
Optimizing Image Processing With vImage



2005-04-29



Apple Computer, Inc.
© 2003, 2005 Apple Computer, Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Computer, Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Computer, Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled or Apple-licensed computers.

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Apple, the Apple logo, Carbon, Cocoa, Mac, Mac OS, Macintosh, Panther, and Velocity Engine are trademarks of Apple Computer, Inc., registered in the United States and other countries.

Tiger is a trademark of Apple Computer, Inc.

PowerPC and the PowerPC logo are trademarks of International Business Machines Corporation, used under license therefrom.

Simultaneously published in the United States and Canada.

Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Introduction	Introduction To vImage 15
	About This Document 15
	What Is vImage? 15
	Who Should Read This Document 16
	Organization of This Document 16
Chapter 1	vImage Basics 17
	Data Types Used by vImage 17
	vImage_Buffer 17
	Pixel Types 19
	Flags 20
	Pixels Outside the Image Buffer 21
	64-Bit Processing 21
Chapter 2	Performance Issues 23
	Tiling for Cache Utilization 23
	Multiprocessing 23
	Real-time Processing 24
	Planar Versus Interleaved 24
	Buffer Size and Alignment 25
	Temporary Buffers for vImage Functions 26
Chapter 3	Convolution Operations 29
	What Is Convolution? 29
	Kernels for Convolution 30
	Normalization 30
	Size Limits 30
	Edge Conditions 30
	Convolution Functions 30
	Convolve 31
	Convolve with Bias 35
	Convolve with Multiple Kernels 36
	High-Speed Box and Tent Filters 37
	vImageGetMinimumTempBufferSizeForConvolution 39

	Deconvolution Functions 39
	Richardson-Lucy Deconvolution Functions 40
Chapter 4	Morphological Operations 43
	What Are Morphological Operations? 43
	Morphological Operations on Grayscale Images 43
	Morphological Operations for Full-color Images 44
	Kernels for Morphological Operations 45
	Edge Conditions for Morphology Functions 45
	Morphological Functions 45
	Dilate 45
	Erode 46
	Max 46
	Min 47
	vImageGetMinimumTempBufferSizeForMinMax 48
Chapter 5	Geometric Operations 49
	What Are Geometric Operations? 49
	Resampling 49
	Shear Functions and User-defined Resampling Filters 50
	Affine Transformations 51
	Edge Conditions 52
	Geometric Functions 52
	High-Level Geometric Functions 52
	Low-level Geometric Functions 57
Chapter 6	Histogram Operations 67
	What Are Histograms? 67
	Histograms for Images in Integer Formats 67
	Histograms For Images In Floating-Point Formats 67
	Histogram Operations 68
	Lookup Tables (LUTs) 69
	Histogram Functions 69
	Histogram Calculation 69
	Equalization 70
	Histogram Specification 72
	Contrast Stretch 73
	Ends-In Contrast Stretch 73
	vImageGetMinimumBufferSizeForHistogram 76
Chapter 7	Alpha Compositing Operations 77
	What Is Alpha Compositing? 77

Premultiplied Alpha Versus Non-premultiplied Alpha	77
Alpha Compositing Functions	78
Non-premultiplied Alpha Blend	78
Premultiplied Alpha Blend	80
Non-premultiplied to Premultiplied Alpha Blend	81
Premultiplied Constant Alpha Blend	82
Unpremultiply Data	84
Premultiply Data	85
Clip Color Values to Alpha	85

Chapter 8 Image Transformation Operations 87

What Are Image Transformations?	87
Matrix Multiplication Functions	87
Gamma Correction	89
Types of Gamma Functions	89
Creating and Destroying Gamma Function Objects	90
Gamma Correction Functions	90
Piecewise Polynomial Functions	91
Piecewise Rational Function	92
Lookup Table Functions	93

