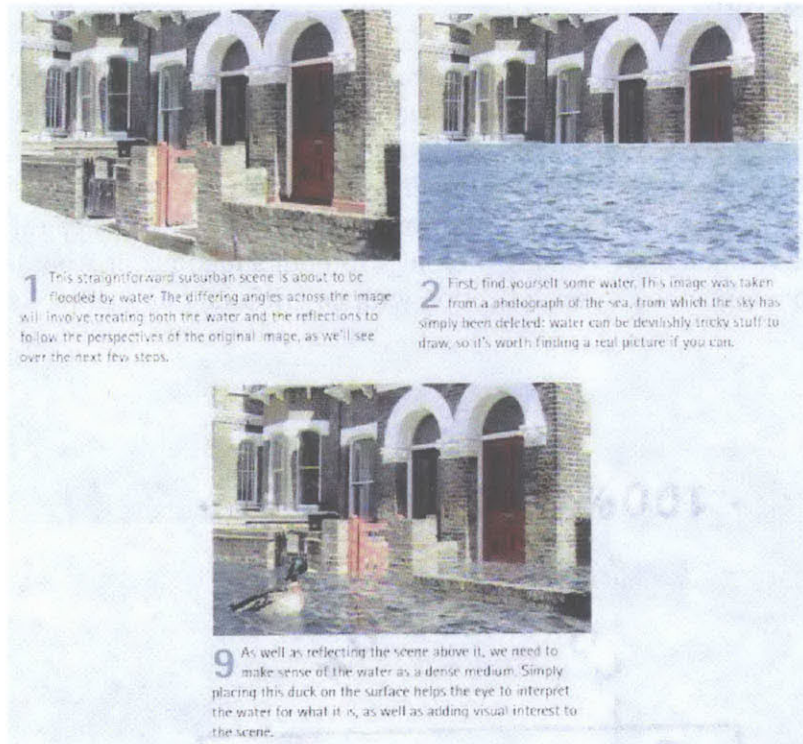


Fig. 10.2. Water reflection made in Photoshop 7.0 with a slanted water boundary [7]. Only the first two steps and the last step are shown for illustration purposes.

Both water reflections have problems. The one in Fig. 10.1 has artificial ripples that attempt to simulate the wavy perturbations of a real water surface. However, these ripples have constant size. In real photographs, water ripples change in size and exhibit perspective foreshortening. Ripples near the viewer appear larger, whereas ripples far away appear smaller. This effect is illustrated in Fig. 10.3 (c), which shows a real water reflection in windy conditions. Furthermore, the book that produced the reflection in Fig. 10.1 was only able to create water reflections with a perfectly horizontal water boundary. In real life, the edge of a body of water is not always a perfectly horizontal straight line. For instance, the water boundary in Fig. 10.3 (b) is a piecewise linear curve, whereas the boundary in Fig. 10.3 (c) is an upward sloping line. This is a significant shortcoming.

Similarly, there are problems with the reflection in Fig. 10.2. The water is barely reflective. No ripples were added to the reflection. The result is so apparently artificial that a duck had to be added to help “the eye interpret the water for what it is” [7]. Still, this water reflection does have a slanted water boundary that is more realistic than the horizontal one in Fig. 10.1.

Neither Photoshop book offers methods for creating reflections in different weather conditions. In real life, there are at least three types of weather conditions, each producing a different type of water surface: still water, water in a gentle breeze, and water in windy conditions. Fig. 10.3 shows three examples of real water reflections, one for each type of water surface.



Fig. 10.3. Real water reflections.

The main reason for the difficulty in generating realistic water reflections is that they have characteristics quite distinct from the perfect reflections seen in mirrors. The calmer the body of water, the greater its ability to reflect an image. At one extreme, reflection in a smooth lake with no currents is virtually identical to a mirror reflection and is simple to create. At the other extreme, a stormy, turbulent sea is highly unreflective, so no reflection is really needed. In between these two extremes, water reflections are more complicated. However, it is this in-between stage that accounts for most water reflections encountered in real life, whether it is a street puddle or a rippled lake surface.

The image processing library developed in this thesis provides a Water Reflection algorithm that solves the problem by automatically generating a realistic water reflection

that can vary according to the weather condition and desired calmness of the water surface. In addition, the algorithm supports water boundaries of arbitrary shape in order to more closely approximate real bodies of water, which often have irregularly shaped contours.

## 10.2 A New Algorithm

The Water Reflection algorithm developed in this thesis has four steps:

1. Set a water boundary and determine the axis of reflection.
2. Reflect the image.
3. Add ripples. (This is skipped for still water.)
4. Blend with a water image.

Note that the third step is skipped for still water. There are three varieties of water surfaces supported by the algorithm:

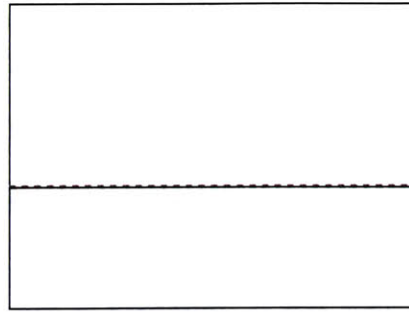
1. Still water
2. Water in a gentle breeze
3. Water in windy conditions

These varieties correspond to the three types of real water reflections shown in Fig. 10.3. The first two steps of the algorithm are exactly the same for all three varieties. The third step is applied only for water in breezy or windy conditions. Although the fourth step is applied for all varieties of water, each variety uses a different water image for blending. The four steps of the algorithm are described in the following subsections.

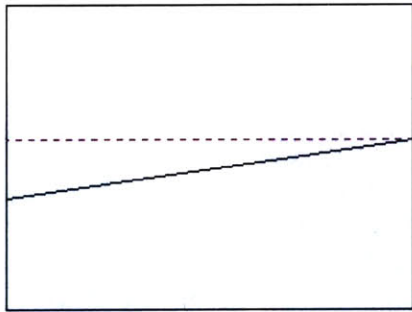
### 10.2.1 Setting a Boundary and Determining the Axis of Reflection

Creating an artificial water reflection requires setting the boundary of the reflecting body of water. The Water Reflection algorithm allows the water boundary to be placed anywhere within the original image. The boundary can be a line or curve of arbitrary shape as long as it is not vertical and can be described by analytical equations. For instance, a simple linear boundary can be specified by selecting two points on the line, as described in Section 10.2.2. Such flexibility in the shape of the boundary helps produce realistic approximations to natural bodies of water, which might have irregularly shaped contours.

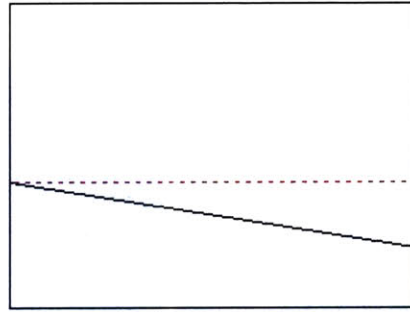
Vertical boundaries are not permitted since a water reflection is typically below, rather than to the left or right of, the original image. However, if a vertical boundary is desired, one can simply rotate the image by  $90^\circ$ , apply the algorithm, and rotate it back by  $90^\circ$ . This might be useful if, for instance, a water reflection needs to be added to a photograph taken with a camera that was rotated  $90^\circ$ . For simplicity, this algorithm focuses on creating realistic water reflections, rather than correcting the orientations of photographs.



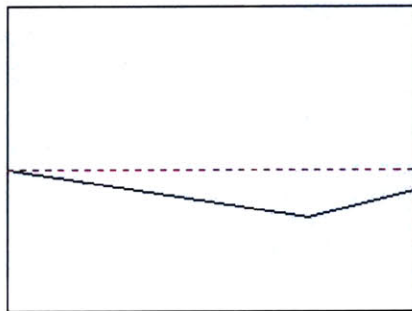
(a)



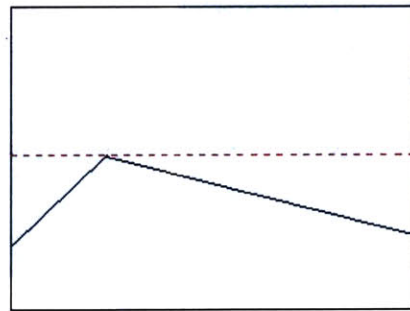
(b)



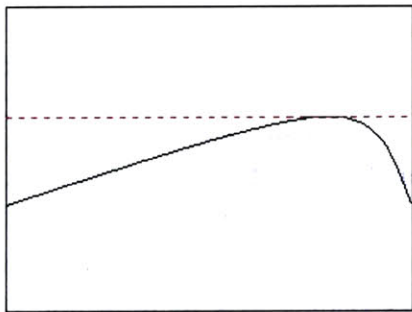
(c)



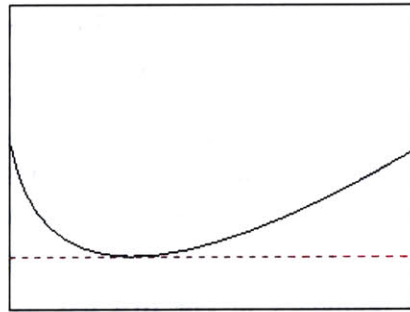
(d)



(e)



(f)



(g)

Fig. 10.4. Possible water boundaries and corresponding axes of reflection.

Fig. 10.4 shows several possible water boundaries. Fig. 10.4 (a), (b), and (c) show simple linear boundaries that are horizontal, upward sloping, and downward sloping, respectively. Fig. 10.4 (d) and (e) show piecewise linear boundaries. Fig. 10.4 (f) and (g) show curved boundaries that can be described by polynomial equations.

Once a boundary is specified, the Water Reflection algorithm automatically determines the axis of reflection, over which the original image is reflected. In real life, a water reflection always has a horizontal axis of reflection, regardless of the actual contour of the reflecting body of water. This fact is illustrated in Fig. 10.5, which shows the same three photographs of real water reflections given in Fig. 10.3, but with reflection axes marked in red. All three photographs have horizontal axes of reflection.

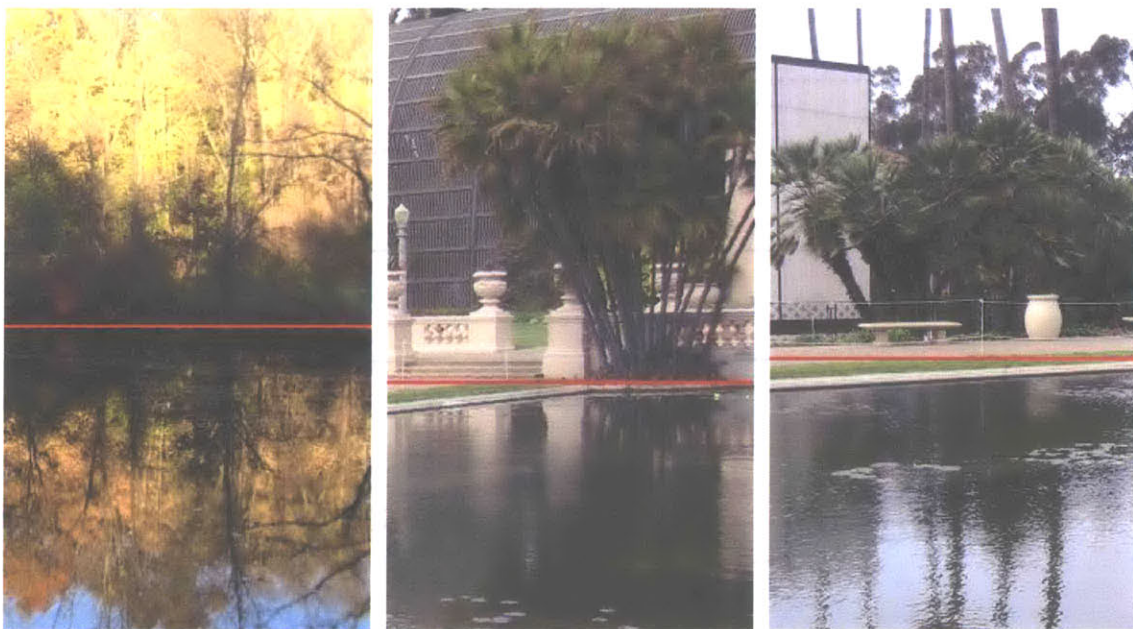


Fig. 10.5. Axes of reflection in real water reflections.

To accurately portray this real-life phenomenon, the Water Reflection algorithm automatically sets the axis of reflection as a horizontal line passing through the highest point on the water boundary. If the coordinate system is defined with an origin at the lower left corner of the image, an x axis pointing to the right, and a y axis pointing upward, as shown in Fig. 10.6, the highest boundary point has the largest y-coordinate, which is labeled  $y_{\max}$ . Theoretically, the reflection axis can be any horizontal line  $y = a$ , where  $a \geq y_{\max}$ . For simplicity, the algorithm always sets the reflection axis to  $y = y_{\max}$ . Each water boundary in Fig. 10.4 has its corresponding axis of reflection shown as a dashed line. Notice that in Fig. 10.4 (a), the axis of reflection coincides with the water boundary since the boundary itself is a horizontal line.

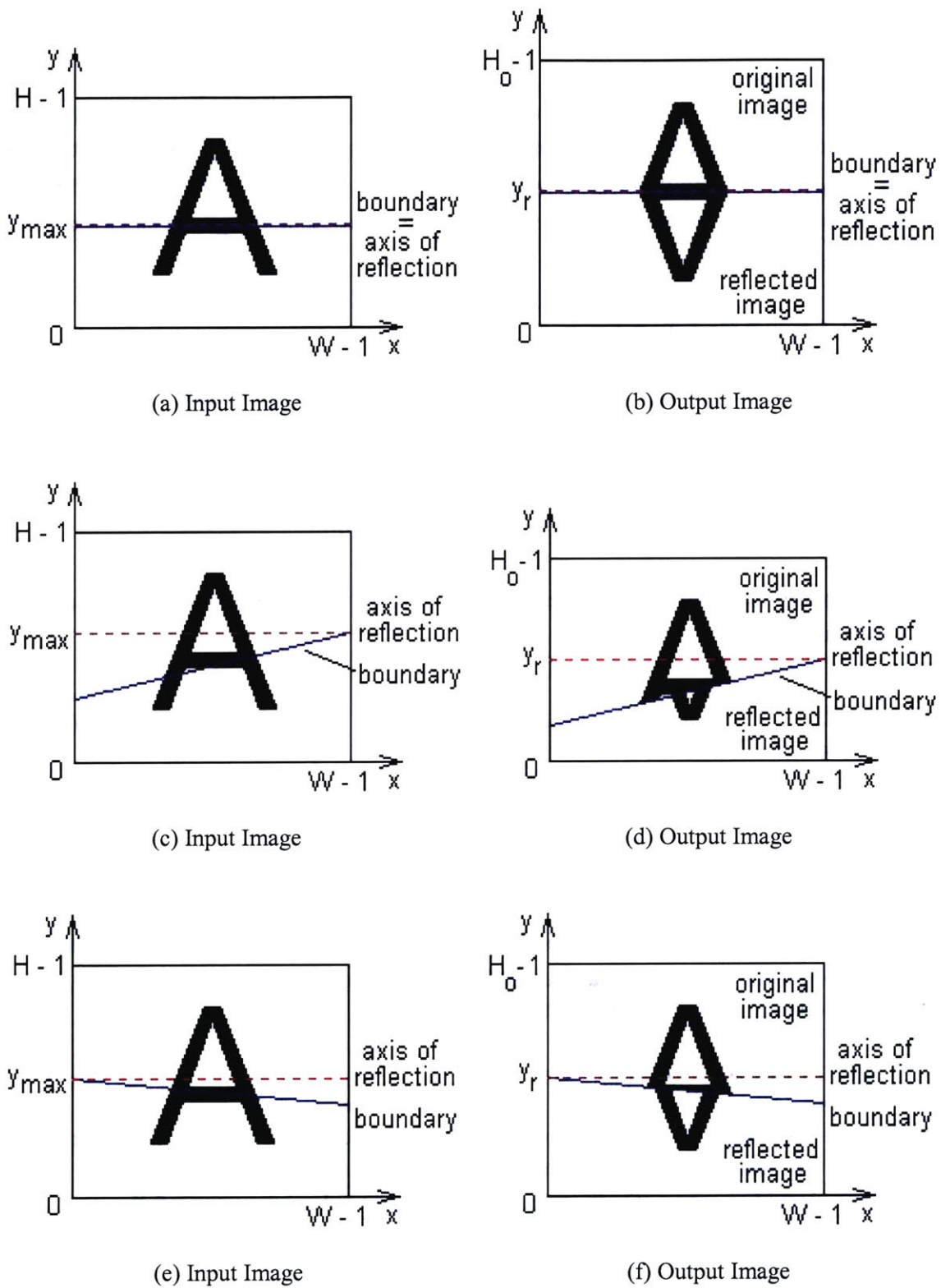


Fig. 10.6. Coordinate system for Water Reflection algorithm.

### 10.2.2 Reflecting the Image

The original input image is reflected over the axis of reflection using a standard affine transform. If the water boundary cuts off part of the reflected image, only the part below the boundary is displayed. Suppose the axis of reflection is at  $y = y_{\max}$  in the input image and at  $y = y_r$  in the output image. The difference in y-coordinates is due to the difference in the sizes and coordinate systems of the input and output images. The relationship between  $y_{\max}$  and  $y_r$  is given by

$$H - y_{\max} = H_o - y_r = y_r,$$

where  $H$  is the input image height and  $H_o$  is the output image height. The process for determining  $H_o$  is explained below. Since the axis of reflection cuts through the middle of the output image,

$$y_r = H_o/2.$$

If the boundary is specified by the equation  $y = f(x)$ , the transformation equations for reflection are

$$\begin{cases} x_{out} = x_{in} \\ y_{out} = y_r + y_{\max} - y_{in} \end{cases} \quad (10.1)$$

for

$$y_{out} \leq f(x_{out}),$$

where  $(x_{in}, y_{in})$  is the input pixel location and  $(x_{out}, y_{out})$  is the output pixel location. For example, if the boundary is a straight line, as it is in Fig. 10.4 (a), (b), and (c), it can be described by an equation in the form

$$y = f(x) = ax + b, \quad (10.2)$$

where  $a$  is the slope of the line and  $b$  is the y-intercept. To specify a simple linear boundary, the user selects two points  $(x_1, y_1)$  and  $(x_2, y_2)$  on the line. The Water Reflection algorithm automatically calculates the slope and y-intercept as follows:

$$a = \frac{y_2 - y_1}{x_2 - x_1}, \quad (10.3)$$

$$b = y_1 - ax_1. \quad (10.4)$$



Similarly, each of the piecewise linear boundaries in Fig. 10.4 (d) and (e) can be described by an equation in the form

$$y = f(x) = \begin{cases} a_1x + b_1 & \text{for } x \leq x_p \\ a_2x + b_2 & \text{for } x > x_p \end{cases} \quad (10.5)$$

where  $a_1$  and  $a_2$  are the slopes of the two linear segments,  $b_1$  and  $b_2$  are the corresponding y-intercepts, and  $x_p$  is the x-coordinate of the point where the two segments meet.

Finally, each of the curved boundaries in Fig. 10.4 (f) and (g) can be described by a polynomial equation in the form

$$y = f(x) = c_0 + c_1x + c_2x^2 + c_3x^3 + \dots, \quad (10.6)$$

where  $c_0, c_1, c_2, c_3, \dots$  are the polynomial coefficients.

As shown in Fig. 10.6, the output image contains the part of the reflected image that is below the water boundary and the part of the original image that is above the boundary. The input image is  $W \times H$  pixels, where  $W$  is the input image width. Input and output images have the same width. The output image is  $W \times H_o$  pixels, where the value of  $H_o$  is determined by the equation

$$H_o = 2(H - y_{\max}).$$

Depending on the value of  $y_{\max}$ , output height  $H_o$  is in the range  $0 < H_o \leq 2H$ . The three sets of input and output images in Fig. 10.6 correspond to the three examples in Fig. 10.4 (a), (b), and (c), which have water boundaries that are straight lines.

### 10.2.3 Adding Ripples

As mentioned previously, ripples are not added to still water. Artificial ripples are added only to reflections that simulate breezy or windy conditions. To add ripples to a reflected image, pixels are shifted horizontally by an offset  $\delta$ , which can be positive or negative and varies according to the output y-coordinate  $y_{out}$ . The value of  $\delta$  determines the amplitude and frequency of ripples. Pixels are shifted horizontally according to the equation

$$x_{out} = x_{in} - \delta, \quad (10.7)$$

where

$$\delta = \left\{ \left[ ((H-1-y_{out}) \bmod 4) \left( \left( \frac{H-1-y_{out}}{2} \bmod 3 \right) - 1 \right) \right] \cdot k \cdot (c + (H-1-y_{out})) \right\} / H. \quad (10.8)$$

$k$  and  $c$  are parameters that affect the amplitude and variation of ripples. Parameter  $k$  controls the maximum amplitude and must be nonzero. The factor  $\left( \frac{c + (H-1-y_{out})}{H} \right)$  gradually changes the amplitude of ripples to simulate perspective foreshortening. Amplitudes are small in regions far from the viewer and large in regions near the viewer. Parameter  $c$  affects how quickly the perspective foreshortening occurs. A detailed explanation of equation (10.8) is given in Section 10.3.