Chapter 9 Conversion Operations 95

What Is Format Conversion?	95
Conversion Functions	95
vImageBufferFill	95
vImageOverwriteChannels	96
vImageOverwriteChannelsWithScalar	96
vImagePermuteChannels	97
vImage_Flatten	97
vImageClip_PlanarF	97
vImageConvert_Planar8ToPlanarF	98
vImageConvert_PlanarFToPlanar8	98
vImageConvert_Planar8toARGB8888	99
vImageConvert_ARGB8888toPlanar8	99
vImageConvert_PlanarFtoARGBFFFF	100
vImageConvert_ARGBFFFFtoPlanarF	100
vImageConvert_16SToF	101
vImageConvert_16UToF	101
vImageConvert_16UToPlanar8	102
vImageConvert_FTo16S	102
vImageConvert_FTo16U	103
vImageConvert_Planar8To16U	104
vImageConvert_Planar16FtoPlanarF	104
vImageConvert_PlanarFtoPlanar16F	105

vImageConvert_ARGB1555toARGB8888 105
vImageConvert_ARGB8888toARGB1555 106
vImageConvert_ARGB1555toPlanar8 106
vImageConvert_Planar8toARGB1555 107
vImageConvert_RGB565toARGB8888 107
vImageConvert_ARGB8888toRGB565 108
vImageConvert_RGB565toPlanar8 109
vImageConvert_Planar8toRGB565 109
vImageConvert_RGB888toARGB8888 110
vImageConvert_ARGB8888toRGB888 110
vImageConvert_Planar8toRGB888 111
vImageConvert_RGB888toPlanar8 111
vImageConvert_PlanarFtoRGBFFF 111
vImageConvert_RGBFFFtoPlanarF 112
vImageTableLookUp_Planar8 112
vImageTableLookUp_ARGB8888 113
General Conversions 113
Convert Chunky to Planar 114
Convert Planar to Chunky 115

Chapter 10 vImage Functions 117

Alpha Compositing Functions 117
vImageAlphaBlend_ARGBFFFF 117
vImageAlphaBlend_ARGB8888 118
vImageAlphaBlend_PlanarF 118
vImageAlphaBlend_Planar8 120
vImagePremultipliedAlphaBlend_ARGBFFFF 121
vImagePremultipliedAlphaBlend_ARGB8888 122
vImagePremultipliedAlphaBlend_PlanarF 122
vImagePremultipliedAlphaBlend_Planar8 123
vImagePremultipliedConstAlphaBlend_ARGBFFFF 124
vImagePremultipliedConstAlphaBlend_ARGB8888 125
vImagePremultipliedConstAlphaBlend_PlanarF 126
vImagePremultipliedConstAlphaBlend_Planar8 127
vImageAlphaBlend_NonpremultipliedToPremultiplied_ARGBFFFF 128
vImageAlphaBlend_NonpremultipliedToPremultiplied_ARGB8888 128
vImageAlphaBlend_NonpremultipliedToPremultiplied_PlanarF 129
vImageAlphaBlend_NonpremultipliedToPremultiplied_Planar8 130
vImagePremultiplyData_ARGBFFFF 131
vImagePremultiplyData_RGBAFFFF 131
vImagePremultiplyData_ARGB8888 132
vImagePremultiplyData_RGBA8888 133
vImagePremultiplyData_PlanarF 133
vImagePremultiplyData_Planar8 134
vImageUnpremultiplyData_ARGBFFFF 135

vImageUnpremultiplyData_RGBAFFFF	136
vImageUnpremultiplyData_ARGB8888	136
vImageUnpremultiplyData_RGBA8888	137
vImageUnpremultiplyData_PlanarF	138
vImageUnpremultiplyData_Planar8	138
Conversion Functions	139
vImageBufferFill_ARGB8888	139
vImageBufferFill_ARGBFFFF	140
vImageClip_PlanarF	140
vImageConvert_16SToF	141
vImageConvert_16UToF	142
vImageConvert_16UtoPlanar8	143
vImageConvert_ARGB1555toARGB8888	143
vImageConvert_ARGB1555toPlanar8	144
vImageConvert_ARGB8888toARGB1555	145
vImageConvert_ARGB8888toPlanar8	146
vImageConvert_ARGB8888toRGB565	147
vImageConvert_ARGB8888toRGB888	147
vImageConvert_ARGBFFFFtoPlanarF	148
vImageConvert_ChunkyToPlanar8	149
vImageConvert_ChunkyToPlanarF	150
vImageConvert_FTo16S	151
vImageConvert_FTo16U	152
vImageConvert_Planar16FtoPlanarF	153
vImageConvert_Planar8to16U	154
vImageConvert_Planar8toARGB1555	154
vImageConvert_Planar8toARGB8888	155
vImageConvert_Planar8toPlanarF	156
vImageConvert_Planar8toRGB565	157
vImageConvert_Planar8toRGB888	158
vImageConvert_PlanarFtoRGBFFF	159
vImageConvert_PlanarFtoARGBFFFF	160
vImageConvert_PlanarFtoPlanar16F	161
vImageConvert_PlanarFtoPlanar8	161
vImageConvert_PlanarToChunky8	162
vImageConvert_PlanarToChunkyF	163
vImageConvert_RGB565toPlanar8	165
vImageConvert_RGB565toARGB8888	166
vImageConvert_RGB888toARGB8888	166
vImageConvert_RGB888toPlanar8	167
vImageConvert_RGBFFFtoPlanarF	168
vImage_FlattenARGB8888ToRGB888	169
vImage_FlattenARGBFFFFToRGBFFF	170
vImageOverwriteChannels_ARGB8888	171
vImageOverwriteChannels_ARGBFFFF	171
vImageOverwriteChannelsWithScalar_ARGB8888	172

vImageOverwriteChannelsWithScalar_ARGBFFFF	173
vImageOverwriteChannelsWithScalar_Planar8	174
vImageOverwriteChannelsWithScalar_PlanarF	174
vImagePermuteChannels_ARGB8888	175
vImagePermuteChannels_ARGBFFFF	176
vImageTableLookUp_ARGB8888	176
vImageTableLookUp_Planar8	177
Convolution Functions	178
vImageConvolve_ARGBFFFF	178
vImageConvolve_ARGB8888	180
vImageConvolve_PlanarF	183
vImageConvolve_Planar8	184
vImageConvolveWithBias_ARGBFFFF	187
vImageConvolveWithBias_ARGB8888	189
vImageConvolveWithBias_PlanarF	191
vImageConvolveWithBias_Planar8	193
vImageConvolveMultiKernel_ARGBFFFF	195
vImageConvolveMultiKernel_ARGB8888	197
vImageBoxConvolve_Planar8	199
vImageBoxConvolve_ARGB8888	201
vImageTentConvolve_Planar8	203
vImageTentConvolve_ARGB8888	205
vImageGetMinimumTempBufferSizeForConvolution	207
vImageRichardsonLucyDeConvolve_ARGBFFFF	208
vImageRichardsonLucyDeConvolve_ARGB8888	211
vImageRichardsonLucyDeConvolve_PlanarF	213
vImageRichardsonLucyDeConvolve_Planar8	215
Image Transformation Functions: Matrix Multiplication	218
vImageMatrixMultiply_Planar8	218
vImageMatrixMultiply_PlanarF	219
vImageMatrixMultiply_ARGB8888	220
vImageMatrixMultiply_ARGBFFFF	221
Image Transformation Functions: Gamma Corrections	222
vImageCreateGammaFunction	222
vImageDestroyGammaFunction	224
vImageGamma_Planar8ToPlanarF	224
vImageGamma_PlanarFToPlanar8	225
vImageGamma_PlanarF	225
vImagePiecewisePolynomial_PlanarF	226
vImagePiecewisePolynomial_Planar8ToPlanarF	227
vImagePiecewisePolynomial_PlanarFToPlanar8	228
vImageLookupTable_Planar8ToPlanarF	229
vImageLookupTable_PlanarFToPlanar8	229
vImageInterpolatedLookupTable_PlanarF	230
Histogram Functions	231
vImageContrastStretch_ARGBFFFF	231