All operations in equation (10.8) are integer arithmetic operations. In particular, the integer divisions should truncate all floating-point results to integers. By using only basic arithmetic operations — modulo, add, subtract, multiply, and divide — equations (10.7) and (10.8) can generate ripples faster than many standard methods. Furthermore, the product of the two modulo terms,  $((H-1-y_{out}) \bmod 4)$  and  $\left( \left( \frac{H-1-y_{out}}{2} \bmod 3 \right) - 1 \right)$ , in equation (10.8) produces pseudorandom numbers using strictly integer arithmetic, as will be explained in Section 10.3.

Alternatively, a standard random number generator, such as the `rand()` function in C, can be used instead of the modulo terms. However, the resulting random number would have to be normalized to a desired range centered on zero and multiplied by  $\left( \frac{c + (H-1-y_{out})}{H} \right)$  to generate perspective foreshortening. Regardless of how pseudorandomness is achieved, the Water Reflection algorithm is flexible enough to be used in both integer and floating-point implementations.

To simulate water in a gentle breeze, motion blur is also added to the rippled reflection image. No motion blur is added when creating reflections in still water or windy conditions. Motion blur helps smudge ripples and make the water appear less perturbed. Like the changing ripple amplitudes, the length of the motion blur filter also increases from small to large down the reflected image. This effect mimics the real water reflection in Fig. 10.1 (b), in which water near the viewer appears gently rippled and blurry, but water far away appears calm and clear.

The motion blur filter has a length  $l$  that varies from 1 to  $l_{\max}$  according to the equation

$$l = \begin{cases} 1 & \text{if } m \leq 1 \\ (m/2) \cdot 2 + 1 & \text{otherwise} \end{cases} \quad (10.9)$$

where

$$m = \lceil l_{\max} \cdot (H - 1 - y_{out}) \rceil / H .$$

$l_{\max}$  is the maximum filter length. Notice that  $m$  varies with  $y_{out}$  and therefore filter length  $l$  is also a function of  $y_{out}$ . As in equation (10.8), all arithmetic operations in equation (10.9) use integer arithmetic, where floating-point results are truncated to integers. This ensures that  $(m/2) \cdot 2$  is always even and therefore  $l$  is always odd. As explained in Chapter 8, the length of a motion blur filter must always be odd. When  $l = 1$ , no blurring occurs.



(a) Still Water



(b) Gentle Breeze



(c) Windy Conditions

Fig. 10.7. Reflection images before blending with water.

Fig. 10.7 shows reflection images obtained at the end of this step of the Water Reflection algorithm. Fig. 10.7 (a) represents still water. It is only a reflection of the original image, with no ripples or motion blur added. Fig. 10.7 (b) represents water in a gentle breeze. It is a reflection image with both ripples and motion blur added. Ripples were created using equation (10.8) with parameters  $k = 3/2$  and  $c = H/8$ . Horizontal motion blur was added with equation (10.9) using  $l_{\max} = 19$ . Fig. 10.7 (c) represents water in windy conditions. It

is a reflection image where only ripples were added using equation (10.8) with parameters  $k = 3/2$  and  $c = H/4$ . No motion blur was added Fig. 10.7 (c).

#### 10.2.4 Blending with Water

As a finishing touch, the reflected and possibly rippled image from the previous step is blended with a pre-stored water image to obtain realistic hues. A different water image is used for each of the three types of water surfaces. Fig. 10.8 shows three sample water images.

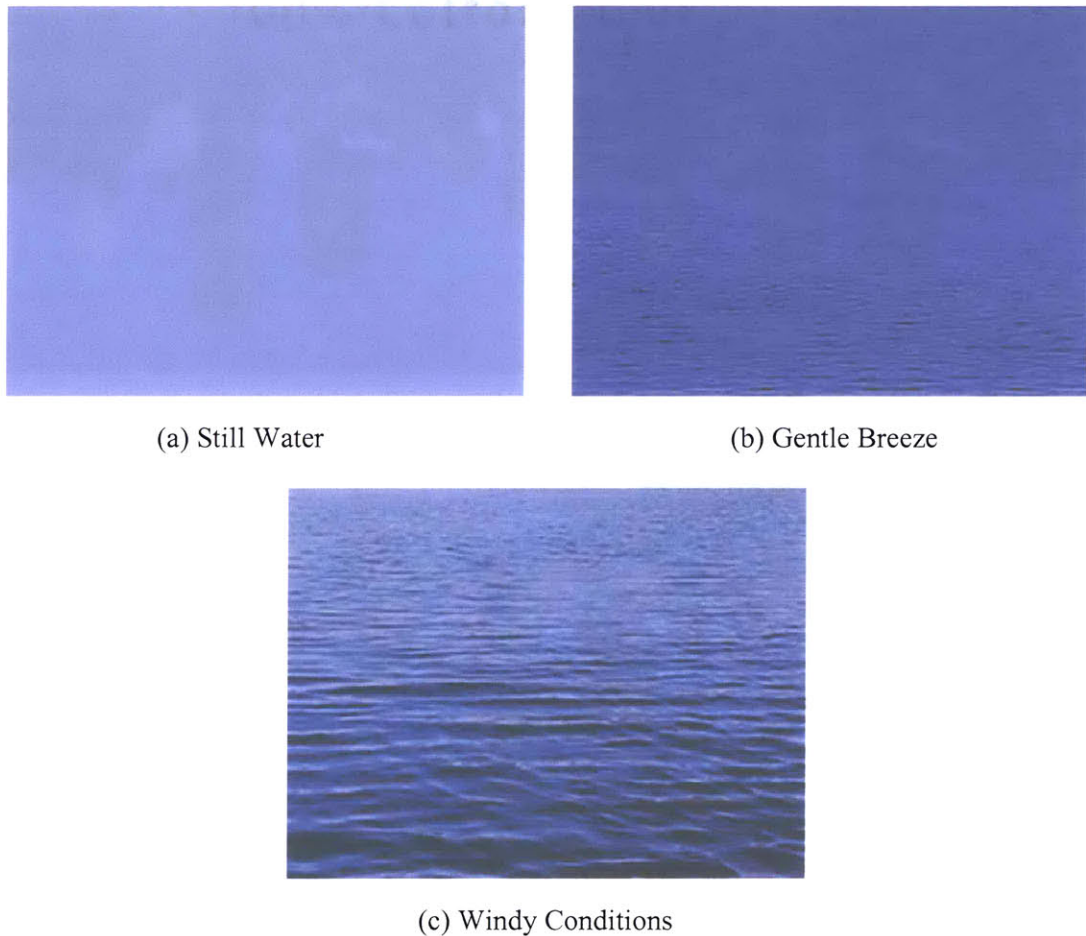
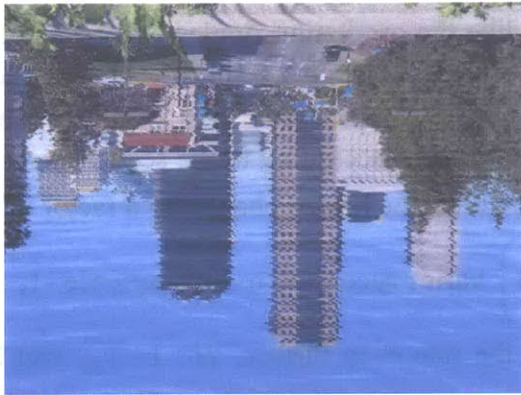


Fig. 10.8. Sample water images used for blending.

A standard alpha blending operation is used to blend the reflection image with a water image. Suppose  $\alpha$  is the alpha blending factor,  $f_1(x, y)$  is the reflected image,  $f_2(x, y)$  is the water image, and  $g(x, y)$  is the blended result. The transformation equation for alpha blending is given by

$$g(x, y) = \alpha \cdot f_1(x, y) + (1 - \alpha) \cdot f_2(x, y) \quad (10.10)$$

The value of  $\alpha$  can range from 0 to 1. The larger  $\alpha$  is, the less apparent the water image will be in the blended reflection. Fig. 10.9 shows the effect of varying  $\alpha$  for water reflections in windy conditions. In Fig. 10.9 (a), (b), and (c),  $\alpha$  equals 70%, 50%, and 30%, respectively.



(a)  $\alpha = 70\%$



(b)  $\alpha = 50\%$



(c)  $\alpha = 30\%$

Fig. 10.9. Effects of varying  $\alpha$  for water reflections in windy conditions.

### 10.3 Case Studies

This section explains the physical meaning and motivation for equations (10.8) and (10.9). Case studies are presented on the effects of varying parameters  $k$  and  $c$ . The visual effects of changing these parameters are illustrated with sample images. All images are 70 x 390 pixels.

First, let us explore the meaning of the two modulo terms in equation (10.8). The first term  $((H - 1 - y_{out}) \bmod 4)$  varies from 0 to 3, depending on the value of  $y_{out}$ . This modulo operation produces a periodic variation in the value of  $\delta$  and the amplitude of ripples. In the second modulo term,  $\left(\frac{H - 1 - y_{out}}{2} \bmod 3\right)$  varies from 0 to 2, so the entire term  $\left(\left(\frac{H - 1 - y_{out}}{2} \bmod 3\right) - 1\right)$  varies from -1 to 1. This term changes the sign of  $\delta$  so that pixels are shifted back and forth like the oscillating ripples of a real water surface. Here,  $H - 1 - y_{out}$  is divided by 2 to ensure that the change in direction of pixel shifts is not too rapid. Without dividing by 2, the second modulo term would become  $((H - 1 - y_{out}) \bmod 3) - 1$  and its value would change according to the sequence -1, 0, 1, -1, 0, 1, ... . Dividing  $H - 1 - y_{out}$  by 2 slows down the rate of change by a factor of 2, so that the value of  $\left(\left(\frac{H - 1 - y_{out}}{2} \bmod 3\right) - 1\right)$  varies according to the sequence -1, -1, 0, 0, 1, 1, -1, -1, 0, 0, 1, 1, ... . To decrease the rate even further,  $H - 1 - y_{out}$  can be divided by a larger integer, such as 3 or 4.

Together, the first two modulo terms vary the magnitude and direction of pixel shifts. The product of these terms varies between -3 and 3 in a pseudorandom way that simulates a realistic ripple effect. Pseudorandomness is achieved by the asynchronous combination of modulo 3 and modulo 4. If the same modulo number had been used in both terms, the pixel shifts would have synchronized changes in direction and magnitude and would appear distinctly non-random. 3 and 4 are simply two small numbers chosen as modulo factors to produce an adequate ripple effect. Theoretically, any pair of two different numbers can be used. For a higher degree of randomness, any number of extra modulo factors, such as  $((H - 1 - y_{out}) \bmod 5)$ , can be included in equation (10.8) as long as the overall combination of modulo terms remains asynchronous.

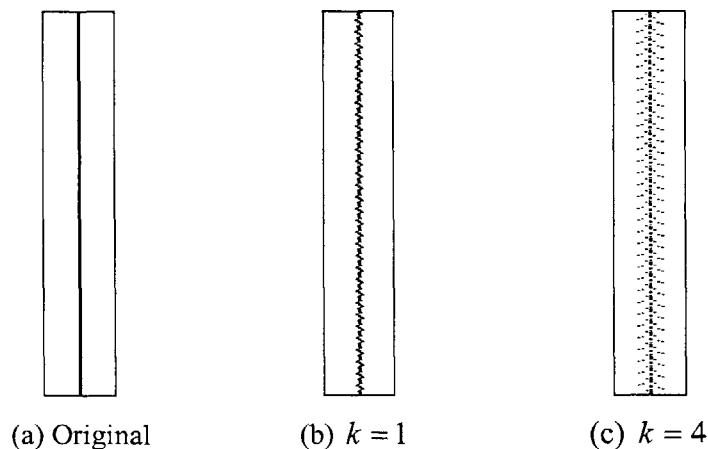


Fig. 10.10. Effects of varying  $k$  in equation (10.11) with no perspective foreshortening.

Next, we focus on the meaning of  $k$  by temporarily dropping the  $\left(\frac{c+(H-1-y_{out})}{H}\right)$  factor from equation (10.8) to obtain

$$\delta = \left[ ((H-1-y_{out}) \bmod 4) \left( \left( \left( \frac{H-1-y_{out}}{2} \bmod 3 \right) - 1 \right) \right) \right] \cdot k \quad (10.11)$$

The value of  $\delta$  in equation (10.11) varies between  $-3k$  and  $3k$ . Parameter  $k$  can be any nonzero real number in the range  $-\infty < k < \infty$ . To illustrate the effects of varying  $k$ , equations (10.7) and (10.11) are now applied on a 70 x 390 test image consisting of a black strip that is 3 pixels wide, as shown in Fig. 10.10 (a). When  $k=1$ ,  $\delta$  varies between -3 and 3, as shown in Fig. 10.10 (b). The maximum amplitude of ripples is 3. When  $k=4$ ,  $\delta$  varies between -12 and 12, as shown in Fig. 10.10 (c). The maximum amplitude of ripples is 12. As  $k$  increases, the ripple amplitude increases proportionately.

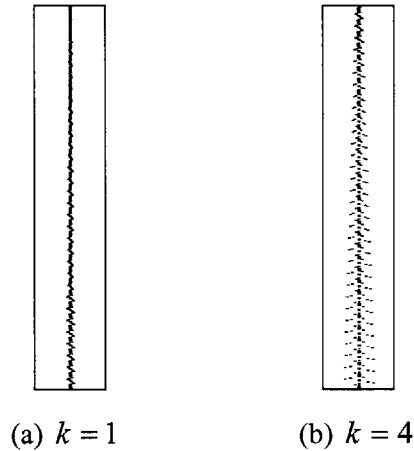


Fig. 10.11. Effects of varying  $k$  in equation (10.8) with  $c = H/4$ .

TABLE 10.1: Selected Values of  $\delta$  for  $c = H/4$ .

$y_{out}$	$\delta$ for $k=1$	$\delta$ for $k=4$
0	1	4
6	3	14
10	-3	-14
389	0	0

Now, the  $\left(\frac{c+(H-1-y_{out})}{H}\right)$  factor can be reinserted to explore the meaning of  $c$ . Equations (10.7) and (10.8) are applied with different values of  $k$  while holding  $c$

constant at  $H/4$ . Fig. 10.11 shows results for  $k=1$  and  $k=4$ . The reinserted factor enables the amplitude of ripples to change gradually from small to large down the image. Assuming that the bottom edge of the image is closer to the viewer, this factor achieves a realistic simulation of perspective foreshortening by creating larger ripples near the viewer. Thus,  $k$  controls the maximum amplitude of ripples, whereas  $\left(\frac{c+(H-1-y_{out})}{H}\right)$  makes the amplitude dependent on the distance from the viewer.

Without  $\left(\frac{c+(H-1-y_{out})}{H}\right)$ , ripples would have constant amplitude.

TABLE 10.1 shows selected values of  $\delta$  for different values of  $y_{out}$  when  $c = H/4$ . Two of the  $y_{out}$  values were chosen to represent the minimum and maximum values of  $y_{out}$ . The other two values were chosen to obtain the most positive and most negative  $\delta$  values. When  $k = 1$ , the maximum magnitude of  $\delta$  is 3. This is also the maximum ripple amplitude. When  $k = 4$ , the maximum magnitude is 14.

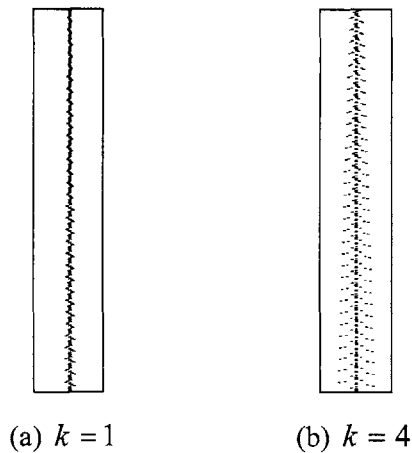


Fig. 10.12. Effects of varying  $k$  in equation (10.8) with  $c = H/2$ .

TABLE 10.2: Selected Values of  $\delta$  for  $c = H/2$ .

$y_{out}$	$\delta$ for $k = 1$	$\delta$ for $k = 4$
0	1	5
6	4	17
10	-4	-17
389	0	0



Fig. 10.12 shows results for  $k = 1$  and  $k = 4$  when  $c = H/2$ . Compared to Fig. 10.11, the perspective foreshortening effect is less severe. TABLE 10.2 shows selected values of  $y_{out}$  for  $c = H/2$ . As in TABLE 10.1, two of the  $y_{out}$  values are the minimum and maximum values, while the other two produce the most positive and most negative  $\delta$  values. When  $k = 1$ , the maximum magnitude of  $\delta$  is 4. When  $k = 4$ , the maximum magnitude is 17. Thus, a larger value of  $c$  produces a lesser degree of perspective foreshortening and a larger maximum ripple amplitude.

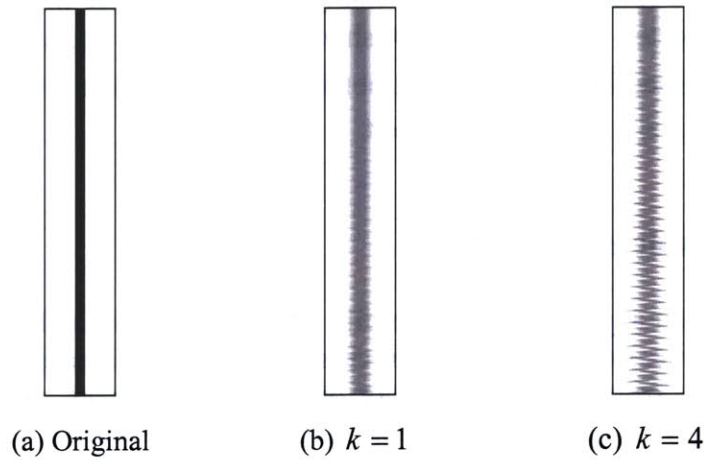


Fig. 10.13. Adding horizontal motion blur with a constant filter length of 19, while varying  $k$  in equation (10.8) with  $c = H/8$ .

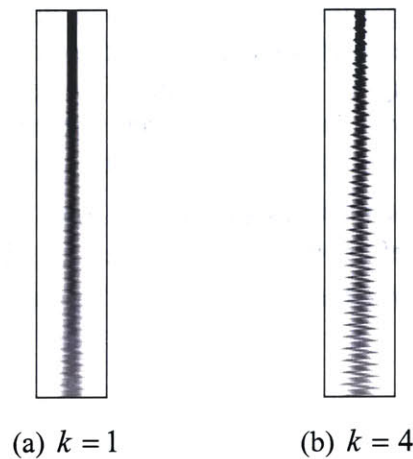


Fig. 10.14. Adding horizontal motion blur with a variable filter length given by equation (10.9) with  $l_{max} = 19$ , while varying  $k$  in equation (10.8) with  $c = H/8$ .

Finally, let us apply motion blur to rippled images obtained from equation (10.8) to show how blurring helps soften ripples simulating water in a gentle breeze. The original test image is 70 x 390 pixels and consists of a black strip that is 10 pixels wide, as shown in Fig. 10.13 (a). Fig. 10.13 (b) and (c) show the results of applying horizontal motion blur with a constant filter length of 19 to rippled images created with equation (10.8) using  $c = H/8$ . Fig. 10.13 (b) is for  $k = 1$  and (c) is for  $k = 4$ .

Using fixed-size motion blur does not produce an adequately realistic water reflection. The real water reflection in Fig. 10.3 (b) shows that water near the viewer should appear blurry and gently rippled, whereas water far away should appear calm and clear. To fix this, equation (10.9) is used to gradually change the motion blur filter length from 1 to  $l_{\max}$  down the reflected image. Fig. 10.14 shows results from applying equation (10.9) with  $l_{\max} = 19$ . Rippled images were first obtained from equation (10.8) using  $c = H/8$ . Fig. 10.14 (a) is for  $k = 1$  and (b) is for  $k = 4$ .

## 10.4 Integer Implementation

The Water Reflection algorithm can be readily implemented with either floating-point or fixed-point arithmetic. On integer microprocessors, floating-point arithmetic requires the use of additional floating-point emulators or software libraries, which can lead to unacceptable latency. Most camera phones have integer microprocessors and require a purely fixed-point, or integer, implementation to achieve real-time performance.

This algorithm was implemented on an actual cell phone chip with an integer microprocessor. Several steps in the algorithm had to be specially handled for a purely integer implementation.

In the first two steps of the algorithm, if the water boundary is a straight line, equations (10.2), (10.3), and (10.4) are used to determine the boundary line. However, these equations all require calculating the slope  $a$ , which is often a floating-point number less than 1. To maintain accuracy in a purely integer implementation, the slope must be converted to a fixed-point format by multiplying with a suitably large integer factor  $f$ , such as  $f = 2^{16} = 65536$ . When using equation (10.3) to find the slope, the numerator must be multiplied by  $f$  before it can be divided by the denominator. Otherwise, accuracy will be lost during intermediate arithmetic operations. The necessary order of operations is

$$a = [(y_2 - y_1) \cdot f] / (x_2 - x_1). \quad (10.12)$$

Equation (10.12) produces a fixed-point value for the slope. All intermediate arithmetic calculations involving the slope must use the fixed-point value. After all calculations are complete, the final result is obtained by dividing by the integer factor  $f$ .

In the third step of the algorithm, equation (10.8) was implemented with  $k = 3/2$  and either  $c = H/8$  for water in a gentle breeze or  $c = H/4$  for water in windy conditions. To achieve a purely integer implementation without loss of accuracy during intermediate arithmetic operations, the order of operations in equation (10.8), with values for parameters  $k$  and  $c$  already substituted, was rearranged to become

$$\delta = \left\{ \left[ (y_{out} \bmod 4) \left( \left( \left( \frac{y_{out}}{2} \right) \bmod 3 \right) - 1 \right) \right] \cdot 3 \cdot \left( \left( \frac{H}{n} \right) + y_{out} \right) \right\} / (2H), \quad (10.13)$$

Here,  $n$  is set to 4 or 8, depending on the type of water. When  $n = 4$ , the division operation  $H/n = H/4$  can be replaced by a fast bit-shift operation  $H \gg 2$ . Similarly, when  $n = 8$ , the division operation  $H/n = H/8$  can be replaced by  $H \gg 3$ . Notice that the 2 in  $k = 3/2$  has been grouped with  $H$  at the end of equation (10.13). This order of operations is extremely important for integer implementations. Any change in the order will result in a loss of accuracy and produce incorrect ripple effects.

Finally, in the fourth step of the algorithm, equation (10.10), which is repeated below for convenience, presents a problem for integer implementation since the alpha blending factor  $\alpha$  is assumed to be a floating-point number in the range  $0 \leq \alpha \leq 1$ .

$$g(x, y) = \alpha \cdot f_1(x, y) + (1 - \alpha) \cdot f_2(x, y) \quad (10.10)$$

One solution is to convert  $\alpha$  to a fixed-point format by multiplying it with a suitably large integer factor  $f$ , such as  $f = 2^7 = 128$ , and rounding to the nearest integer. For instance, if  $\alpha = 70\% = 0.7$ , it can be converted to  $0.7 \times 2^7 = 89.6 \cong 90$ . All intermediate arithmetic operations are performed using the new fixed-point representation for  $\alpha$ . The final result is obtained by dividing by  $f$ .

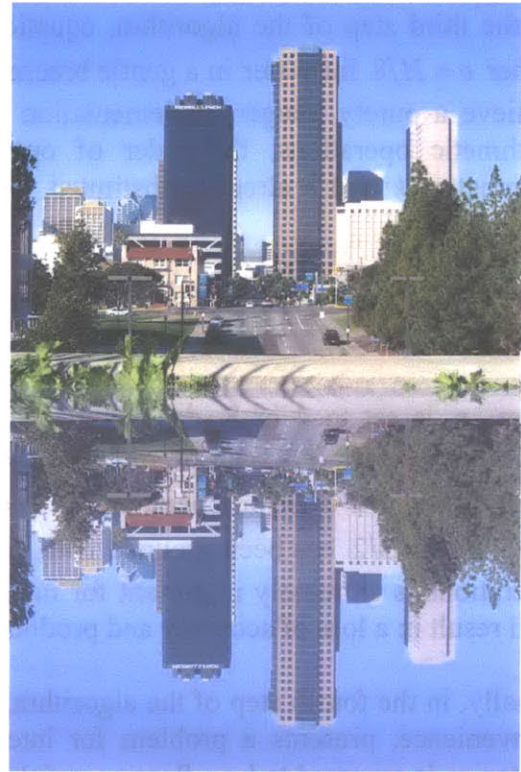
## 10.5 Results

Results from the integer implementation described in the previous section are shown in Figs. 10.15, 10.16, and 10.17. All original images are 520 x 390 pixels. Ripples were generated using equation (10.13), where  $k = 3/2$  and either  $c = H/8$  (i.e.  $n = 8$ ) for water in a gentle breeze or  $c = H/4$  (i.e.  $n = 4$ ) for water in windy conditions. Alpha blending was performed using equation (10.10) with  $\alpha = 70\%$ . All reflections that simulate a gentle breeze have motion blur added with equation (10.9) using  $l_{\max} = 19$ .

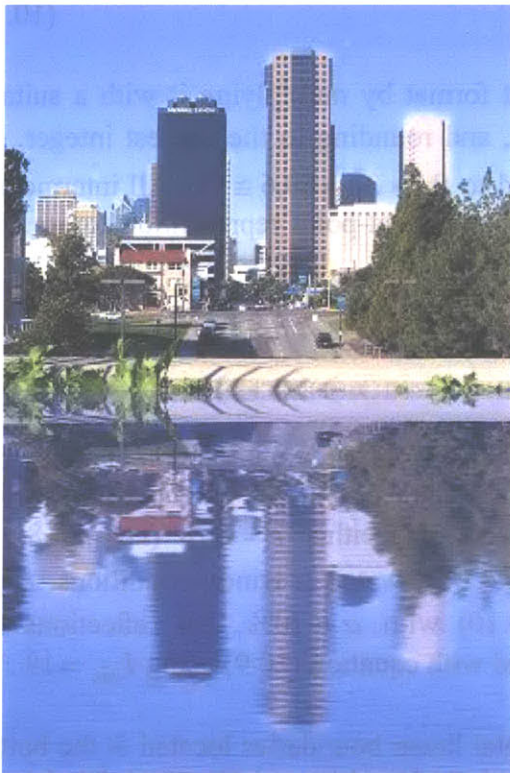
Fig. 10.15 shows water reflections with horizontal linear boundaries located at the bottom edge of the original image. Fig. 10.15 (a) gives the original image. Fig. 10.15 (b), (c), and (d) give results for still water, gentle breeze, and windy conditions, respectively. All three results are 520 x 780 pixels.



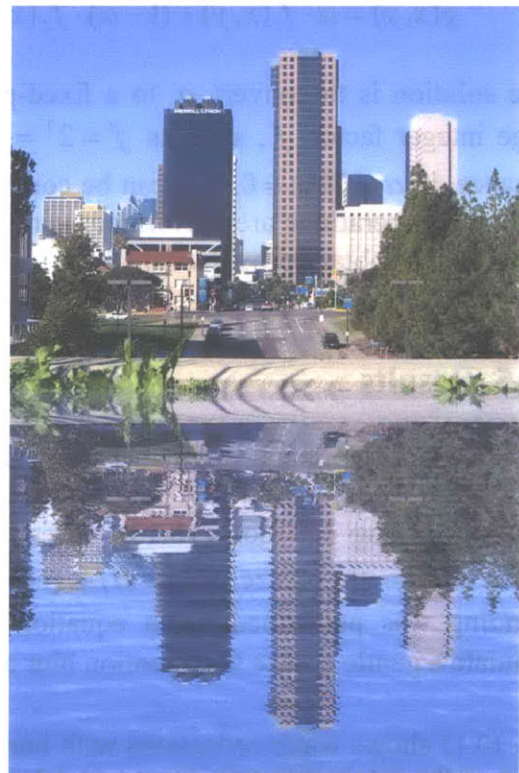
(a) Original Image



(b) Water Reflection in Still Water



(c) Water Reflection in Gentle Breeze



(d) Water Reflection in Windy Conditions

Fig. 10.15. Water reflections with horizontal linear boundaries.



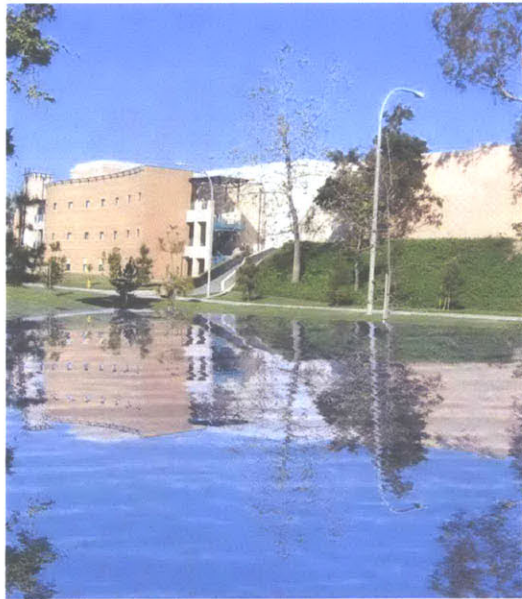
(a) Original Image



(b) Water Reflection in Still Water



(c) Water Reflection in Gentle Breeze

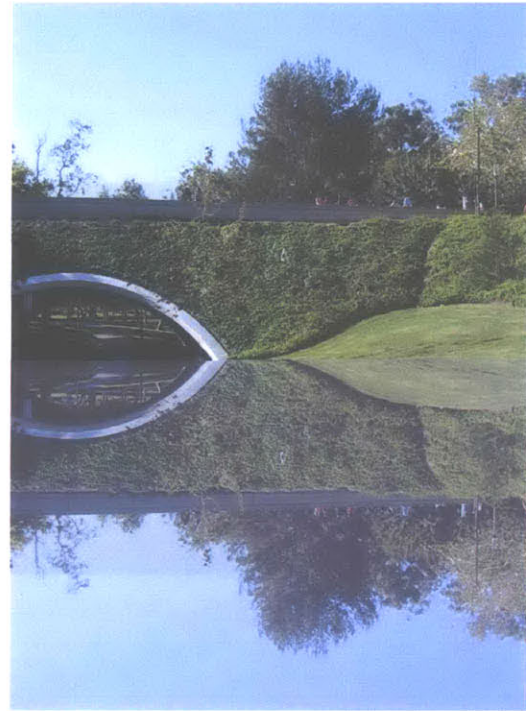


(d) Water Reflection in Windy Conditions

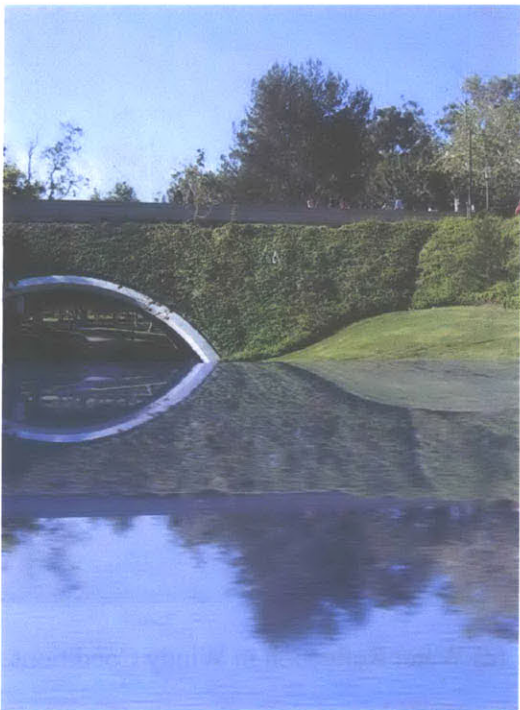
Fig. 10.16. Water reflections with downward sloping linear boundaries.



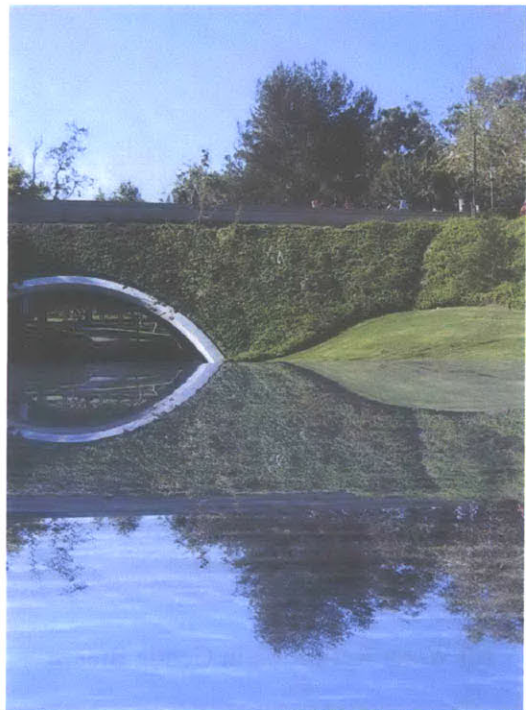
(a) Original Image



(b) Water Reflection in Still Water



(c) Water Reflection in Gentle Breeze



(d) Water Reflection in Windy Conditions

Fig. 10.17. Water reflections with upward sloping linear boundaries.

Fig. 10.16 shows water reflections with downward sloping linear boundaries. The boundary endpoints are  $(x_1, y_1) = (0, 93)$  and  $(x_2, y_2) = (519, 64)$ . Fig. 10.16 (a) gives the original image. Fig. 10.16 (b), (c), and (d) give results for still water, gentle breeze, and windy conditions, respectively. All three results are 520 x 594 pixels.

Fig. 10.17 shows water reflections with upward sloping linear boundaries. The boundary endpoints are  $(x_1, y_1) = (0, 31)$  and  $(x_2, y_2) = (519, 36)$ . Fig. 10.17 (a) gives the original image. Fig. 10.17 (b), (c), and (d) give results for still water, gentle breeze, and windy conditions, respectively. All three results are 520 x 708 pixels.

## 10.6 Implementation for Subsampled YCbCr and YCrCb

The current implementation for Water Reflection supports RGB, YCbCr, and YCrCb color formats, including subsampled formats such as YCbCr 4:2:2, YCbCr 4:2:0, and YCrCb 4:2:0. RGB images are processed by directly applying the Water Reflection algorithm. However, images in a subsampled YCbCr or YCrCb color format cause problems due to their subsampled chrominance components, Cb and Cr.

One common form of subsampled YCbCr is YCbCr 4:2:2, in which the Cb and Cr components of either pixel rows or pixel columns are subsampled by a factor of two. Two formats are H2V1 YCbCr 4:2:2, in which pixel columns are subsampled but pixel rows are unaffected, and H1V2 YCbCr 4:2:2, in which pixel rows are subsampled but pixel columns are unaffected. Fig. 10.18 (a) shows the H2V1 format. For every 2 x 2 block of pixels, there are 4 Y values, 2 Cb values, and 2 Cr values, hence the name 4:2:2. If columns are numbered starting from zero, only even columns have the Cb component and only odd columns have Cr. Hereafter, YCbCr 4:2:2 refers to H2V1 YCbCr 4:2:2.

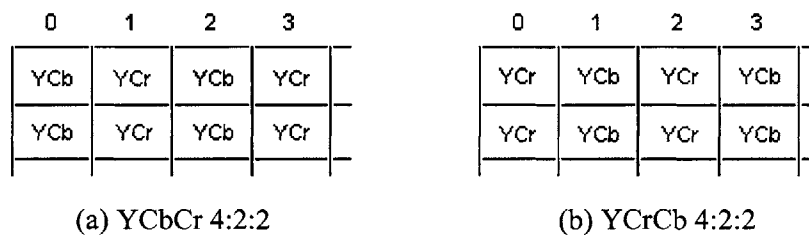
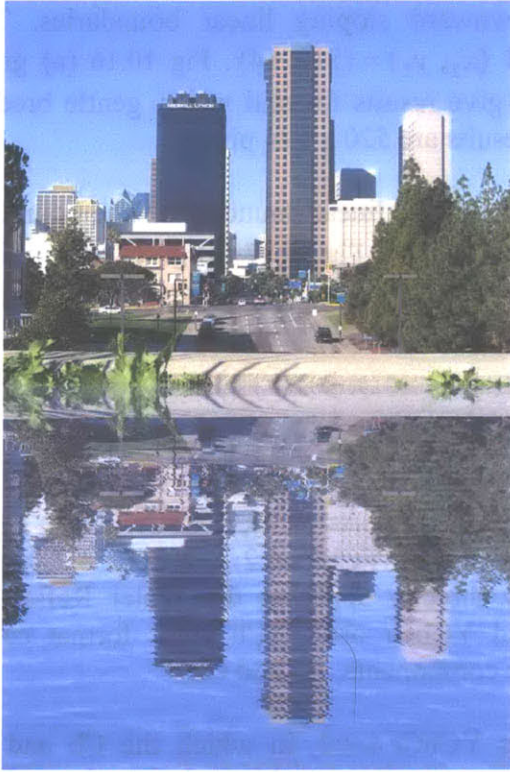


Fig. 10.18. H2V1 formats.

YCrCb formats are the same as YCbCr, but with the order of Cb and Cr components switched. For instance, an H2V1 YCrCb 4:2:2 image has the pixel arrangement shown in Fig. 10.18 (b). Hereafter, YCrCb 4:2:2 refers to H2V1 YCrCb 4:2:2.



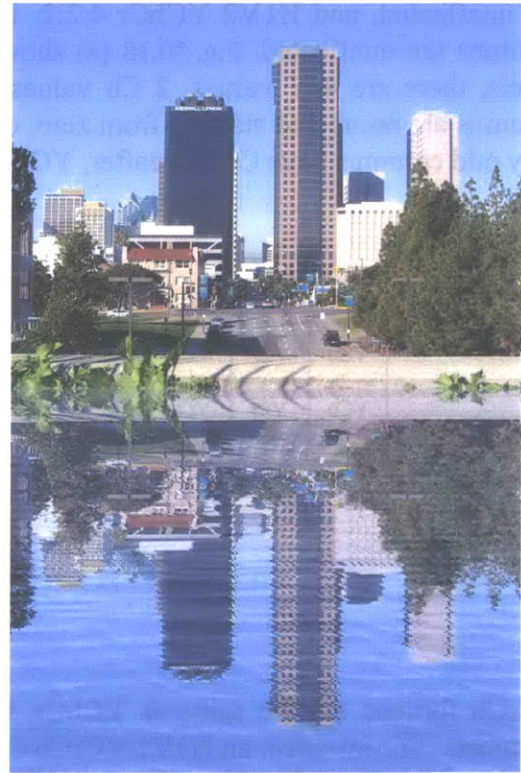
(a) RGB



(b) YCbCr 4:2:2



(c) YCbCr 4:2:0



(d) YCrCb 4:2:0

Fig. 10.19. Comparison of water reflections for 4 different color formats.



Directly applying the Water Reflection algorithm on a YCbCr 4:2:2 image will fail to guarantee correctly alternating YCb-YCr pixels in the resulting image. To solve this problem, a temporary YCbCr 4:4:4 image is created from the original subsampled YCbCr 4:2:2 image. The algorithm is applied to the YCbCr 4:4:4 image to obtain a temporary YCbCr 4:4:4 output, which is then subsampled to produce the final YCbCr 4:2:2 output.

The temporary YCbCr 4:4:4 image contains all three YCbCr components for each pixel. It is created by taking pairs of adjacent YCb and YCr pixels and having each pixel borrow the missing chrominance component from its partner. The YCb pixel borrows a Cr component from its paired YCr pixel. The YCr pixel borrows a Cb component from the YCb pixel. Normally, this method does not produce an accurate, visually pleasing YCbCr 4:4:4 image from a YCbCr 4:2:2 image. However, since the temporary YCbCr 4:4:4 image is used only for intermediate processing, the accuracy of the intermediate result does not matter as long as the final output is correct.

For the purpose of applying the Water Reflection algorithm, this method of creating a YCbCr 4:4:4 image and doing all processing in the YCbCr 4:4:4 domain works very well. The final YCbCr 4:2:2 output is obtained from the temporary YCbCr 4:4:4 output by discarding the extra Cb or Cr component in each pixel. There is no visible difference between a YCbCr 4:2:2 water reflection produced in this manner and an equivalent RGB water reflection. Fig. 10.19 (a) and (b) compare an RGB result produced from RGB input with the corresponding YCbCr 4:2:2 result produced from YCbCr 4:2:2 input. Both images simulate water in windy conditions and have a horizontal linear water boundary located at the bottom edge of the original image. Ripples were generated with equation (10.13), where  $k = 3/2$  and  $c = H/4$  (i.e.  $n = 4$ ).

This method of creating a temporary YCbCr 4:4:4 image can be extended to handle input images in any subsampled YCbCr or YCrCb color format. For instance, another common format is YCbCr 4:2:0, in which each block of 2 x 2 pixels has only one Cb and one Cr component. A temporary YCbCr 4:4:4 image is created by duplicating the Cb and Cr components for each pixel in each 2 x 2 block. The Water Reflection algorithm is applied to the temporary YCbCr 4:4:4 image to obtain a YCbCr 4:4:4 output, which is finally subsampled to YCbCr 4:2:0 by discarding the extra chrominance components. There is no visible difference between YCbCr 4:2:0 water reflections produced in this manner and RGB water reflections produced from RGB input. This can be seen by comparing Fig. 10.19 (a) with Fig. 10.19 (c), which shows results for YCbCr 4:2:0. Similarly, Fig. 10.19 (d) gives YCrCb 4:2:0 results for comparison. All images simulate water in windy conditions and have a horizontal linear water boundary located at the bottom edge of the original image. Ripples were generated with equation (10.13), where  $k = 3/2$  and  $c = H/4$  (i.e.  $n = 4$ ).



# Chapter 11

## Alpha Blending and Related Applications

### 11.1 Introduction

Alpha blending is used in computer graphics to create the illusion of transparency. It is achieved by combining a translucent foreground image with a background image to create a blended result. In film and animation sequences, alpha blending is used to gradually fade from one scene to another.

In image processing, alpha blending is useful for image fusion. One particular application is multisensor image fusion, in which alpha blending can be applied in the wavelet domain to integrate complementary information from multisensor data and produce new images that are more suitable for human visual perception and further processing [25]. The simplest approach is to average two wavelet coefficients to obtain a hybrid coefficient. More advanced approaches can find the optimal alpha blending factor and perform a weighted average to maximize output quality.

In Adobe Photoshop 7.0, alpha blending must be done manually. Two images are blended together by creating a layer for each image, placing the layers on top of each other in a new image, and manually setting the opacity value for each layer. Fig. 11.1 shows a sample blended image from Photoshop. The opacity value is 70% for the foreground image and 30% for the background image.