vImageContrastStretch_ARGB8888	233
vImageContrastStretch_PlanarF	233
vImageContrastStretch_Planar8	235
vImageEndsInContrastStretch_ARGBFFFF	235
vImageEndsInContrastStretch_ARGB8888	237
vImageEndsInContrastStretch_PlanarF	238
vImageEndsInContrastStretch_Planar8	240
vImageEqualization_ARGBFFFF	241
vImageEqualization_ARGB8888	242
vImageEqualization_PlanarF	243
vImageEqualization_Planar8	244
vImageGetMinimumTempBufferSizeForHistogram	245
vImageHistogramCalculation_ARGBFFFF	246
vImageHistogramCalculation_ARGB8888	247
vImageHistogramCalculation_PlanarF	248
vImageHistogramCalculation_Planar8	248
vImageHistogramSpecification_ARGBFFFF	249
vImageHistogramSpecification_ARGB8888	251
vImageHistogramSpecification_PlanarF	251
vImageHistogramSpecification_Planar8	253
Morphological Functions	254
vImageDilate_ARGBFFFF	254
vImageDilate_ARGB8888	255
vImageDilate_PlanarF	256
vImageDilate_Planar8	257
vImageErode_ARGBFFFF	258
vImageErode_ARGB8888	259
vImageErode_PlanarF	261
vImageErode_Planar8	262
vImageGetMinimumTempBufferSizeForMinMax	263
vImageMax_ARGBFFFF	264
vImageMax_ARGB8888	265
vImageMax_PlanarF	267
vImageMax_Planar8	268
vImageMin_ARGBFFFF	270
vImageMin_ARGB8888	271
vImageMin_PlanarF	272
vImageMin_Planar8	274
Geometric Functions	275
vImageAffineWarp_ARGBFFFF	275
vImageAffineWarp_ARGB8888	277
vImageAffineWarp_PlanarF	278
vImageAffineWarp_Planar8	280
vImageDestroyResamplingFilter	281
vImageHorizontalReflect_ARGBFFFF	282
vImageHorizontalReflect_PlanarF	282

vImageHorizontalReflect_Planar8 283
vImageHorizontalReflect_ARGB8888 284
vImageHorizontalShear_ARGBFFFF 284
vImageHorizontalShear_ARGB8888 286
vImageHorizontalShear_PlanarF 287
vImageHorizontalShear_Planar8 289
vImageGetMinimumGeometryTempBufferSize 291
vImageGetResamplingFilterSize 292
vImageNewResamplingFilter 292
vImageNewResamplingFilterForFunctionUsingBuffer 293
vImageRotate90_ARGBFFFF 295
vImageRotate90_ARGB8888 296
vImageRotate90_PlanarF 297
vImageRotate90_Planar8 298
vImageRotate_ARGBFFFF 299
vImageRotate_ARGB8888 301
vImageRotate_PlanarF 302
vImageRotate_Planar8 304
vImageScale_ARGBFFFF 305
vImageScale_ARGB8888 306
vImageScale_PlanarF 308
vImageScale_Planar8 309
vImageVerticalReflect_ARGBFFFF 310
vImageVerticalReflect_ARGB8888 311
vImageVerticalReflect_PlanarF 311
vImageVerticalReflect_Planar8 312
vImageVerticalShear_ARGBFFFF 313
vImageVerticalShear_ARGB8888 314
vImageVerticalShear_PlanarF 316
vImageVerticalShear_Planar8 318

Chapter 11 vImage Callback Functions 321

MyResamplingFilter 321

Chapter 12 vImage Data Types 323

vImage_Buffer 323
vImage_AffineTransform 324
vImage_Error 324
vImage_Flags 324
Pixel_8 324
Pixel_F 324
Pixel_8888 324
Pixel_FFFF 325
ResamplingFilter 325

C O N T E N T S

Chapter 13 vImage Constants 327

vImage Error Codes 327
vImage Flags 328
Rotation Constants for Use With Rotate90 Function 329

Document Revision History 331

Index 333

C O N T E N T S

Figures and Listings

Chapter 1	vImage Basics	17
	Figure 1-1	A vImage_Buffer 18
	Figure 1-2	A region of interest within a vImage_Buffer 19
Chapter 2	Performance Issues	23
	Listing 2-1	Initializing and freeing a vImage_Buffer structure 25
Chapter 3	Convolution Operations	29
	Figure 3-1	Facade unmodified 32
	Figure 3-2	Facade blurred by convolution 33
	Figure 3-3	Facade embossed by convolution 34
	Figure 3-4	Image before sharpening 34
	Figure 3-5	image sharpened by convolution 35
Chapter 5	Geometric Operations	49
	Figure 5-1	Bluejay, unmodified 53
	Figure 5-2	Bluejay, rotated 54
	Figure 5-3	Bluejay, after affine warp. 56
	Figure 5-4	Bluejay, after a horizontal reflection 57
	Figure 5-5	Bluejay, after vertical reflection 58
	Figure 5-6	Bluejay, after horizontal shear 61
	Figure 5-7	Bluejay, after vertical shear 63
Chapter 6	Histogram Operations	67
	Figure 6-1	Image before histogram equalization 71
	Figure 6-2	Image after histogram equalization 71
	Figure 6-3	Space shuttle before ends-in contrast stretch 75
	Figure 6-4	Space shuttle after ends-in contrast stretch 75

F I G U R E S A N D L I S T I N G S

Introduction To vImage

Framework: vImage

About This Document

This document, *Optimizing Image Processing with vImage*, describes the use of vImage, Apple’s image processing framework. vImage includes functions for the manipulation of images, as well as utility functions for format conversions and other operations.