Fig. 11.1. Alpha blending using Photoshop 7.0. Opacity is 70% for the foreground and 30% for the background.

The image processing library developed in this thesis performs alpha blending automatically. The following sections describe algorithms for alpha blending and related

applications. Library functions provide many applications that use alpha blending, including Water Reflection, Corner Fold, Fog, Shadow, and Fade-in/Fade-out. None of these applications are available as functions in Photoshop. They must all be done manually. Water Reflection was described in Chapter 10. Corner Fold is described in Chapter 14. This chapter describes Fog, Shadow, and Fade-in/Fade-out. Results for these three applications are provided, along with instructions for achieving an integer, or fixed-point, implementation.

## 11.2 Alpha Blending

Alpha blending is performed by taking a weighted average of pixel values in two images. Suppose  $g(x,y)$  is the blended image,  $\alpha$  is the weighting factor, and  $f_1(x,y)$  and  $f_2(x,y)$  are the foreground and background images. The transformation equation for alpha blending is

$$g(x,y) = \alpha \cdot f_1(x,y) + (1 - \alpha) \cdot f_2(x,y) \quad (11.1)$$

Note that weights for  $f_1(x,y)$  and  $f_2(x,y)$  must add up to 1, which is why only one weighting factor  $\alpha$  needs to be specified.



(a) Alpha Blending Function

(b) Photoshop 7.0

Fig. 11.2. Comparison of alpha blending results for  $\alpha = 0.7$ .

The operation of alpha blending takes its name from the weighting factor  $\alpha$ , which can be any real number from 0 to 1. When  $\alpha = 0$ , the blended result consists of only the background image. When  $\alpha = 1$ , the result consists of only the foreground. Any intermediate value for  $\alpha$  creates a mixture of the two images. An example of alpha blending where  $\alpha = 0.7$  for the foreground is given in Fig. 11.2. A result from the Alpha Blending function is compared with the Photoshop result previously shown in Fig. 11.1.

### 11.3 Fog

Alpha blending can be used in many different image processing operations, producing results that are aesthetically pleasing yet straightforward to achieve. One application of alpha blending is the creation of foggy images. Although real-life fog tends to obscure a landscape and can be a photographer's worst nightmare, artificial fog is a powerful tool for creating moody images capable of evoking emotions ranging from solitude to serenity. Fog is an atmospheric effect that can transform an otherwise mundane image into a dramatic piece of art.

The Fog function produces fog by blending a stock cloud image with a given input image using different alpha factors to obtain light or heavy fog, as shown in Fig. 11.3.



(a) Original

(b) Light Fog

(c) Heavy Fog

Fig. 11.3. Fog.

### 11.4 Shadow

Another useful application of alpha blending is the casting of a shadow over part of an image. A fast implementation is to store a shadow image in permanent memory and use alpha blending to combine it with part of an input image. The shape of the shadow can be pre-defined. Fig. 11.4 shows a sample palm leaf shadow.



Fig. 11.4. Palm leaf shadow.

One complication in the Shadow function is that unlike the simple alpha blending used in Fog, casting a shadow might require parts of a shadow image to be treated as transparent when blended with the input image. A simple way to handle this is to define a value  $\rho$  as the transparent pixel value. Pixels in a transparent region can be assigned the value  $\rho$  and the standard alpha blending equation can be modified to become

$$g(x, y) = \begin{cases} w \cdot f_1(x, y) + (1 - w) \cdot f_2(x, y) & \text{if } f_2(x, y) \neq \rho \\ f_1(x, y) & \text{if } f_2(x, y) = \rho \end{cases}$$

where  $f_1(x, y)$  is the input image,  $f_2(x, y)$  is the shadow image, and  $g(x, y)$  is the output.

To provide more flexibility, the Shadow function allows four different orientations for each shadow image. Fig. 11.5 shows the four orientations for a palm leaf shadow placed within an all-white sample image. For each orientation, the palm leaf can be placed anywhere along an edge of the image.



Fig. 11.5. Four different orientations for palm leaf shadow within a sample input image.



(a) Original Image

(b) Image with Shadow

Fig. 11.6. Casting a palm leaf shadow. Original image is from Microsoft Word 2000 SP3 clip art.

To obtain the four different shadow orientations, the original palm leaf shadow image is rotated by increments of  $90^\circ$  using the affine transform equation

$$\begin{bmatrix} x_{out} & y_{out} \end{bmatrix} = \begin{bmatrix} x_{in} & y_{in} \end{bmatrix} \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

where  $(x_{in}, y_{in})$  is the input pixel location and  $(x_{out}, y_{out})$  is the output pixel location. Fig. 11.6 shows an example of adding a palm leaf shadow.

## 11.5 Fade In, Fade Out

A natural extension of alpha blending is fade-in/fade-out for an image sequence. Fading into or out of a scene is a technique commonly used in motion pictures and television to provide a smooth transition during a scene change. To achieve this effect, alpha blending is applied on each still frame in the sequence while gradually changing the factor  $\alpha$ . Fade-in is produced by gradually changing  $\alpha$  from 0 to 1. Fade-out is produced by changing  $\alpha$  from 1 to 0.

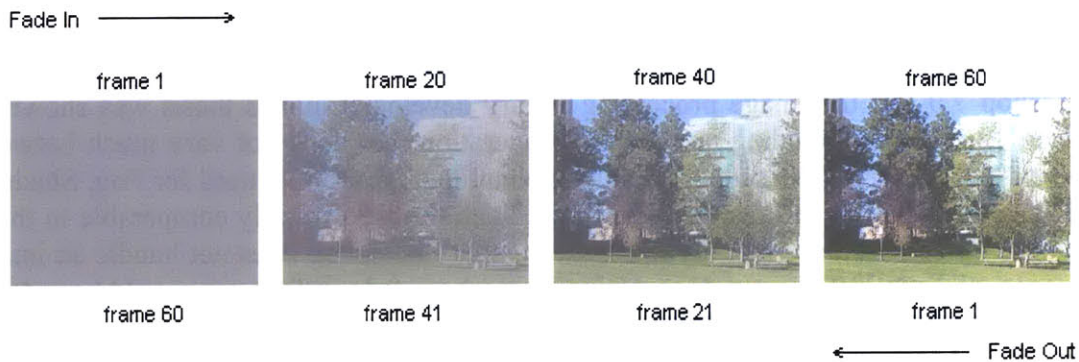


Fig. 11.7. Fade-in and fade-out for a 60-frame image sequence.

In fade-in, the output image sequence starts with an unaltered frame from the input sequence and gradually fades into a given still image by the end of the sequence. In fade-out, the opposite happens. The output sequence starts with a given still image and gradually fades out to the input image sequence. Fig. 11.7 shows a set of results for fade-in and fade-out using a test image sequence of 60 frames. For ease of comparison, each frame in the input sequence repeats the same still image. The still image that the sequence fades to or from is an all-gray image. The instantaneous shots in Fig. 11.7 show fading in when seen from left to right and fading out when seen from right to left. In general, the input image sequence need not consist of identical frames, and the still image need not be monochromatic.

## 11.6 Integer Implementation

All functions for alpha blending and related applications were implemented on camera phones with integer microprocessors, where floating-point arithmetic produced unacceptable latency. To achieve real-time performance, implementation had to be done using strictly integer, or fixed-point, arithmetic. This integer restriction presented a problem during implementation. Recall that the transformation equation for alpha blending, given by equation (11.1) and repeated below for convenience, assumes that  $\alpha$  is a floating-point number in the range  $0 \leq \alpha \leq 1$ .

$$g(x, y) = \alpha \cdot f_1(x, y) + (1 - \alpha) \cdot f_2(x, y) \quad (11.1)$$

One solution is to convert  $\alpha$  to a fixed-point representation by multiplying it with a suitably large integer factor  $f$ , such as  $f = 2^7 = 128$ , and rounding to the nearest integer. For instance, if  $\alpha = 0.7$ , it can be converted to  $0.7 \times 2^7 = 89.6 \cong 90$ . All intermediate arithmetic operations are done using the new fixed-point representation. The final result is obtained by dividing by  $f$ .

## 11.7 Comparison with Photoshop

As mentioned previously, Photoshop does not have a standard alpha blending function. Alpha blending must be created manually in Photoshop. A comparison of results from Photoshop 7.0 and the image processing library developed in this thesis was shown in Section 12.2. Alpha blending is a standard operation that does not vary much between different applications. The same basic alpha blending algorithm is used for Fog, Shadow, and Fade-in/Fade-out, and results from these functions are similarly comparable to those produced by Photoshop. The only difference is that Photoshop does not handle an image sequence, as Fade-in/Fade-out does. Using Photoshop, fade-in/fade-out would have to be manually created on frame-by-frame basis.

## 11.8 Implementation for Subsampled YCbCr and YCrCb

Alpha blending and related applications have been implemented for all RGB, YCbCr, and YCrCb color formats without requiring special processing of subsampled YCbCr or YCrCb formats.



# Chapter 12

## Arbitrary Rotation (Patent Pending)

### 12.1 Introduction

Arbitrary rotation of two-dimensional images is a fundamental operation in image processing. Although standard equations and algorithms for rotation have long been developed, most of these are designed for hardware platforms with the power and memory capacity of average desktop computers. The algorithms often require floating-point arithmetic, multipass transformations, or large look-up tables, all of which are generally avoided on mobile devices, where power and memory space are highly limited. Many mobile devices, including camera phones, have only integer microprocessors. On such platforms, floating-point arithmetic requires additional floating-point emulators or software libraries, which result in unacceptable latency and power consumption. The growing demand for more image processing capabilities on camera phones requires the development of new real-time algorithms that avoid floating-point arithmetic, minimize memory use, and reduce power consumption.



(a) Original Image

(b) Rotated Image

Fig. 12.1. Rotation by  $45^\circ$  clockwise around the center of an image.

This thesis provides a real-time, purely integer rotation algorithm and implementation that fulfills all three criteria. This new approach can rotate an image about a given point by an arbitrary degree angle  $\alpha$  that can be any integer in the range  $-\infty < \alpha < \infty$ . Positive angles produce clockwise rotation and negative angles produce counterclockwise

rotation. A sample result from this new approach is shown in Fig. 12.1, in which a 520 x 390 image is rotated by 45° clockwise around its center.

Camera phones and other mobile devices are not the only applications that can benefit from the new rotation algorithm. For instance, the purely integer algorithm can be readily implemented as specialized hardware on a high-speed integer microprocessor. Such microprocessors are a necessary component of biomedical image processing applications, such as medical resonance imaging (MRI) [11]. A fast rotation implementation is also needed for real-time visual pattern recognition, including registration-based feature extraction and feature matching [10]. It can even be used for films and games, as part of image warping and morphing for computer-animated sequences [16].

## 12.2 Existing Equations and Algorithms

To perform a rotation, standard transformation equations are used to map input pixels in an original image to output pixels in a rotated image. The equations are

$$x_{out} = x_o + (x_{in} - x_o) \cdot \cos \theta - (y_{in} - y_o) \cdot \sin \theta, \quad (12.1)$$

$$y_{out} = y_o + (x_{in} - x_o) \cdot \sin \theta + (y_{in} - y_o) \cdot \cos \theta, \quad (12.2)$$

where  $\theta$  is the rotation angle in radians,  $(x_o, y_o)$  is the center of rotation,  $(x_{in}, y_{in})$  is the input pixel location, and  $(x_{out}, y_{out})$  is the output pixel location.

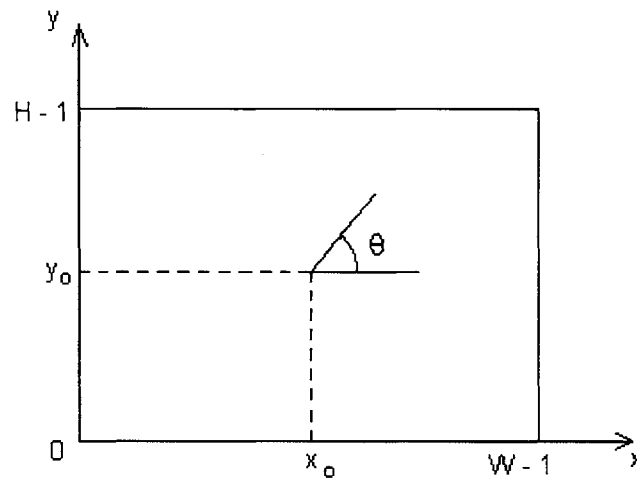


Fig. 12.2. Coordinate system used for rotation equations.

The coordinate system used in these equations has an origin located at the lower left corner of the image, an x axis pointing to the right, and a y axis pointing upward, as

shown in Fig. 12.2. For illustration purposes, the figure shows a center of rotation that is the center of the image. In practice, any arbitrary point can be used as the center of rotation.

Equations (12.1) and (12.2) can be rewritten in terms of a transformation matrix as follows:

$$\begin{bmatrix} x_{out} - x_o \\ y_{out} - y_o \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \cdot \begin{bmatrix} x_{in} - x_o \\ y_{in} - y_o \end{bmatrix} \quad (12.3)$$

Most existing rotation algorithms use floating-point arithmetic and can be grouped into two categories, one-pass and multipass [10]. One-pass algorithms directly apply the matrix transformation in equation (12.3). These algorithms are straightforward and obtain highly accurate results. The tradeoff is that many floating-point computations are needed. Each pixel requires four floating-point multiplications and two floating-point additions, which result in a very slow implementation.

To speed up the implementation, multipass floating-point algorithms have been developed. These algorithms decompose the transformation matrix into multiple subtransformations that can be quickly performed. A typical subtransformation is a skewing transform that moves the image in only one direction and can be implemented by rapid row or column shifts. The problem is that frequency aliasing tends to accumulate over multiple passes. For instance, when a two-pass approach is used with

$$\begin{bmatrix} x_{out} - x_o \\ y_{out} - y_o \end{bmatrix} = \begin{bmatrix} \cos \theta & 0 \\ \sin \theta & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & -\tan \theta \\ 0 & 1/\cos \theta \end{bmatrix} \cdot \begin{bmatrix} x_{in} - x_o \\ y_{in} - y_o \end{bmatrix}, \quad (12.4)$$

the first subtransformation, given by the middle matrix on the right-hand side, causes vertical compression in the intermediate image. The resulting loss of high-frequency information must be prevented by resampling and enlarging the original image. This extra processing might wipe out the time saved by a multipass approach. For example, a two-pass transformation that corrects for frequency aliasing is

$$\begin{bmatrix} x_{out} - x_o \\ y_{out} - y_o \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \tan \theta & 1 + \tan \theta \cdot \tan(\theta/2) \end{bmatrix} \cdot \begin{bmatrix} 1 - \sin \theta \cdot \tan(\theta/2) & -\sin \theta \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_{in} - x_o \\ y_{in} - y_o \end{bmatrix}$$

Notice the increased complexity compared to equation (12.4). The large number of floating-point calculations required in such multipass algorithms are often too slow for real-time performance on camera phones.

What is needed is a purely integer algorithm that can be directly implemented on an integer microprocessor. In general, integer microprocessors are much faster than floating-point microprocessors. The resulting speed-up in an integer implementation would make it unnecessary to use complex multipass methods for real-time applications.

The only problem is that integer arithmetic produces insufficient accuracy when directly calculating sines and cosines for arbitrary rotation. While standard software library functions can be called to find sine and cosine values, these functions are usually implemented with floating-point arithmetic and tend to slow down the performance significantly.

To maintain speed and accuracy without using floating-point arithmetic, it has been suggested that look-up tables be used to store sine and cosine values [5]. If arbitrary integer rotation angles are supported, this approach would require large look-up tables that occupy a considerable amount of memory. Unfortunately, on camera phones, memory space is extremely scarce. A look-up table approach would therefore fail to work on such platforms.

This thesis solves all of the above problems by providing a real-time, purely integer algorithm for arbitrary rotation that requires no use of look-up tables.

### 12.3 A New Integer Implementation

This section describes the new integer algorithm and implementation for arbitrary rotation. The new algorithm takes a direct one-pass approach to rotation by applying transformation equations (12.1) and (12.2) without using float-point arithmetic or look-up tables.

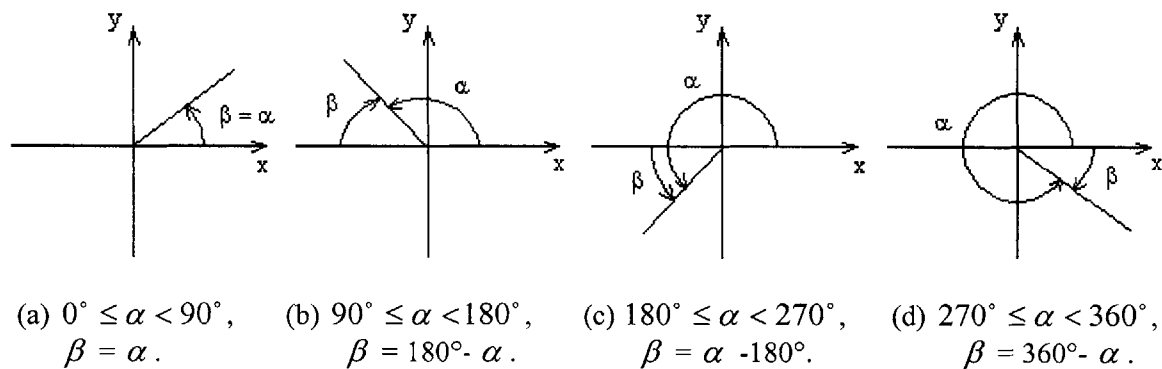


Fig. 12.3. Mapping  $\alpha$  to  $\beta$  when  $0^\circ \leq \alpha < 360^\circ$ .

Suppose the angle of rotation is  $\alpha$  degrees. Before applying the transformation,  $\alpha$  is first mapped to an equivalent angle  $\beta$  in the range  $0^\circ \leq \beta \leq 90^\circ$ . If  $\alpha$  is in the range  $0^\circ \leq \alpha < 360^\circ$ , the mapping is performed as shown in Fig. 12.3.

If  $\alpha < 0^\circ$ , it is first mapped to the range  $0^\circ \leq \alpha < 360^\circ$  using  $[(\alpha \bmod 360^\circ) + 360^\circ]$  and then mapped to  $\beta$  according to Fig. 12.3. Similarly, if  $\alpha > 360^\circ$ , it is first mapped to the range  $0^\circ \leq \alpha < 360^\circ$  using  $(\alpha \bmod 360^\circ)$  and then mapped according to Fig. 12.3.

Once  $\alpha$  is in the range  $0^\circ \leq \alpha < 360^\circ$ , the sine and cosine relationships between  $\alpha$  and  $\beta$  are as follows:

$$\begin{aligned} \text{If } 0^\circ \leq \alpha < 90^\circ, \quad \beta = \alpha \quad \text{and} \quad & \begin{cases} \sin \alpha = \sin \beta \\ \cos \alpha = \cos \beta \end{cases} \\ \text{If } 90^\circ \leq \alpha < 180^\circ, \quad \beta = 180^\circ - \alpha \quad \text{and} \quad & \begin{cases} \sin \alpha = \sin(180^\circ - \beta) \\ \cos \alpha = -\cos(180^\circ - \beta) \end{cases} \\ \text{If } 180^\circ \leq \alpha < 270^\circ, \quad \beta = \alpha - 180^\circ \quad \text{and} \quad & \begin{cases} \sin \alpha = -\sin(\beta - 180^\circ) \\ \cos \alpha = -\cos(\beta - 180^\circ) \end{cases} \\ \text{If } 270^\circ \leq \alpha < 360^\circ, \quad \beta = 360^\circ - \alpha \quad \text{and} \quad & \begin{cases} \sin \alpha = -\sin(360^\circ - \beta) \\ \cos \alpha = \cos(360^\circ - \beta) \end{cases} \end{aligned}$$

To apply equations (12.1) and (12.2) using purely integer arithmetic, Taylor series expansions are used to approximate the sine and cosine functions. In general, the series expansions for the sine and cosine of an arbitrary radian angle are

$$\sin \theta = \theta - \frac{1}{3!}\theta^3 + \frac{1}{5!}\theta^5 - \frac{1}{7!}\theta^7 + \dots \quad (12.5)$$

$$\cos \theta = 1 - \frac{1}{2!}\theta^2 + \frac{1}{4!}\theta^4 - \frac{1}{6!}\theta^6 + \dots \quad (12.6)$$

where  $\theta$  is the equivalent of  $\beta$  in radians. That is,

$$\theta = \beta \cdot \frac{\pi}{180^\circ} \quad (12.7)$$

When angle  $\beta$  is in the range  $0^\circ \leq \beta \leq 40^\circ$ , only the first two terms of equations (12.5) and (12.6) are needed to produce accurate results. When  $\beta$  is in the range  $40^\circ < \beta \leq 90^\circ$ , the first four terms in the series are needed for an accurate approximation. Higher order terms can still be used if desired. However, these terms will have little effect on accuracy and will only decrease the speed of the implementation. The implementation in this thesis used the following approximations:

$$\sin \theta \cong \begin{cases} \theta - \frac{1}{3!}\theta^3 & \text{if } 0^\circ \leq \beta \leq 40^\circ \\ \theta - \frac{1}{3!}\theta^3 + \frac{1}{5!}\theta^5 - \frac{1}{7!}\theta^7 & \text{if } 40^\circ < \beta \leq 90^\circ \end{cases} \quad (12.8)$$

$$\cos \theta = \begin{cases} 1 - \frac{1}{2!}\theta^2 & \text{if } 0^\circ \leq \beta \leq 40^\circ \\ 1 - \frac{1}{2!}\theta^2 + \frac{1}{4!}\theta^4 - \frac{1}{6!}\theta^6 & \text{if } 40^\circ < \beta \leq 90^\circ \end{cases} \quad (12.9)$$

In equations (12.8) and (12.9), the relationship between  $\theta$  and  $\beta$  is given by equation (12.7). Although equations (12.7), (12.8), and (12.9) do not contain strictly integer terms, the non-integer terms can be converted to integers by multiplying with a suitably large integer factor. For instance,  $\frac{1}{3!}$  in equation (12.8) is first computed as a real number, then

multiplied by  $2^{10} = 1024$ , and finally rounded to an integer. Thus,  $\frac{1}{3!}$  is converted to

$\frac{1}{3!} \cdot 2^{10} = (0.16667) \cdot 1024 \cong 171$ . Other non-integer terms such as  $\frac{1}{5!}$  are converted to

integers in the same manner. Intermediate arithmetic operations are calculated using the new integer representations. After all intermediate operations are completed, the final result is obtained by dividing by the same integer factor, e.g.  $2^{10}$  for equation (12.8). This technique preserves accuracy during intermediate integer arithmetic operations. For more details on this method, a typical integer implementation is given in the next section.

## 12.4 Avoiding Integer Overflow

The rotation algorithm was implemented on a 32-bit RISC integer microprocessor. To reduce latency and achieve real-time performance, no integers greater than 32 bits were used. This 32-bit restriction caused problems with integer overflow. The solution was to rearrange the order of intermediate arithmetic operations to avoid overflow while maintaining the necessary degree of accuracy.

The following is a sample 32-bit integer implementation in C for equations (12.1) and (12.2) using the approximations in equations (12.8) and (12.9). In this example, the center of rotation is the center of the image. However, the rotation algorithm supports an arbitrary center of rotation located anywhere within an input image.

```
uint32 w, h, beta, radian;
int8 sinSign, cosSign;
int32 xin, yin, xout, yout, xo, yo, sintheta, costheta;

/*
** Center of rotation (xo, yo) is the center of the image in this
** example
```

```

*/
xo = w >> 1;
yo = h >> 1;

/* Indicates positive/negative sign for the sin and cos values */
sinSign = 1;
cosSign = 1;

/*
** "angle" is a signed 32-bit integer (int32) input parameter
** indicating the angle of rotation
*/
while (angle < 0) {
    angle += 360;
}
angle %= 360;

/* Map "angle" to beta in the range [0, 90] degrees */
if (angle <= 90) {
    beta = angle;
} else if (angle <= 180) {
    beta = 180 - angle;
    cosSign = -1;
} else if (angle <= 270) {
    beta = angle - 180;
    sinSign = -1;
    cosSign = -1;
} else {
    beta = 360 - angle;
    sinSign = -1;
}

/*
** Convert from degrees to (radians * 2^9) in fixed-point format
*/
radian = beta * 9;

/* Calculate sin and cos */
if (radian <= 360) {
    /* beta <= 40 degrees: use first 2 terms in Taylor series */
    sintheta = ((radian*(262144-(radian*radian)/6))>>8)*sinSign;
    costheta = (524288-radian*radian)*cosSign;
} else {
    if (radian == 810) {
        /* beta = 90 degrees */
        sintheta = 524288*sinSign;
        costheta = 0;
    } else {
        /* 40 < beta < 90: use first 4 terms in Taylor series */
        sintheta = (((radian*(262144-(radian*radian)/6))>>8) +
            (((((((radian*radian*radian)/120)>>3)*radian)>>10)*
            radian)/42)*(((11010048-radian*radian)>>10))>>21))*
            sinSign;
        costheta = ((524288-radian*radian) +
            (((((((radian*radian*radian)/24)>>3)*radian/30)>>10)*
            ((7864320-radian*radian)>>10))>>12))*cosSign;
    }
}

```

```

}

/*
** For each output pixel (xout, yout), map to the corresponding
** input pixel (xin, yin). Center of rotation is (xo, yo).
*/
xin = xo+(((xout-xo)*costheta + (yout-yo)*sintheta)>>19);
yin = yo+(((xo-xout)*sintheta + (yout-yo)*costheta)>>19);

```

In the above implementation,  $w$  is the width of the image,  $h$  is the height, and  $angle$  is the angle of rotation, all of which are stored as 32-bit integers. A similar implementation can be made in C++, Java, or any other programming language.

This method of dealing with integer overflow can be applied to any implementation on any microprocessor, including 64-bit processors, 128-bit processors, and so forth.

## 12.5 Implementation for Subsampled YCbCr and YCrCb

The current implementation supports RGB, YCbCr, and YCrCb color formats, including subsampled formats such as YCbCr 4:2:2, YCbCr 4:2:0, and YCrCb 4:2:0. RGB images can be processed by directly applying equations (12.1) and (12.2) along with the Taylor series approximation given in equations (12.5) and (12.6). However, images in a subsampled YCbCr or YCrCb color format cause problems due to their subsampled chrominance components, Cb and Cr.

For instance, one common form of subsampled YCbCr is YCbCr 4:2:2, in which the Cb and Cr components of either pixel rows or pixel columns are subsampled by a factor of two. Two formats are H2V1 YCbCr 4:2:2, in which pixel columns are subsampled but pixel rows are unaffected, and H1V2 YCbCr 4:2:2, in which pixel rows are subsampled but pixel columns are unaffected. Fig. 12.4 shows the H2V1 format. For every four columns, there are 4 Y values, 2 Cb values, and 2 Cr values, hence the name 4:2:2. If columns are numbered starting from zero, only even columns have the Cb component and only odd columns have Cr. Hereafter, YCbCr 4:2:2 refers to H2V1 YCbCr 4:2:2.

0	1	2	3
YCb	YCr	YCb	YCr
YCb	YCr	YCb	YCr

Fig. 12.4. H2V1 YCbCr 4:2:2.

Directly applying equations (12.1) and (12.2) will fail to guarantee correctly alternating YCb-YCr pixels in the rotated image. To solve this problem, a temporary YCbCr 4:4:4 image is created from the original subsampled YCbCr 4:2:2 image. The rotation



equations are applied to the YCbCr 4:4:4 image to obtain a temporary YCbCr 4:4:4 output, which is then subsampled to produce the final YCbCr 4:2:2 output.

The temporary YCbCr 4:4:4 image contains all three YCbCr components for each pixel. It is created by taking pairs of adjacent YCb and YCr pixels and having each pixel borrow the missing chrominance component from its partner. The YCb pixel borrows a Cr component from its paired YCr pixel. The YCr pixel borrows a Cb component from the YCb pixel. Normally, this method does not produce an accurate, visually pleasing YCbCr 4:4:4 image from a YCbCr 4:2:2 image. However, since the temporary YCbCr 4:4:4 image is used only for intermediate processing, the accuracy of the intermediate result does not matter as long as the final output is accurate.



Fig. 12.5. Comparison of rotated images for 4 different color formats. Angle of rotation is 30° clockwise.

For the purpose of applying equations (12.1) and (12.2), this method of creating a YCbCr 4:4:4 image and performing rotation in the YCbCr 4:4:4 domain works very well. The final YCbCr 4:2:2 output is obtained from the temporary YCbCr 4:4:4 output by

discarding the extra Cb or Cr component for each pixel. There is no visible difference between a YCbCr 4:2:2 rotated image produced in this manner and an equivalent RGB rotated image. Fig. 12.5 (a) and (b) compare an RGB rotated image produced from RGB input with the corresponding YCbCr 4:2:2 image produced from YCbCr 4:2:2 input. Both images were rotated  $30^\circ$  clockwise with a center of rotation at the center of the image.

This method of creating a temporary YCbCr 4:4:4 image can be extended to handle input images in any subsampled YCbCr or YCrCb color format. For instance, another common format is YCbCr 4:2:0, in which each block of  $2 \times 2$  pixels has only one Cb and one Cr component. A temporary YCbCr 4:4:4 image is created by duplicating the Cb and Cr components for each pixel in each  $2 \times 2$  block. Transformation equations are applied to the temporary YCbCr 4:4:4 image to obtain a YCbCr 4:4:4 output, which is subsampled to YCbCr 4:2:0 by discarding the extra chrominance components. There is no visible difference between YCbCr 4:2:0 rotated images produced in this manner and RGB rotated images produced from RGB input. This can be seen by comparing Fig. 12.5 (a) with (c), which shows results for YCbCr 4:2:0. For further comparison, Fig. 12.5 (d) gives results for YCrCb 4:2:0, where the order of the Cb and Cr components has been switched. All images were created with a rotation angle of  $30^\circ$  and a center of rotation at the center of the image.

# Chapter 13

## Other Affine Transforms

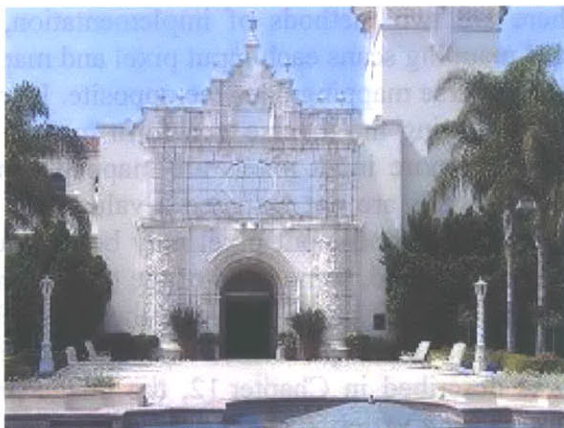
### 13.1 Introduction: Geometric Spatial Transformations

Geometric spatial transformations change the spatial relationships between pixels in an image. These transformations are also known as rubber-sheet transformations and can be visualized as printing an image on a sheet of rubber and stretching the sheet according to a given set of rules. One common application for geometric transformations is image registration, in which two different images must be realigned or resized before they can be merged together for quantitative comparison or visualization [17].

A geometric spatial transformation is expressed by a mapping of pixel coordinates in an input image to pixel coordinates in an output image. Suppose an image  $f$  is defined over the  $(x_{in}, y_{in})$  coordinate system. A geometric transformation is used to produce a distorted image  $g$  defined over the  $(x_{out}, y_{out})$  coordinate system. The transformation operation is defined as

$$(x_{out}, y_{out}) = T\{(x_{in}, y_{in})\}$$

For instance, if  $(x_{out}, y_{out}) = T\{(x_{in}, y_{in})\} = (3x_{in}, 3y_{in})$ , image  $f$  is simply downsized by a factor of three. Fig. 13.1 (a) shows an original 520 x 390 image. Fig. 13.1 (b) shows the downsized image created in Adobe Photoshop 7.0.



(a) Original Image



(b) Downsized Image

Fig. 13.1. Downsizing by a factor of 3.

## 13.2 Affine Transforms

One of the most widely used geometric transformations is the affine transform. An affine transform is any transformation that preserves collinearity, parallelism, and ratios of distances. Straight lines remain straight, parallel lines remain parallel, but rectangles may become parallelograms. While an affine transform preserves linear proportions, it does not necessarily preserve angles or lengths.

An affine transform is also called an affinity [17]. Translation, rotation, scaling, stretching, and shearing are all affine transforms. Other examples include dilation, similarity transformations, and spiral transformations. A composition of various affine transforms is also an affine transform.

The general affine transform matrix equation is

$$\begin{bmatrix} x_{out} \\ y_{out} \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \cdot \begin{bmatrix} x_{in} \\ y_{in} \\ 1 \end{bmatrix}, \quad (13.1)$$

where input and output pixels,  $(x_{in}, y_{in}, 1)$  and  $(x_{out}, y_{out}, 1)$ , are given in homogeneous coordinates. The corresponding Cartesian coordinates are  $(x_{in}/1, y_{in}/1) = (x_{in}, y_{in})$  and  $(x_{out}/1, y_{out}/1) = (x_{out}, y_{out})$ . In general, any set of homogeneous coordinates  $(x, y, t)$  is equivalent to the Cartesian coordinates  $(x/t, y/t)$ .

Depending on the values of elements  $a_{ij}$  in the transformation matrix, equation (13.1) can perform rotation, shearing, scaling, and translation. Each of these operations is described in a subsequent section. Affine transforms can be combined by multiplying their transformation matrices and finding a single composite matrix.

For any affine transform matrix, there are two methods of implementation, forward mapping and inverse mapping. Forward mapping scans each input pixel and maps it to an output pixel using the transform matrix. Inverse mapping does the opposite. It scans each output pixel and maps it to an input pixel using the inverse transform matrix. Forward mapping can cause problems when two or more input pixels are mapped to the same output pixel or when some of the output pixels are not assigned a value. For instance, when an image is being downsized, more than one input pixel may be mapped to the same output pixel. When an image is being sheared, some parts of the output image may not be mapped to any input pixel and should be left blank. To avoid these complications, all affine transform functions in this thesis are implemented with inverse mapping. Besides Arbitrary Rotation, which was described in Chapter 12, the image processing library in this thesis uses affine transforms for a variety of functions, which are described in the following sections.

### 13.3 Shear

Shear is an affine transform that changes rectangles to parallelograms. There are two types of shear, horizontal and vertical. Horizontal shear is produced by the equation

$$\begin{bmatrix} x_{out} \\ y_{out} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & \alpha & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_{in} \\ y_{in} \\ 1 \end{bmatrix},$$

where  $\alpha$  is the horizontal shear factor, which can be any real number in the range  $-\infty < \alpha < \infty$ . Vertical shear is produced by the equation

$$\begin{bmatrix} x_{out} \\ y_{out} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ \beta & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_{in} \\ y_{in} \\ 1 \end{bmatrix},$$

where  $\beta$  is the vertical shear factor, which can be any real number in the range  $-\infty < \beta < \infty$ .

Besides the two basic forms of shear, horizontal and vertical shear can be combined to produce more advanced shearing. The order of combination affects the transformation matrix and the final output. If horizontal shear is performed first, followed by vertical shear, the transformation equation is

$$\begin{bmatrix} x_{out} \\ y_{out} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & \alpha & 0 \\ \beta & 1 + \alpha\beta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_{in} \\ y_{in} \\ 1 \end{bmatrix}.$$

If vertical shear performed first, the transformation equation becomes

$$\begin{bmatrix} x_{out} \\ y_{out} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 + \alpha\beta & \alpha & 0 \\ \beta & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_{in} \\ y_{in} \\ 1 \end{bmatrix}.$$

The order of combining horizontal and vertical shear significantly affects the resulting image. Fig. 13.2 compares two results from combined horizontal and vertical shear. Fig. 13.2 (a) shows the result of performing horizontal shear first. Fig. 13.2 (b) shows the result of performing vertical shear first. For both images, the same set of horizontal and vertical shear factors,  $\alpha = 0.4$  and  $\beta = 0.6$ , were used and blank regions were colored black. For simplicity, the Shear function in this thesis always performs combined shearing by applying horizontal shear first, followed by vertical shear.



(a) Horizontal, Vertical



(b) Vertical, Horizontal

Fig. 13.2. Comparison of results from combined horizontal and vertical shear with  $\alpha = 0.4$  and  $\beta = 0.6$ .

In this thesis, the constraints of camera phone screen size made it necessary to create output images that were the same size as the input images. As a result, images produced by combined horizontal and vertical shear are resized to have the same dimensions as the original input image. Fig. 13.3 (a) shows a combined horizontal and vertical shear result for  $\alpha = 0.3$  and  $\beta = 0.3$  that has been resized. Images produced by strictly horizontal or strictly vertical shear are wrapped around, as shown in Fig. 13.3 (b), where  $\alpha = -0.5$  and  $\beta = 0$ .



(a)  $\alpha = 0.3$ ,  $\beta = 0.3$



(b)  $\alpha = -0.5$ ,  $\beta = 0$

Fig. 13.3. Shear results.

### 13.4 Scaling: Overlap and Resize

Scaling is an affine transform produced by the matrix equation

$$\begin{bmatrix} x_{out} \\ y_{out} \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_{in} \\ y_{in} \\ 1 \end{bmatrix},$$

where  $s_x$  is the horizontal scaling factor and  $s_y$  is the vertical scaling factor, both of which can be arbitrary real numbers in the range  $(-\infty, \infty)$ .



(a) Foreground



(b) Background

Fig. 13.4. Original 398 x 298 images.



Fig. 13.5. Overlap result. Overlap region is 100 x 120 pixels with the top left corner located at  $(x, y) = (10, 269)$ .

Scaling is used in the Overlap function, which combines two input images, a foreground and a background, so that the foreground image is inserted into a region in the background image. Foreground and background images do not have to be the same size. The region of overlap can be any size as long as it fits within the background image. The foreground image may be resized to fit the overlap region. Fig. 13.4 shows two original 398 x 298 images. Fig. 13.4 (a) shows the foreground image and Fig. 13.4 (b) shows the background image. Fig. 13.5 shows a result from Overlap, where the overlap region is 100 x 120 pixels and has an upper left corner located at  $(x, y) = (10, 269)$ . The coordinate system is defined with an origin at the lower left corner of the image, an x axis pointing to the right, and a y axis pointing upward.

### 13.5 Translation: Film Strip

Translation is an affine transform produced by the matrix equation

$$\begin{bmatrix} x_{out} \\ y_{out} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & \delta_x \\ 0 & 1 & \delta_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_{in} \\ y_{in} \\ 1 \end{bmatrix},$$

where  $\delta_x$  is the amount of horizontal translation and  $\delta_y$  is the amount of vertical translation, both of which can be arbitrary real numbers in the range  $(-\infty, \infty)$ .



Fig. 13.6. Original 398 x 298 images.

Translation is used in the Film Strip function, which combines the right part of the first input image with the left part of the second input to simulate the effect of a moving film strip. The two input images must be the same size. The user specifies the location of the vertical boundary between the two inputs. The boundary is where the second input starts. If the boundary is at  $x = 0$ , the output is simply the second input image. If the boundary is



at  $x = W$ , where  $W$  is the input image width, the output is the first image. Fig. 13.6 shows two original  $398 \times 298$  input images. Fig. 13.7 shows a Film Strip result with a boundary located at  $x = 275$ . The coordinate system is defined with an origin at the lower left corner of the image, an x axis pointing to the right, and a y axis pointing upward.



Fig. 13.7. Film Strip. Boundary is at  $x = 275$ .

### 13.6 Composites: Horizontal, Vertical, and Center

There are three different functions that create horizontal, vertical, and center composite images, respectively. For all composite functions, input and output images must have the same size and the coordinate system is defined with an origin at the lower left corner of the image, an x axis pointing to the right, and a y axis pointing upward.



Fig. 13.8. Horizontal Composite. Boundary is at  $x = 250$ .

In Horizontal Composite, two images are combined horizontally. The left part of the first image is combined with the right part of the second image. The user specifies the location of the vertical boundary between the two inputs. The boundary is where the second image starts. If the boundary is at  $x = 0$ , the output is simply the second image. If the boundary is at  $x = W$ , where  $W$  is the input image width, the output is the first image. Fig. 13.8 shows a result where the boundary is located at  $x = 250$ .



Fig. 13.9. Vertical Composite. Boundary is at  $y = 197$ .

Vertical Composite combines two images vertically. The top part of the first image is combined with the bottom part of the second image. The user specifies the location of the horizontal boundary between the two inputs. The boundary is where the second image starts. If the boundary is at  $y = 0$ , the output is simply the first image. If the boundary is at  $y = H$ , where  $H$  is the input image height, the output is the second image. Fig. 13.9 shows a result where the boundary is located at  $y = 197$ .



Fig. 13.10. Center Composite. Width of center part is 199 pixels.

Center Composite combines the center part of the first input image with the sides of the second input. The user specifies the width of the center part. If the width is 0, the output is simply the second input image. If the width is  $W$ , where  $W$  is the input image width, the output is the first input image. Fig. 13.10 shows a result where the width is 199 pixels.

### 13.7 Integer Implementation

All affine transform functions were implemented on a 32-bit RISC integer microprocessor, where floating-point arithmetic produced unacceptable latency. Each function was implemented with strictly integer arithmetic. All floating-point elements in the transformation matrices were converted to fixed-point format. Floating-point numbers were multiplied by a suitably large integer factor  $f$ , such as  $f = 2^7 = 128$ , to obtain fixed-point representations, which were used in intermediate integer operations to maintain accuracy. The final results were obtained by dividing by  $f$ .

### 13.8 Comparison with Photoshop

Photoshop 7.0 offers Shear as a standard function but does not offer any of the other functions in this thesis that use affine transforms: Overlap, Film Strip, Horizontal Composite, Vertical Composite, and Center Composite. All of these operations must be done manually in Photoshop. In addition, Photoshop's Shear provides neither vertical shear nor combined horizontal and vertical shear.



(a) Shear Function

(b) Photoshop 7.0

Fig. 13.11. Comparison of Shear with Photoshop 7.0.

Rather than using a shear factor as input, Photoshop's Shear asks the user to move the control vertices of a line to indicate the desired amount of shear. As a result, Photoshop does not support all possible values for the shear factor. The range of supported values is

limited by how far the control vertices can be moved. Fig. 13.11 compares results from the Shear function and Photoshop. The Shear function parameters are  $\alpha = -0.3$  and  $\beta = 0$ . Fig. 13.12 shows the parameter settings in Photoshop.