You can check <http://developer.apple.com/> for information about updates to this and other developer documents. To receive notification of documentation updates, you can sign up for ADC’s free Online Program and receive their weekly Apple Developer Connection News email newsletter. (See <http://developer.apple.com/membership/> for more details about the Online Program.)

What Is vImage?

vImage is Apple’s image processing framework. It includes high-level functions for image manipulation—convolutions, geometric transformations, histogram operations, morphological transformations, and alpha compositing—as well as utility functions for format conversions and other operations.

vImage contains vectorized code to make use of the Velocity Engine processing unit, when available, for high performance. All functions also contain scalar code that performs identical operations when the Velocity Engine processing unit is not available, such as on a Macintosh G3. The framework uses the best code for the hardware it is running on, in a manner completely transparent to the calling application.

vImage can be called from Carbon, Cocoa, and CLI applications, but not directly from the kernel. It is included as part of Panther, and of future Mac OS X releases.

Who Should Read This Document

You should read this document if you want to use the vImage library for high-speed image processing in your application. You should be familiar with Macintosh application development and the basics of image representation and manipulation.

Organization of This Document

This document contains the following sections:

[“Introduction To vImage”](#) (page 15). Introduces the vImage image processing library, describes the document organization.

Chapter 1, [“vImage Basics”](#) (page 17). Discusses data structures used by vImage, regions of interest, pixels outside the image buffer.

Chapter 2, [“Performance Issues”](#) (page 23). Discusses image tiling, real-time and multiprocessing issues, optimal use of buffers.

Chapter 3, [“Convolution Operations”](#) (page 29). Defines convolution, discusses convolution kernels and normalization, describes the vImage convolution functions.

Chapter 4, [“Morphological Operations”](#) (page 43). Defines morphological operations, discusses kernels for morphological operations, describes the vImage morphology functions.

Chapter 5, [“Geometric Operations”](#) (page 49). Defines geometric operations, discusses resampling and resampling filters, describes the vImage high-level and low-level geometric functions.

Chapter 6, [“Histogram Operations”](#) (page 67). Defines histograms, histogram operations, and look-up tables, describes the vImage histogram functions.

Chapter 7, [“Alpha Compositing Operations”](#) (page 77). Defines alpha compositing, discusses premultiplied and non-premultiplied alpha, describes the vImage alpha compositing functions.

Chapter 8, [“Image Transformation Operations”](#) (page 87). Defines alpha compositing, discusses premultiplied and non-premultiplied alpha, describes the vImage alpha compositing functions.

Chapter 9, [“Conversion Operations”](#) (page 95). Describes the vImage functions for format conversions.

Reference sections for the vImage functions, callbacks, constants, and data types.

vImage Basics

This chapter discusses the data types used by vImage and how the vImage calls assign pixel values to pixel locations outside the bounds of an image. It also discusses 64-bit processing in vImage.

Data Types Used by vImage

vImage_Buffer

The principal data type used by vImage is `vImage_Buffer`. A `vImage_Buffer` structure represents an image in one of four formats:

Planar8: the image is a single channel (one color or alpha value). Each pixel is an 8-bit unsigned value, representing intensity levels from 0 to 255, inclusive. 255 is full intensity (white). 0 is no intensity (black).

PlanarF: the image is a single channel (one color). Each pixel is a 32-bit floating-point value, representing an intensity level. For most images the values should range from 0.0f (lowest possible intensity) to 1.0f (highest possible intensity). However, vImage does not enforce this range restriction, and does not clip calculated values that lie outside this range.

ARGB8888: the image has four channels, for Alpha, Red, Green, and Blue, in that order. Each pixel is 32 bits—an array of four 8-bit unsigned integers—representing each of the channels as an 8-bit number.