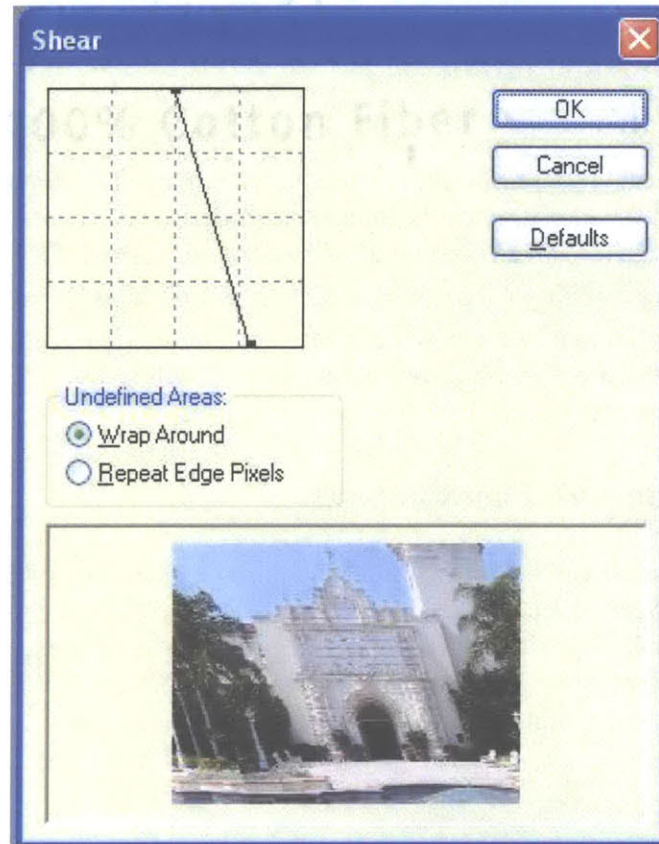


Fig. 13.12. Photoshop 7.0 parameter settings for Shear.

### 13.9 Implementation for Subsampled YCbCr and YCrCb

All affine transform functions developed in this thesis support RGB, YCbCr, and YCrCb color formats, including subsampled formats such as YCbCr 4:2:2, YCbCr 4:2:0, and YCrCb 4:2:0. However, the Shear function requires special processing for subsampled YCbCr and YCrCb images using the method described in Chapter 2. There is no visible difference between a subsampled YCbCr or YCrCb image sheared with this method and an equivalent RGB sheared image.

For comparison, Fig. 13.13 (a), (b), (c), and (d) give Shear results produced from RGB, YCbCr 4:2:2, YCbCr 4:2:0, and YCrCb 4:2:0 inputs. All four images were produced with the same horizontal and vertical shear factors,  $\alpha = -0.5$  and  $\beta = 0.4$ .

Other affine transform functions do not require special processing for subsampled YCbCr and YCrCb images. The only restriction is that for subsampled YCbCr and YCrCb images in Film Strip, Horizontal Composite, and Center Composite, the boundary location and width must always be even. In these functions, if the user enters a boundary location or width that is odd, the inappropriate value changed to an even number by simply incrementing or decrementing by 1.



Fig. 13.13. Shear results for 4 different color formats with  $\alpha = -0.5$  and  $\beta = 0.4$ .



# Chapter 14

## Corner Fold

### 14.1 Introduction

Folding over a corner of an image is an effect used primarily in computer graphics. Existing graphics applications cannot create a corner fold where a translucent, warped version of the original corner region can be seen. Instead, the folded corner is left blank, and illumination and shading effects are added to simulate a curved surface, as illustrated in Fig. 14.1.

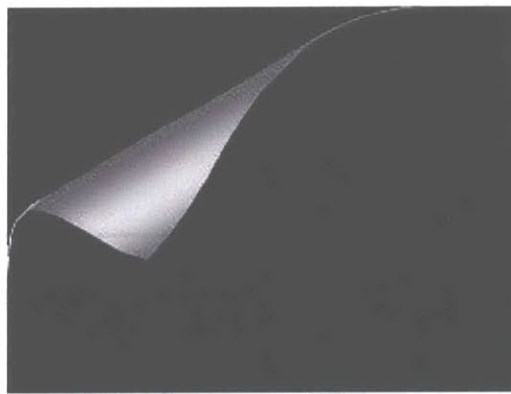


Fig. 14.1. A typical corner fold with a blank, opaque folded region.

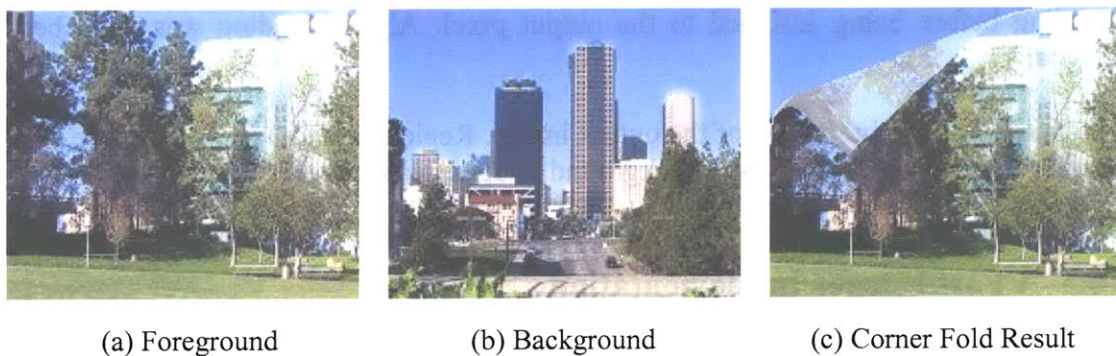


Fig. 14.2. Corner Fold effect produced by the Corner Fold algorithm.

A more sophisticated, visually informative approach is to display a translucent folded corner, as shown in Fig. 14.2 (c), which was produced by the Corner Fold algorithm

developed in this thesis. The original 176 x 144 foreground and background images are shown in Fig. 14.2 (a) and (b).

## 14.2 Algorithm

There are three main steps in the Corner Fold algorithm. The first step generates the shape of the corner fold. Parametric curve-fitting is used to find a piecewise linear approximation for the smooth, curved boundaries of the corner fold region. Gaussian elimination is used to determine the parameters in the approximation.

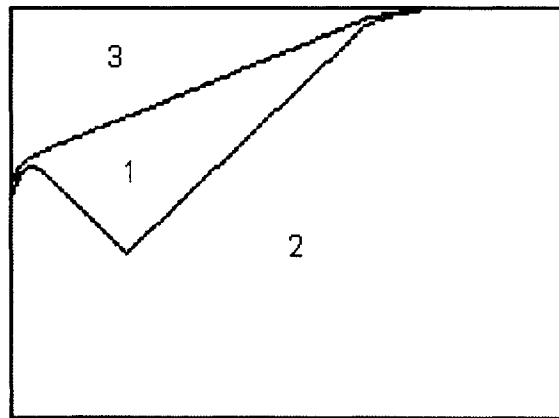


Fig. 14.3. Regions in Corner Fold output image.

The second step fills in the corner fold region in the output image. This is region 1 in Fig. 14.3. The region is scanned pixel by pixel. Each output pixel is mapped to the corresponding input pixel using the equations from step one. To make the folded corner seem more realistic and translucent, the input pixel value is mixed with white using alpha blending before being assigned to the output pixel. Alpha blending was described in Chapter 11.

The third step fills in the rest of the output image. Regions 2 and 3 in Fig. 14.3 are copied directly from the foreground and background images, respectively.

## 14.3 A Fast Integer Implementation

The Corner Fold function was implemented on a 32-bit RISC integer microprocessor, where floating-point arithmetic produced unacceptable latency. The curve-fitting process in the first step of the algorithm was streamlined to achieve a real-time integer implementation. Instead of calculating the corner fold boundaries and mapping pixels on the fly, the boundaries were calculated in advance and stored in a template image, shown in Fig. 14.4.



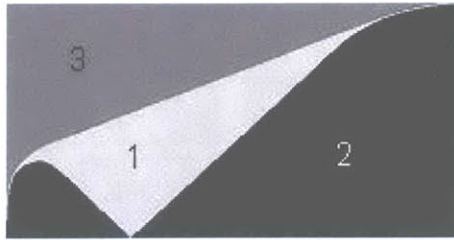


Fig. 14.4. Corner fold template.

During implementation, the template image is resized to fit the input and output images. There are three distinct regions in the template, each with a different color to make it quick and easy to determine the region to which each output pixel belongs. The template regions correspond to the three regions labeled in Fig. 14.3. Region 1 is light gray and represents the corner fold area. Region 2 is black and represents the area directly copied from the foreground image. Region 3 is dark gray and represents the area directly copied from the background image.

To speed up the implementation even further, the mapping equations for region 1 are approximated by a combination of reflection, rotation, and shifting of pixel rows.

## 14.4 Results

The Corner Fold function allows one of four corners in an original image to be folded over. The four possibilities are the upper left, upper right, lower left, and lower right corners. Two sets of Corner Fold results are given in the following figures. Fig. 14.5 shows the first pair of original foreground and background images. Both are 520 x 390 pixels. Fig. 14.6 shows the four possible Corner Fold outputs. Fig. 14.7 shows the second pair of images, which are 398 x 298 pixels. The four outputs are shown in Fig. 14.8.



(a) Foreground



(b) Background

Fig. 14.5. Original 520 x 390 images.

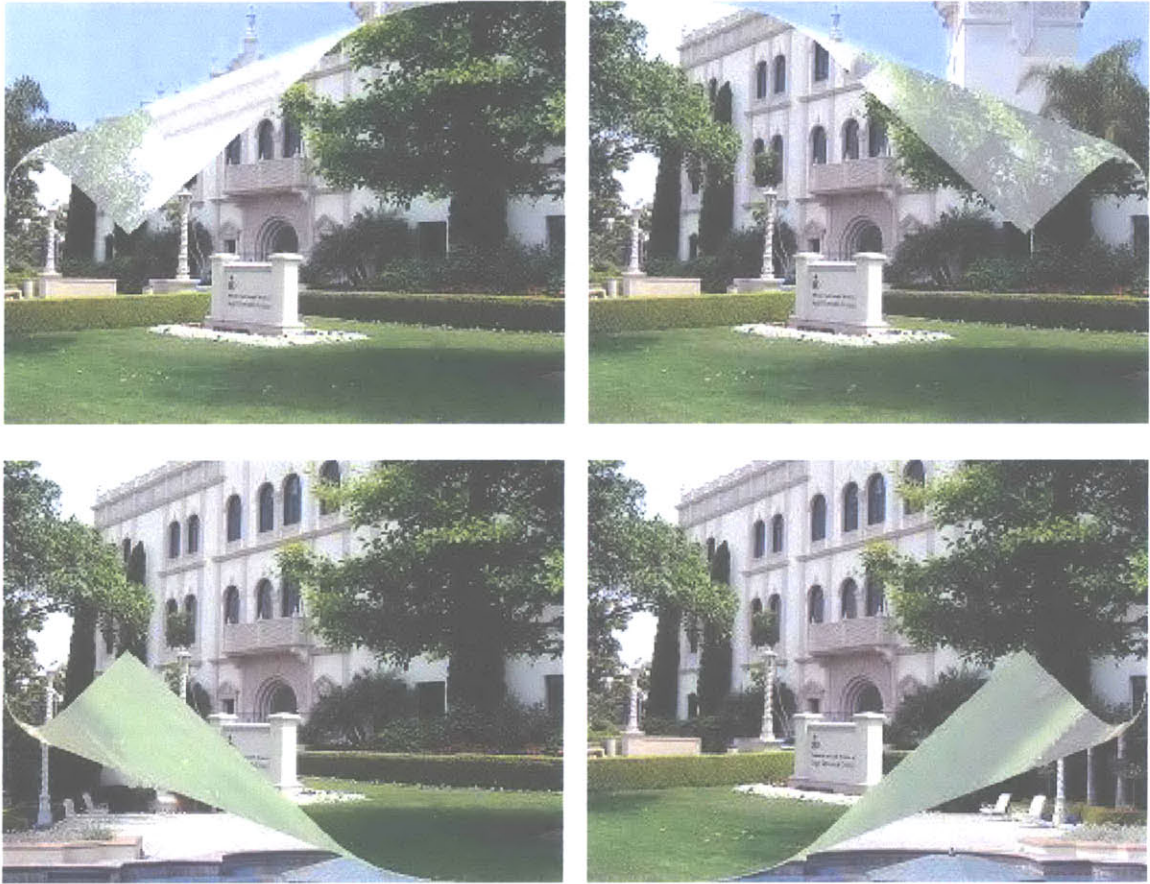


Fig. 14.6. 520 x 390 Corner Fold results.



(a) Foreground



(b) Background

Fig. 14.7. Original 398 x 298 images.

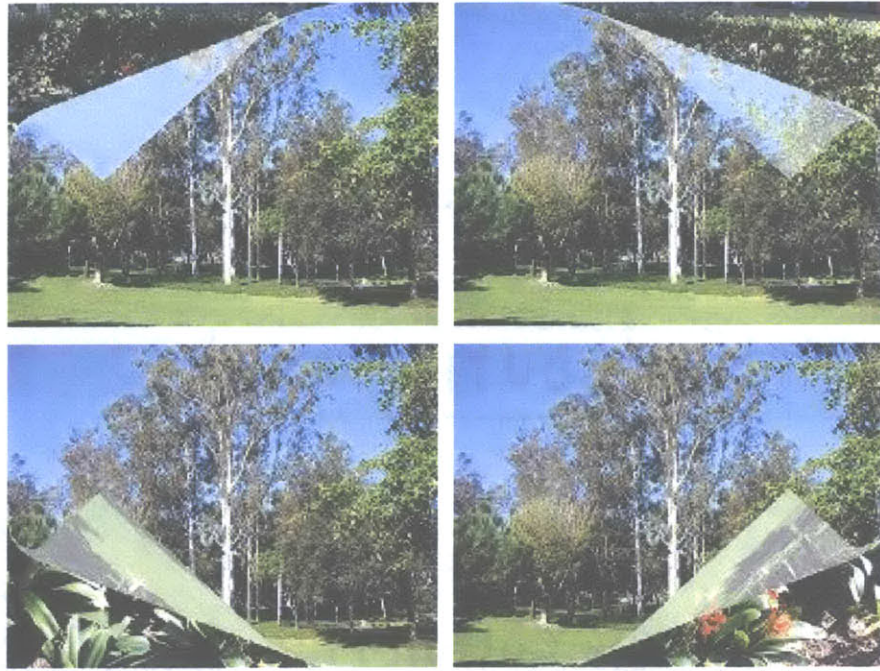


Fig. 14.8. 398 x 298 Corner Fold results.

In the above Corner Fold results, an alpha blending factor of  $\alpha = 50\%$  was used to blend the corner fold region with white and produce a translucent effect. This is equivalent to simply averaging input pixel values with white, which is  $(R, G, B) = (255, 255, 255)$  or  $(Y, Cb, Cr) = (235, 128, 128)$ . In theory, any alpha blending factor between 0 and 1 can be used. A factor of 0 produces a blank corner fold region, as shown in Fig. 14.9 (a). A factor of 1 produces a completely transparent region, as shown in Fig. 14.9 (b).



Fig. 14.9. Comparison of results for different values of  $\alpha$ , the alpha blending factor.

## 14.5 Comparison with Photoshop

Adobe Photoshop 7.0 does not provide a Corner Fold function.

## 14.6 Implementation for Subsampled YCbCr and YCrCb

The current implementation of Corner Fold supports RGB, YCbCr, and YCrCb color formats, including subsampled formats such as YCbCr 4:2:2, YCbCr 4:2:0, and YCrCb 4:2:0. RGB images can be processed by directly applying the algorithm described in Section 14.2. However, images in a subsampled YCbCr or YCrCb color format cause problems due to their subsampled chrominance components, Cb and Cr.



Fig. 14.10. Comparison of Corner Fold results for 4 different color formats.

The method described in Chapter 2 was used to handle subsampled YCbCr and YCrCb color formats by creating an intermediate YCbCr 4:4:4 or YCrCb 4:4:4 image. There is

no visible difference between a subsampled YCbCr or YCrCb Corner Fold result obtained in this manner and an equivalent RGB result. For comparison, Fig. 14.10 (a), (b), (c), and (d) give Corner Fold results produced from RGB, YCbCr 4:2:2, YCbCr 4:2:0, and YCrCb 4:2:0 inputs.



# Chapter 15

## Color Change

### 15.1 Introduction

Color mapping is the rearrangement of colors in an image. It is commonly used to achieve special effects or enhance the appearance of a digital photograph. There is a variety of color mapping operations. The three RGB color channels can be mixed or swapped with each other. The saturation can be changed to rob color from selected areas or convert an image to grayscale. Hues can be rotated around the color spectrum to tint the image. Hue and saturation changes can be combined to balance skin tones, adjust highlights and shadows, and bolster contrast. Entire ranges of colors can be mapped to new values based on their hues, saturation levels, and brightness values. Through color mapping, grayscale images can be colorized and badly developed photographs can have their contrast and color range recalibrated. With its many practical uses, color mapping is an indispensable tool in image processing and computer graphics.

Four common color mapping functions available in Adobe Photoshop 7.0 are Invert, Equalize, Threshold, and Posterize. The Invert function converts every color to its exact opposite, as if creating a photographic negative. The Equalize function maps the darkest and lightest colors in a selected image region to black and white, respectively, and redistributes pixel values over the brightness spectrum to produce a higher contrast image. The Threshold function changes colors lighter than a given threshold value to white and colors darker than the threshold to black. Finally, the Posterize function performs color quantization by dividing the full range of 256 color values into a specified number of quantization levels.

In general, color mapping functions are relatively straightforward to implement. For completeness, the current image processing library offers a basic color mapping function called Color Change, which swaps the RGB color channels of an image.

### 15.2 Algorithm

The Color Change algorithm permutes the order of the three RGB color components to create five different color schemes for an image. Including the original order, the six permutations are RGB, RBG, BGR, BRG, GRB, and GBR.

### 15.3 Results

Given a single input image, the Color Change function can produce five different outputs, depending on the color permutation selected by the user. Fig. 15.1 (a) shows an original

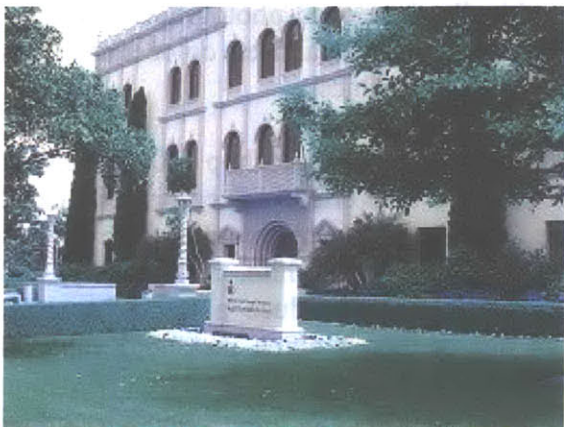
520 x 390 image in RGB format. Fig. 15.1 (b) through (f) show the five possible results labeled with the corresponding permutations of color components. Notice that each output emphasizes a different color: purple, green, blue, orange, and magenta.



(a) Original Image (RGB)



(b) RBG



(c) BGR



(d) BRG



(e) GRB



(f) GBR

Fig. 15.1. Color Change results.



## **15.4 Comparison with Photoshop**

Adobe Photoshop 7.0 does not provide a Color Change function.

## **15.5 Implementation for YCbCr and YCrCb**

The current implementation of Color Change supports RGB, YCbCr, and YCrCb color formats, including subsampled formats such as YCbCr 4:2:2, YCbCr 4:2:0, and YCrCb 4:2:0. RGB images can be processed by directly applying the Color Change algorithm given in Section 15.2. However, YCbCr and YCrCb images must be transformed to the RGB domain using equation (2.1) before the algorithm can be applied. The final YCbCr and YCrCb outputs are obtained by converting back to the YCbCr domain using equation (2.2).



# Chapter 16

## Conclusion and Future Work

### 16.1 Conclusion

This thesis has developed a smart image processing library that provides a number of advanced image processing functions, most of which are not found on existing camera phones and some of which are lacking in Adobe Photoshop and other desktop applications. All library functions have been implemented for real-time or near real-time performance on actual cell phone chips with strictly fixed-point arithmetic. The algorithms and implementations presented here can be used on any hardware platform since cell phone platforms are the most restrictive in terms of microprocessor speed, memory space, and power supply.

The library functions provide many frequently used operations in image processing and useful special effects in computer graphics. All operations and special effects are produced automatically with little or no human intervention. This thesis has established a solid framework for an easy-to-use advanced image processing system. Any number of new functions and features can be added as desired.

### 16.2 Future Work

In real life, there are many complex problems in image processing. Some of the functions in this thesis can serve as a starting point or as motivation to pursue solutions to these problems. One such problem is the automatic detection and replacement of foreground objects. In several functions described in Chapter 11, regions in images are selected manually and blended with new objects and scenery. Instead of relying on humans to identify a region for blending, these functions can be extended to automatically detect foreground objects from complex backgrounds and allow not only blending but also complete removal or replacement of the foreground by new objects and scenery.

Automatic foreground object detection can be done in two steps. The first step divides an image into partially overlapping small blocks (e.g. 64x64) and computes two-dimensional discrete wavelet coefficients for each block. Regions can be classified as background or potential foreground using a maximum likelihood rule or a neural network based on the energy of wavelet coefficients [31] and other possible features. The second step finds foreground objects from potential foreground regions. To accurately detect foreground object boundaries, local statistics such as color co-occurrence and gradient vectors can be used with a Bayes decision rule. This two-step approach will be faster and more robust to noise than existing algorithms [26].

Once a foreground object is successfully detected, one of three possible operations can be performed. If blending of the foreground is desired, alpha blending can be used. If the foreground object needs to be further processed for such applications as human identification, a set of features can be computed from the object and used for classification. Finally, if the foreground needs to be completely removed and replaced by the background, a wavelet-domain hidden Markov tree (HMT) can be used to synthesize new background patches. Gaps left by the removal of foreground objects can be filled by the synthesized patches. Background patches created by an HMT approach should be statistically and visually better than those created by the algorithm in [12].

These detection and replacement techniques are useful in numerous applications, including video surveillance, human motion analysis, and object based video encoding. Other functions developed in this thesis also have potential uses in more sophisticated, special-purpose applications.

# References

- [1] Adobe Systems Incorporated, “Adobe Photoshop 7.0 Help: Blur Filters.”
- [2] Advanced RISC Machines Limited, “Application Note 55: Floating-Point Performance.” ARM DAI 0055A. Jan. 1998.
- [3] Advanced RISC Machines Limited, “ARM926EJ-S Technical Reference Manual.” ARM DDI 0198D. Jan. 2004.
- [4] H. C. Andrews and B. R. Hunt, *Digital Image Restoration*. Upper Saddle River, New Jersey: Prentice Hall, 1977.
- [5] Y.-W. Bai and C.-H. Lai, “A Bitmap Scaling and Rotation Design For SH1 Low Power CPU,” *Proceedings of the 2<sup>nd</sup> ACM International Workshop on Modeling, Analysis and Simulation of Mobile Systems*, 101-106. 1997.
- [6] Boston Museum of Science, “Exploring Linear Perspective.”  
<http://www.mos.org/sln/Leonardo/ExploringLinearPerspective.html>.
- [7] S. Caplin, *How to Cheat in Photoshop, 2<sup>nd</sup> ed.* Burlington, MA: Focal Press, 2004.
- [8] S. Card, T. Moran, and A. Newell, *The Psychology of Human-Computer Interaction*. Hillsdale, New Jersey: Erlbaum Assoc., 1983.
- [9] I. Carlbom and J. Paciorek, “Planar Geometric Projections and Viewing Transformations,” *Computing Surveys*, vol. 10, no. 4, 465-502. Dec. 1978.
- [10] X. Chen, S. Lu, X. Yuan, L. Chen, and B. Zeng, “Midpoint Line Algorithm for High-speed High-accuracy Rotation of Images,” *IEEE International Conference on Systems, Man, and Cybernetics*, vol. 4, 2739-2744. Oct. 1996.
- [11] R. W. Cox and R. Tong, “Two- and Three-Dimensional Image Rotation Using the FFT,” *IEEE Transactions on Image Processing*, vol. 8, no. 9, 1297-1299. Sept. 1999.
- [12] A. Criminisi, P. Pérez, and Kentaro Toyama, “Region Filling and Object Removal by Exemplar-Based Image Inpainting,” *IEEE Transactions on Image Processing*, vol. 13, 1200-1212, Sept. 2004.
- [13] F. Durand, “Digital Photography and Image-Based Editing.”  
<http://graphics.csail.mit.edu/~fredo/photo.html>.
- [14] R. Fisher, S. Perkins, A. Walker, and E. Wolfart, *Sobel Edge Detector*. 2003.  
<http://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm>.
- [15] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practice, Second Edition in C*. Boston, MA: Addison-Wesley, 1996.
- [16] D. Fraser, H. He, and R. A. Schowengerdt, “High-Fidelity Image Warping for Serial and Parallel Processing,” *Proceedings of the International Conference on Image Processing*, vol. 3, 719-722. Sept. 1996.
- [17] R. C. Gonzalez, R. E. Woods, and S. L. Eddins, *Digital Image Processing Using MATLAB*. Upper Saddle River, NJ: Pearson Prentice Hall, 2004.
- [18] E. Hamilton, “JPEG File Interchange Format.”  
<http://home.intercom.it/~fsoft/ablast97/fileform/JFIF.html>. Sept. 1992.
- [19] B. Huggins, *Creative Photoshop Lighting Techniques*. New York, NY: Lark Books, 2004.

- [20] T. A. Keahey and E. L. Robertson, "Nonlinear Magnification Fields," *Proceedings of the IEEE Symposium on Information Visualization '97*, 51-58, 121. Oct. 1997.
- [21] T. A. Keahey and E. L. Robertson, "Techniques for Non-Linear Magnification and Transformations," *Proceedings of the IEEE Symposium on Information Visualization '96*, 38-45. Oct. 1996.
- [22] E. LaMar, B. Hamann, and K. Joy, "A Magnification Lens for Interactive Volume Visualization," *Proceedings of the 9th Pacific Conference on Computer Graphics and Applications*, 223-232. Oct. 2001.
- [23] C. Lampton, *Gardens of Imagination: Programming 3D Maze Games in C/C++*. Corte Madera, CA: Waite Group Press, 1994.
- [24] E. Lengyel, *Mathematics for 3D Game Programming & Computer Graphics*, 2<sup>nd</sup> ed. Hingham, MA: Charles River Media, 2004.
- [25] H. Li, B. S. Manjunath, and S. K. Mitra, "Multisensor Image Fusion Using the Wavelet Transform," *Proceedings of IEEE International Conference on Image Processing*, vol. 1, pp. 13-16, Nov. 1994.
- [26] L. Li, W. Huang, I. Y. Gu, and Q. Tian, "Statistical Modeling of Complex Backgrounds for Foreground Object Detection," *IEEE Transactions on Image Processing*, vol. 13, 1459-1472, Nov. 2004.
- [27] J. S. Lim, *Two-dimensional Signal and Image Processing*. Englewood Cliffs, NJ: Prentice Hall, 1990.
- [28] N. L. Max and D. M. Lerner, "A Two-and-a-Half-D Motion-Blur Algorithm," *ACM SIGGRAPH Computer Graphics, Proceedings of the 12<sup>th</sup> Annual Conference on Computer Graphics and Interactive Techniques*, vol. 19, no. 3, pp.85-93, July 1985.
- [29] N. Max, "SIGGRAPH '84 Call for Omnimax Films," *Computer Graphics*, vol. 16, no. 4, 208-214. Dec. 1982.
- [30] D. McClelland, *Photoshop 7 Bible*. New York, NY: Wiley Publishing, 2002.
- [31] A. Mojsilovic, M. V. Popovic, and D. M. Rackov, "On the selection of an optimal wavelet basis for texture characterization," *IEEE Transactions on Image Processing*, vol. 9, 2043-2050, Dec. 2000.
- [32] W. B. Pennebaker and J. L. Mitchell, *JPEG Still Image Data Compression Standard*. New York, NY: Van Nostrand Reinhold, 1993.
- [33] M. Potmesil and I. Chakravarty, "Modeling Motion Blur in Computer-Generated Images," *ACM SIGGRAPH Computer Graphics, Proceedings of the 10<sup>th</sup> Annual Conference on Computer Graphics and Interactive Techniques*, vol. 17, no. 3, pp.389-399, July 1983.
- [34] C. Poynton, "Color FAQ - Frequently Asked Questions Color." [http://www.poynton.com/notes/colour\\_and\\_gamma/ColorFAQ.html#RTFTToC30](http://www.poynton.com/notes/colour_and_gamma/ColorFAQ.html#RTFTToC30). Dec. 2002.
- [35] M. Seul, L. O'Gorman, and M. J. Sammon, *Practical Algorithms for Image Analysis: Description, Examples, and Code*. Cambridge, United Kingdom: Cambridge University Press, 2000.
- [36] R. Smith and P. Anderson, "Relating Distortion to Performance in Distortion Oriented Displays," *Proceedings of the 6<sup>th</sup> Australian Conference on Computer-Human Interaction*, 6-11. Nov. 1996.
- [37] B. Steinmueller and U. Steinmueller, "Digital Outback Fine Art Photography Handbook." <http://www.outbackphoto.com/handbook/rawfileprocessing.html>. 2004

- [38] H. Tolsby, "Navigating in a Process Landscape," *HCI Proceedings of the 3<sup>rd</sup> International Conference EWCHI '93, Moscow, Russia*, 141-151. Aug. 1993.





# Appendix: SIP API

The following is the application program interface (API) for all functions in SIP.

## 1 sip\_magnify

### Parameters

```
#include "ipl.h"
ipl_status_type sip_magnify (
                                ipl_image_type *   in_img_ptr,
                                ipl_image_type *   out_img_ptr,
                                ipl_circle_type *   circle
                                );
```

→	in_img_ptr	Pointer to the input image
↔	out_img_ptr	Pointer to the output image
→	circle	Circular region to be magnified
←	status	Status returned at the end of the operation. Valid values are: IPL_SUCCESS – Operation completed successfully IPL_FAILURE – Failed to complete because of an error

### Description

This function magnifies an image in a circular region of arbitrary location and radius. It simulates the effect of viewing the image with a magnifying glass. The `circle` parameter specifies the center and radius of the magnified region. If part of the magnified region extends outside the image boundaries, only the part that remains within is displayed. The center of the magnified region must be within the input image. Pixels that are covered up by the magnified region are not displayed.

Figure 1 gives a 520 x 390 image with a magnified region of radius 100 pixels.

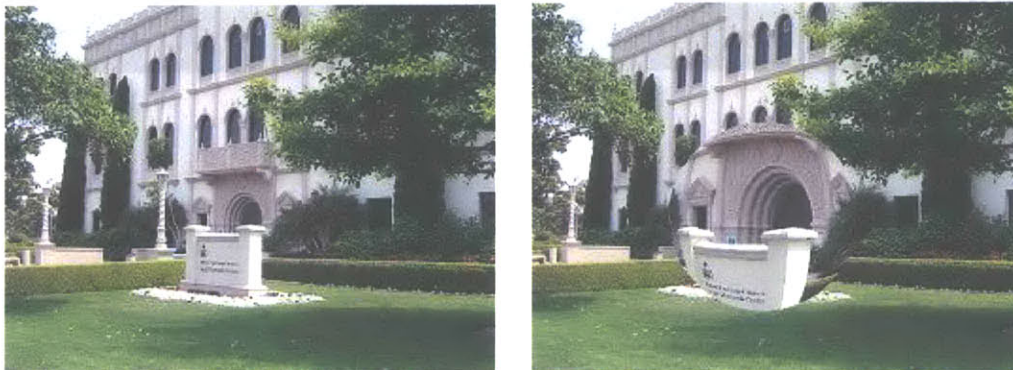


Figure 1 Magnify

## 2 sip\_pinch

### Parameters

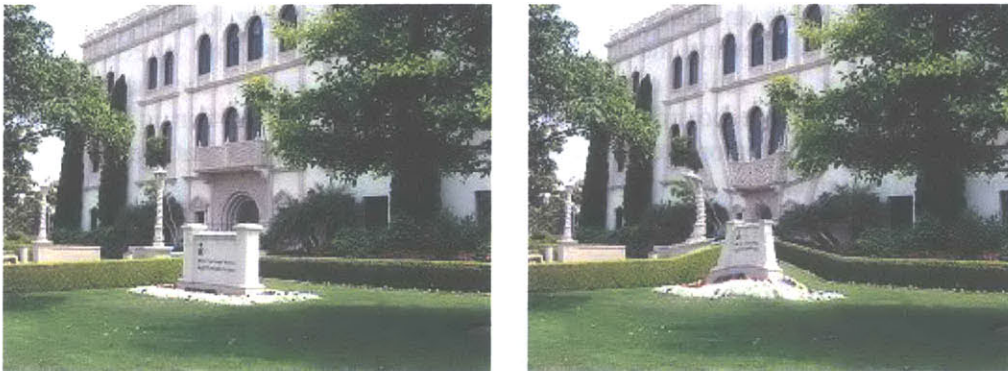
```
#include "ipl.h"
ipl_status_type sip_pinch (
                                ipl_image_type *   in_img_ptr,
                                ipl_image_type *   out_img_ptr,
                                ipl_circle_type *   circle
                                );
```

→	in_img_ptr	Pointer to the input image
↔	out_img_ptr	Pointer to the output image
→	circle	Circular region to be pinched
←	status	Status returned at the end of the operation. Valid values are: IPL_SUCCESS – Operation completed successfully IPL_FAILURE – Failed to complete because of an error

### Description

This function pinches an image in a circular region of arbitrary location and radius. The `circle` parameter specifies the center and radius of the pinched region. If part of the pinched region extends outside the image boundaries, only the part that remains within is displayed. The center of the pinched region must be within the input image. Pixels that are covered up by the pinched region are not displayed.

Figure 2 gives an image with a pinched region of radius 100 pixels. Image size is 520 x 390 pixels.



**Figure 2 Pinch**

### 3 sip\_perspective

#### Parameters

```
#include "ipl.h"
ipl_status_type sip_perspective (
    ipl_image_type *    in_img_ptr,
    ipl_image_type *    out_img_ptr,
    uint32              topWidth,
    uint32              xStart,
    uint16              fillerPixel,
    uint8               orientation
);
```

→	in_img_ptr	Pointer to the input image
↔	out_img_ptr	Pointer to the output image
→	topWidth	Width of top edge of trapezoid (must be smaller than bottom width)
→	xStart	Starting coordinate for the top edge (range: [0, width] or [0, height])
→	fillerPixel	Pixel value used to fill in blank regions outside the trapezoid
→	orientation	Orientation of output image: up (0), down (1), left (2), or right (3)
←	status	Status returned at the end of the operation. Valid values are: IPL_SUCCESS – Operation completed successfully IPL_FAILURE – Failed to complete because of an error

#### Description

This function creates a one-point perspective image shaped like a trapezoid. There are four input parameters:

1. Width of the top edge of the trapezoid
2. Starting coordinate of the top edge
3. A pixel value to fill in blank output regions outside the trapezoid
4. Orientation of the trapezoid

Regardless of the orientation, the top edge is defined as the smaller of the two parallel edges of the trapezoid. The top edge must be at least 1 pixel wide and no more than the width of the input image. A large top width corresponds to a vanishing point that is far away from the viewer or camera. A small top width indicates a vanishing point near the camera.

The filler pixel value must be given in a format compatible with the output color format. For instance, black is 0 for RGB565 and 0x8010 for YCbCr 4:2:2. White is 0xFFFF for RGB565 and 0x80EB for YCbCr 4:2:2. Notice that for YCbCr 4:2:2, the chrominance value (Cb or Cr) comes before the luma value, so white is 0x80EB instead of 0xEB80.

The trapezoid's height and bottom width are assumed to be the input height and width for "up" and "down" orientations. For "left" and "right" orientations, the trapezoid's height and bottom width are the input width and height, respectively. The orientation of the trapezoid can be: up (0), down (1), left (2), and right (3).

Figure 3 shows a 520 x 390 perspective image with a top edge of width 260 pixels and a starting coordinate of 130. The orientation is “up.” The coordinate system is defined with an origin at the lower left corner of the output image, an x axis pointing to the right, and a y axis pointing upward.

Figure 4 shows a 520 x 390 perspective image with a top edge of width 130 pixels and a starting coordinate of 324. The orientation is “left,” meaning that the vanishing point is on the left and is relatively close to the camera since the top width is rather small. The blank output regions have been filled in with black.



**Figure 3 Perspective, Orientation = Up**



**Figure 4 Perspective, Orientation = Left**

## 4 sip\_neon

### Parameters

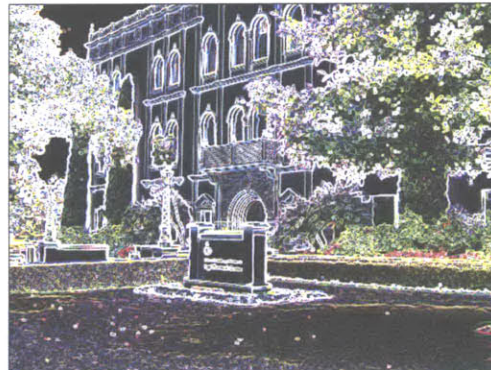
```
#include "ipl.h"
ipl_status_type sip_neon (
                                ipl_image_type *   in_img_ptr,
                                ipl_image_type *   out_img_ptr
                                );
```

→	in_img_ptr	Pointer to the input image
↔	out_img_ptr	Pointer to the output image
←	status	Status returned at the end of the operation. Valid values are: IPL_SUCCESS – Operation completed successfully IPL_FAILURE – Failed to complete because of an error

### Description

This function creates a neon image by outlining the edges of the input image with bright colors to resemble neon lights.

Figure 5 shows a sample neon image.



**Figure 5 Neon Effect**

## 5 sip\_radial\_blur

### Parameters

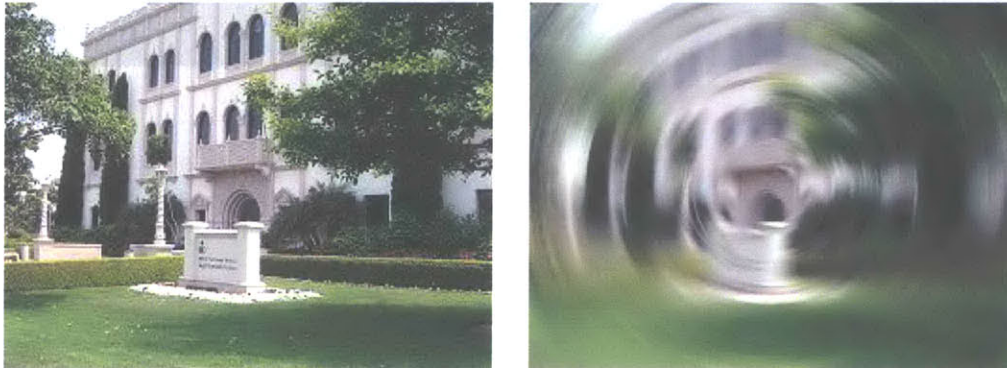
```
#include "ipl.h"
ipl_status_type sip_radial_blur (
    ipl_image_type *    in_img_ptr,
    ipl_image_type *    out_img_ptr,
    int32               xo,
    int32               yo,
    uint8               amount
);
```

→	in_img_ptr	Pointer to the input image
↔	out_img_ptr	Pointer to the output image
→	xo	X-coordinate of center of rotation
→	yo	Y-coordinate of center of rotation
→	amount	Amount of blur
←	status	Status returned at the end of the operation. Valid values are: IPL_SUCCESS – Operation completed successfully IPL_FAILURE – Failed to complete because of an error

### Description

This function produces spin radial blur. The center of rotation can be any point within the input image. The coordinate system is defined with an origin at the lower left corner of the image, an x axis pointing to the right, and a y axis pointing upward. The blur amount can be any nonnegative integer. No blurring occurs if the amount is 0.

Figure 6 shows a sample radial blur image with a blur amount of 20 and a center of rotation at the center of the image.



**Figure 6 Radial Blur**

## 6 sip\_motion\_blur

### Parameters

```
#include "ipl.h"
ipl_status_type sip_motion_blur (
                                ipl_image_type *   in_img_ptr,
                                ipl_image_type *   out_img_ptr,
                                uint32             size,
                                uint16            angle
                                );
```

→	in_img_ptr	Pointer to the input image
↔	out_img_ptr	Pointer to the output image
→	size	Length of motion blur window
→	angle	Angle of motion
←	status	Status returned at the end of the operation. Valid values are: IPL_SUCCESS – Operation completed successfully IPL_FAILURE – Failed to complete because of an error

### Description

This function produces motion blur in an arbitrary direction. The length of the motion blur window is measured in pixels and must be an odd, positive integer. The angle of motion is measured in degrees from the horizontal and can be any nonnegative integer.

Figure 7 shows an example of motion blur for a window size of 21 pixels and an angle of 30°.

Figure 8 shows an example for a window of size 35 pixels and an angle of 0°.



**Figure 7 Motion Blur, Size = 21, Angle = 30°**



**Figure 8 Motion Blur, Size = 35, Angle = 0°**



## 7 sip\_gaussian\_blur

### Parameters

```
#include "ipl.h"
ipl_status_type sip_gaussian_blur (
    ipl_image_type *   in_img_ptr,
    ipl_image_type *   out_img_ptr,
    uint32             size
);
```

→	in_img_ptr	Pointer to the input image
↔	out_img_ptr	Pointer to the output image
→	size	Size of Gaussian filter
←	status	Status returned at the end of the operation. Valid values are: IPL_SUCCESS – Operation completed successfully IPL_FAILURE – Failed to complete because of an error

### Description

This function produces Gaussian blur. The size of the Gaussian filter can be any positive, odd integer. Alternatively, the user can make multiple function calls with the same filter size to obtain a desired size. For instance, after  $n$  repeated function calls using a  $3 \times 3$  filter, the result is equivalent to a single function call using a  $(2n+1) \times (2n+1)$  filter.

Figure 9 shows an example of Gaussian blur using a  $3 \times 3$  filter.

Figure 10 shows an example of Gaussian blur using a  $35 \times 35$  filter. Alternatively, the  $35 \times 35$  result can also be obtained by applying a  $3 \times 3$  filter 17 times. This second approach, through repeated function calls, might be preferable if the user wants to visually gauge the amount of blurring desired.



**Figure 9 Gaussian Blur, Size = 3**



**Figure 10 Gaussian Blur, Size = 35**

## 8 sip\_uniform\_blur

### Parameters

```
#include "ipl.h"
ipl_status_type sip_uniform_blur (
    ipl_image_type *   in_img_ptr,
    ipl_image_type *   out_img_ptr,
    uint32             size
);
```

→	in_img_ptr	Pointer to the input image
↔	out_img_ptr	Pointer to the output image
→	size	Size of uniform filter
←	status	Status returned at the end of the operation. Valid values are: IPL_SUCCESS – Operation completed successfully IPL_FAILURE – Failed to complete because of an error

### Description

This function produces uniform blur using a uniform filter of arbitrary size. The filter size must be positive, odd, and no more than 63 to avoid integer overflow. Repeated function calls are not recommended since the cumulative effect will not be uniform blur. As the number of repeated calls increases, the result will approach Gaussian blur. To obtain actual Gaussian blur, it is recommended that `sip_gaussian_blur` be used.

Figure 11 shows an example of uniform blur using a filter size of 3x3 pixels.

Figure 12 shows an example using a filter size of 35x35 pixels.



**Figure 11 Uniform Blur, Size = 3**



**Figure 12 Uniform Blur, Size = 35**

## 9 sip\_alpha\_blend

### Parameters

```
#include "ipl.h"
ipl_status_type sip_alpha_blend (
    ipl_image_type *   in1_img_ptr,
    ipl_image_type *   in2_img_ptr,
    ipl_image_type *   out_img_ptr,
    uint8              alpha
);
```

→	in1_img_ptr	Pointer to the first input image
→	in2_img_ptr	Pointer to the second input image
↔	out_img_ptr	Pointer to the output image
→	alpha	Alpha blending factor (range: 0 to 100)
←	status	Status returned at the end of the operation. Valid values are: IPL_SUCCESS – Operation completed successfully IPL_FAILURE – Failed to complete because of an error

### Description

This function performs alpha blending for two images of the same size. The alpha blending factor  $\alpha$  can be any integer from 0 to 100. When the two input images are blended, the first image is weighted by  $\alpha$  % and the second is weighted by  $(100 - \alpha)$  %.

Figure 13 shows an example of alpha blending for  $\alpha = 70$ .



**Figure 13 Alpha Blending**

## 10 sip\_water\_reflection

### Parameters

```
#include "ipl.h"
ipl_status_type sip_water_reflection (
                                ipl_image_type *   in_img_ptr,
                                ipl_image_type *   out_img_ptr,
                                ipl_line_type *    boundary,
                                uint8              watertype
                                );
```

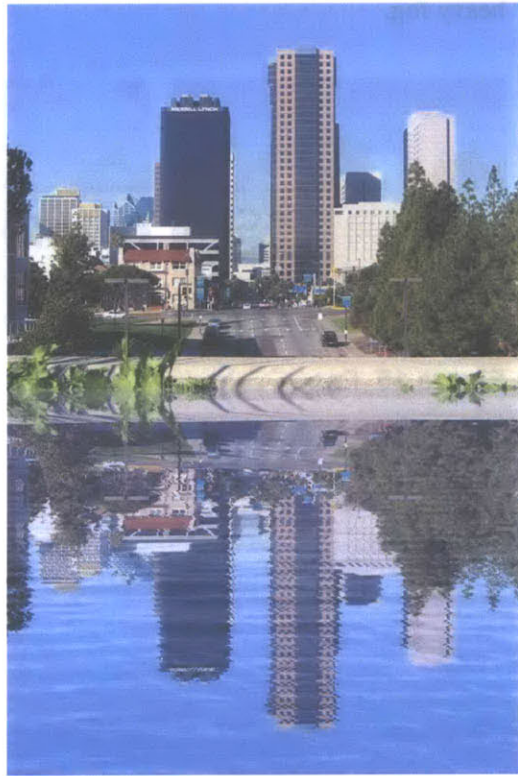
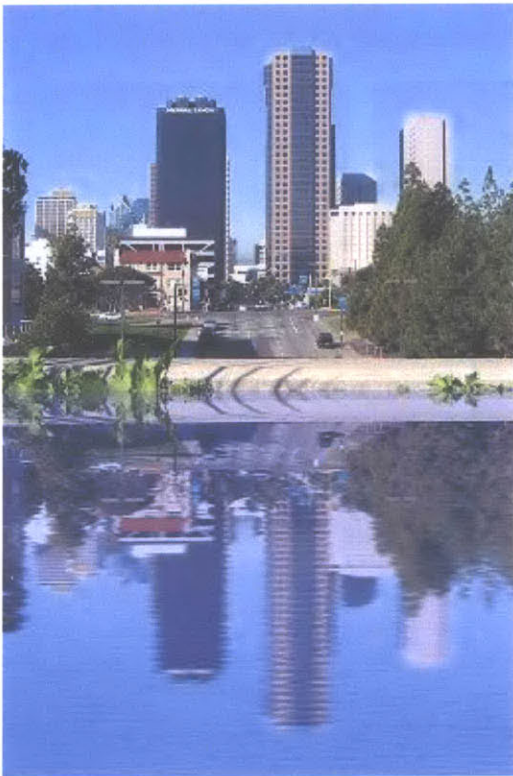
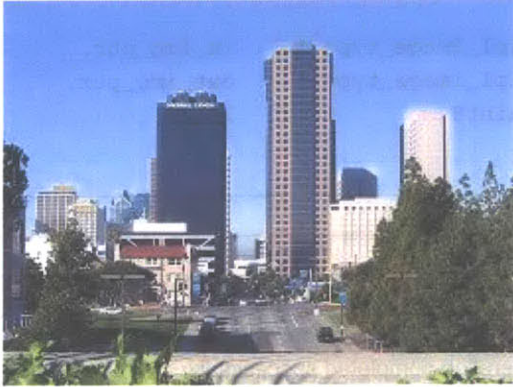
→	in_img_ptr	Pointer to the input image
↔	out_img_ptr	Pointer to the output image
→	boundary	Boundary line between original and reflected image
→	watertype	Type of water surface
←	status	Status returned at the end of the operation. Valid values are: IPL_SUCCESS – Operation completed successfully IPL_FAILURE – Failed to complete because of an error

### Description

This function automatically creates a realistic water reflection for varying weather conditions. The reflected image has a water surface with one of three levels of perturbation: still water (`watertype = 0`), water in a gentle breeze (`watertype = 1`), and water in windy conditions (`watertype = 2`). The output image must be large enough to fit both the original and reflected images.

The boundary line indicates where the reflected image begins. This line can be horizontal, downward sloping, or upward sloping. It cannot be vertical since water reflections are general below, rather than to the left or right of, an image. However, a user may create a vertical reflection boundary by rotating the original image by 90°, calling this function, and rotating the output back by 90°. This might be helpful if the original image was a photograph taken on a camera that was rotated 90°.

Figure 14 shows three sample water reflections where the boundary line is the bottom edge of the original image. The upper left image is the original image, the upper right image is a reflection in still water, the lower left image is a reflection in a gentle breeze, and the lower right image is a reflection in windy conditions.



**Figure 14 Water Reflection**

## 11 sip\_fog

### Parameters

```
#include "ipl.h"
ipl_status_type sip_fog (
                                ipl_image_type *   in_img_ptr,
                                ipl_image_type *   out_img_ptr,
                                uint8               level
                                );
```

→	in_img_ptr	Pointer to the input image
↔	out_img_ptr	Pointer to the output image
→	level	Level of fog: light (0) or heavy (1)
←	status	Status returned at the end of the operation. Valid values are: IPL_SUCCESS – Operation completed successfully IPL_FAILURE – Failed to complete because of an error

### Description

This function creates fog. There are two levels of fog, light (0) and heavy (1).

Figure 15 shows two sample results. The center image is light fog and the rightmost image is heavy fog.



**Figure 15 Fog**



## 12 sip\_shadow

### Parameters

```
#include "ipl.h"
ipl_status_type sip_shadow (
                                ipl_image_type *   in_img_ptr,
                                ipl_image_type *   out_img_ptr,
                                int32              xo,
                                int32              yo,
                                uint8              orientation
                                );
```

→	in_img_ptr	Pointer to the input image
↔	out_img_ptr	Pointer to the output image
→	xo	x-coordinate of top left corner of shadow
→	yo	y-coordinate of top left corner of shadow
→	orientation	Orientation of shadow (valid values: 0, 1, 2, 3)
←	status	Status returned at the end of the operation. Valid values are: IPL_SUCCESS – Operation completed successfully IPL_FAILURE – Failed to complete because of an error

### Description

This function casts a palm leaf shadow. Four different orientations are supported. The orientations are number from 0 to 3. The coordinate system is defined with the origin at the lower left corner of the image, the x axis pointing to the right, and the y axis pointing upward.

Figure 16 shows a sample result for a 520 x 390 input image. The top left corner is at  $(x_o, y_o) = (-20, 539)$  and orientation is 0.



**Figure 16 Shadow**

## 13 sip\_fadein\_fadeout

### Parameters

```
#include "ipl.h"
ipl_status_type sip_fadein_fadeout (
                                ipl_image_type *   in1_img_ptr,
                                ipl_image_type *   in2_img_ptr,
                                ipl_image_type *   out_img_ptr,
                                uint16             numFrames,
                                uint8              in_out
                                );
```

→	in1_img_ptr	Pointer to the input image sequence
→	in2_img_ptr	Pointer to the still image to be faded to/from
↔	out_img_ptr	Pointer to the output image sequence
→	numFrames	Number of frames in the image sequence
→	in_out	Type of fading (fade-in = 0, fade-out = 1)
←	status	Status returned at the end of the operation. Valid values are: IPL_SUCCESS – Operation completed successfully IPL_FAILURE – Failed to complete because of an error

### Description

This function performs fade-in/fade-out for an image sequence. Fading in is option 0 and fading out is option 1. The user specifies the number of image frames in the sequence and the width and height of each frame. All frames must have the same size. The user also provides the still image to fade to or from. This can be any arbitrary image, including images of blank, solid-color screens.

Figure 17 shows a sample output sequence where the still image is a uniform gray screen. These instantaneous shots show fading in when seen from left to right and fading out when seen from right to left.



**Figure 17** Fade In, Fade Out

## 14 sip\_rotate

### Parameters

```
#include "ipl.h"
ipl_status_type sip_rotate (
    ipl_image_type*    in_img_ptr,
    ipl_image_type*    out_img_ptr,
    uint16             angle,
    uint32             xo,
    uint32             yo,
    uint16             fillerPixel
);
```

→	in_img_ptr	Pointer to the input image
↔	out_img_ptr	Pointer to the output image
→	angle	Rotation angle in degrees
→	xo	X-coordinate of center of rotation
→	yo	Y-coordinate of center of rotation
→	fillerPixel	Pixel value used to fill in blank regions outside rotated image
←	status	Status returned at the end of the operation. Valid values are: IPL_SUCCESS – Operation completed successfully IPL_FAILURE – Failed to complete because of an error

### Description

This function performs rotation for any integer angle – positive, negative, or zero. The rotation angle must be given in degrees. Positive angles produce clockwise rotation. Negative angles produce counterclockwise rotation. An angle of zero produces no rotation.

Input and output image sizes must be equal. Any part of the input that is rotated outside the image boundaries is cut off.  $(x_0, y_0)$  is the center of rotation. The coordinate system is defined with an origin at the lower left corner of an image, an x axis pointing to the right, and a y axis pointing upward. The `fillerPixel` value is used to fill in blank output regions outside the rotated image. As in `sip_perspective`, `fillerPixel` must be given in a format compatible with the output color format. For instance, black is specified as 0 for RGB565 and 0x8010 for YCbCr 4:2:2.

Figure 18 shows a 520 x 390 image rotated 30° clockwise. Blank regions have been filled with black. The center of rotation is  $(x_0, y_0) = (260, 195)$ .



**Figure 18 Arbitrary Rotation**

## 15 sip\_shear

### Parameters

```
#include "ipl.h"
ipl_status_type sip_shear (
    ipl_image_type*   in_img_ptr,
    ipl_image_type*   out_img_ptr,
    int32             alpha,
    int32             beta,
    uint16            fillerPixel
);
```

→	in_img_ptr	Pointer to the input image
↔	out_img_ptr	Pointer to the output image
→	alpha	Horizontal shear factor
→	beta	Vertical shear factor
→	fillerPixel	Pixel value used to fill in blank regions outside sheared image
←	status	Status returned at the end of the operation. Valid values are: IPL_SUCCESS – Operation completed successfully IPL_FAILURE – Failed to complete because of an error

### Description

This function performs shearing. Horizontal and vertical shear factors, alpha and beta, can be any integer – positive, negative, or zero. Input and output images must be the same size.

Shearing produces an output shaped like a parallelogram. When at least one of the shear factors is equal to 0, the resulting parallelogram is wrapped around the output image to fit inside image boundaries. If both shear factors are nonzero, the parallelogram is downsized to fit inside image boundaries. In this case, since images are stored as rectangular objects, the user must specify a pixel value used to fill in blank output regions not occupied by the parallelogram. As in `sip_perspective`, the pixel value must be in a format that is compatible with the output color format. For instance, black is 0 for RGB565 and 0x8010 for YCbCr 4:2:2.

Figure 19 shows an example of combined horizontal and vertical shearing with `alpha = 3` and `beta = 3`. The parallelogram-shaped result is downsized to fit inside output image boundaries. Blank output regions are filled with black.

Figure 20 shows an example of strictly horizontal shearing with `alpha = -5` and `beta = 0`. Since one of the shear factors is equal to zero, the resulting parallelogram is wrapped around the output image.



Figure 19 Shear,  $\alpha = 3$ ,  $\beta = 3$



Figure 20 Shear,  $\alpha = -5$ ,  $\beta = 0$

## 16 sip\_overlap

### Parameters

```
#include "ipl.h"
ipl_status_type sip_overlap (
    ipl_image_type*   in1_img_ptr,
    ipl_image_type*   in2_img_ptr,
    ipl_image_type*   out_img_ptr,
    ipl_rect_type*    overlap
);
```

→	in1_img_ptr	Pointer to the first input image (background)
→	in2_img_ptr	Pointer to the second input image (foreground)
↔	out_img_ptr	Pointer to the output image
→	overlap	Overlap region
←	status	Status returned at the end of the operation. Valid values are: IPL_SUCCESS – Operation completed successfully IPL_FAILURE – Failed to complete because of an error

### Description

This function overlaps one image on top of another. The first input image becomes the background and the second input becomes the foreground. The overlap region is rectangular and can be anywhere within the background image. The foreground image is automatically resized to fit the overlap region.

Figure 21 shows an example of overlap where the two input images are 398 x 298 pixels. The overlap region is 100 x 120 pixels and has an upper left corner located at  $(x, y) = (10, 269)$ . The coordinate system is defined with the origin at the lower left corner of the output image, the x axis pointing to the right, and the y axis pointing upward.



**Figure 21** Overlap

## 17 sip\_compose\_horizontal

### Parameters

```
#include "ipl.h"
ipl_status_type sip_compose_horizontal (
    ipl_image_type*    in1_img_ptr,
    ipl_image_type*    in2_img_ptr,
    ipl_image_type*    out_img_ptr,
    uint32             boundary
);
```

→	in1_img_ptr	Pointer to the first input image
→	in2_img_ptr	Pointer to the second input image
↔	out_img_ptr	Pointer to the output image
→	boundary	Boundary at which the second input image starts
←	status	Status returned at the end of the operation. Valid values are: IPL_SUCCESS – Operation completed successfully IPL_FAILURE – Failed to complete because of an error

### Description

This function combines two images by taking the left part of the first image and the right part of the second image. Input and output images must have the same size. The `boundary` variable specifies where the second image starts.

Examples:

1. If `boundary = 0`, `output = second image only`
2. If `boundary = output width`, `output = first image only`

Figure 22 shows an example where the `boundary` is located at `x = 199` in a `398 x 298` output image. This allows the left half of the first input and the right half of the second input to be displayed. The coordinate system is defined with the origin located at the lower left corner of the output image, the `x` axis pointing to the right, and the `y` axis pointing upward.



Figure 22 Horizontal Composite



## 18 sip\_compose\_vertical

### Parameters

```
#include "ipl.h"
ipl_status_type sip_compose_vertical (
    ipl_image_type*    in1_img_ptr,
    ipl_image_type*    in2_img_ptr,
    ipl_image_type*    out_img_ptr,
    uint32             boundary
);
```

→	in1_img_ptr	Pointer to the first input image
→	in2_img_ptr	Pointer to the second input image
↔	out_img_ptr	Pointer to the output image
→	boundary	Boundary at which the second input image starts
←	status	Status returned at the end of the operation. Valid values are: IPL_SUCCESS – Operation completed successfully IPL_FAILURE – Failed to complete because of an error

### Description

This function combines two images by taking the top part of the first image and the bottom part of the second image. Input and output images must have the same size. The `boundary` variable specifies where the second image starts.

Examples:

1. If `boundary = 0`, output = second image only
2. If `boundary = output width`, output = first image only

Figure 23 shows a sample 398 x 298 output image where the boundary is at `y = 149`, which is half the output height. This allows the top half of the first input and the bottom half of the second input to be displayed. The coordinate system is defined with the origin located at the lower left corner of the output image, the x axis pointing to the right, and the y axis pointing upward.



**Figure 23 Vertical Composite**

## 19 sip\_center

### Parameters

```
#include "ipl.h"
ipl_status_type sip_center (
    ipl_image_type*    in1_img_ptr,
    ipl_image_type*    in2_img_ptr,
    ipl_image_type*    out_img_ptr,
    uint32             width
);
```

→	in1_img_ptr	Pointer to the first input image
→	in2_img_ptr	Pointer to the second input image
↔	out_img_ptr	Pointer to the output image
→	width	Width of the center part taken from the first input
←	status	Status returned at the end of the operation. Valid values are: IPL_SUCCESS – Operation completed successfully IPL_FAILURE – Failed to complete because of an error

### Description

This function combines two images by taking the center part of the first image and the left and right parts of the second image. Input and output images must be the same size. The `width` indicates the width of the center image and must be even for subsampled YCbCr images.

Examples:

1. If `width = 0`, output = second image only
2. If `width = output width`, output = first image only
3. If `width = half of output width`, output = center half of first image with rest of second image

Figure 24 shows a sample 398 x 298 output image where the width of the center part is 199 pixels, which is half of the output width. This allows the center half of the first input image and the first quarter and last quarter of the second input to be displayed. The coordinate system used here has an origin at the lower left corner of the output image, an x axis pointing to the right, and a y axis pointing upward.



Figure 24 Center

## 20 sip\_film\_strip

### Parameters

```
#include "ipl.h"
ipl_status_type sip_film_strip (
    ipl_image_type*    in1_img_ptr,
    ipl_image_type*    in2_img_ptr,
    ipl_image_type*    out_img_ptr,
    uint32             boundary
);
```

→	in1_img_ptr	Pointer to the first input image
→	in2_img_ptr	Pointer to the second input image
↔	out_img_ptr	Pointer to the output image
→	boundary	Boundary at which the second input image starts
←	status	Status returned at the end of the operation. Valid values are: IPL_SUCCESS – Operation completed successfully IPL_FAILURE – Failed to complete because of an error

### Description

This function combines the beginning part of the first input and the end part of the second input to simulate the effect of a moving film strip. Input and output images must have the same size. The `boundary` variable specifies where the second image starts.

Examples:

1. If `boundary = 0`, output = second image only
2. If `boundary = output width`, output = first image only

Figure 25 shows a sample 520 x 390 output image with a boundary located at `x = 275`. The coordinate system used here has an origin located at the lower left corner of the output image, an `x` axis pointing to the right, and a `y` axis pointing upward.



**Figure 25 Film Strip**

## 21 sip\_corner\_fold

### Parameters

```
#include "ipl.h"
ipl_status_type sip_corner_fold (
    ipl_image_type *   in1_img_ptr,
    ipl_image_type *   in2_img_ptr,
    ipl_image_type *   out_img_ptr,
    uint8              position
);
```

→	in1_img_ptr	Pointer to the first input image
→	in2_img_ptr	Pointer to the second input image
↔	out_img_ptr	Pointer to the output image
→	position	Corner position (top left = 0, top right = 1, bottom left = 2, bottom right = 3)
←	status	Status returned at the end of the operation. Valid values are: IPL_SUCCESS – Operation completed successfully IPL_FAILURE – Failed to complete because of an error

### Description

This function folds over a corner of an image. The folded corner can be in one of 4 positions: top left corner (0), top right corner (1), bottom left corner (2), or bottom right corner (3). The first input image is the image whose corner is folded over. The second input image is behind the first input and is partially revealed by the folded corner.

Figure 26 shows two sample input images. Figure 28 shows four sample corner fold images.



**Figure 26 Input Images**



**Figure 27 Corner Fold**

## 22 sip\_color\_change

### Parameters

```
#include "ipl.h"
ipl_status_type sip_color_change (
    ipl_image_type *   in_img_ptr,
    ipl_image_type *   out_img_ptr,
    uint8              option
);
```

→	in_img_ptr	Pointer to the input image
↔	out_img_ptr	Pointer to the output image
→	option	Color change option (range: 0 to 4)
←	status	Status returned at the end of the operation. Valid values are: IPL_SUCCESS – Operation completed successfully IPL_FAILURE – Failed to complete because of an error

### Description

This function changes the color scheme of an image. There are 5 different color change options, each producing a different change in the color scheme. Options are numbered from 0 to 4.

Figure 28 shows a sample color change result when `option = 0`.



**Figure 28 Color Change**