ARGBFFFF: the image has four channels, for Alpha, Red, Green, and Blue, in that order. Each pixel is an array of four floating-point numbers, representing each of the channels as a floating-point value. As with PlanarF, these values are often in the range 0.0f to 1.0f, but vImage does not enforce this convention.

In addition, the `vImagePremultiplyData_RGBA8888`, `vImagePremultiplyData_RGBAFFF`, `vImageUnpremultiplyData_RGBA8888`, and `vImageUnpremultiplyData_RGBAFFF` functions accept images in RGBA8888 and RGBAFFF formats. (These formats are like the ARGB8888 and ARGBFFFF formats, but with the channels in a different order.) Please see “[Premultiply Data](#)” (page 84) and “[Unpremultiply Data](#)” (page 84).

Finally, there are conversion functions that convert these image formats to and from a number of other formats. Please see “[Conversion Operations](#)” (page 95).

The vImage_Buffer structure is defined as:

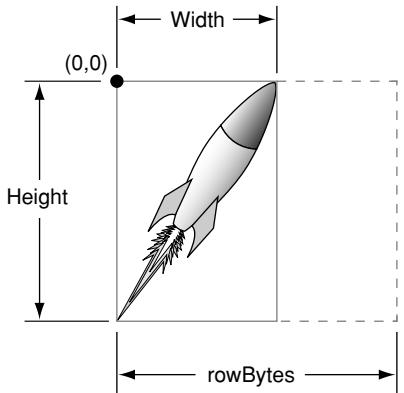
```
typedef unsigned long vImagePixelCount; /* Pedantic: A number of pixels. For
                                         LP64 (ppc64) this is a 64-bit quantity.*/

typedef struct vImage_Buffer
{
    void           *data; /* Pointer to the top left pixel of the buffer */
    vImagePixelCount height; /* The height (in pixels) of the buffer */
    vImagePixelCount width; /* The width (in pixels) of the buffer */
    size_t          rowBytes; /* The number of bytes in a pixel row */
} vImage_Buffer;
```

rowBytes can be chosen so that the pixel row extends beyond the last pixel in each row. This makes it possible to align the rows for best performance, and to create vImage_Buffer objects that refer to a smaller region within an image without copying or moving the pixel data.

Figure 1-1 illustrates a vImage_Buffer. height and width are measured in pixels, but rowBytes is measured in bytes. The pointer in the data field of the vImage_Buffer points to the pixel location labeled (0,0).

Figure 1-1 A vImage_Buffer



The vImage_Buffer can represent either an entire captured image or some smaller rectangular region within that image. To refer to a smaller region of your original image, you can:

- Decrease height, so that some pixel rows at the bottom of the image are ignored
- Decrease width, so that some pixel columns at the left side of the image are ignored
- Change the data pointer so that it points to a pixel other than the upper left pixel of the original image. (This requires that you also change height and width so that your new region does not extend past the edges of the data buffer.)

rowBytes should remain unchanged.

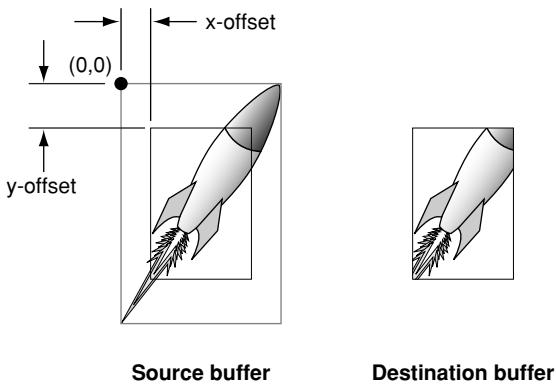
When you pass a vImage_Buffer to a vImage function, vImage does not attempt to read pixels outside of the buffer's data. They are considered undefined. If you pass a vImage_Buffer as a destination buffer, vImage does not write any pixels outside the buffer's data.

Whenever you pass a vImage_Buffer as a destination buffer, you are responsible for setting the fields of the vImage_Buffer structure and allocating the memory for the data. When you are done with the structure, you are responsible for deallocating the memory for the data.

Sometimes it is useful to refer to a **region of interest** within your image but also maintain the information defining the original image, so that a function can read pixels beyond the edges of the region of interest and know which of them are valid. For this reason, some vImage functions accept as input a vImage_Buffer along with two values, *srcOffsetToROI_X* and *srcOffsetToROI_Y*. These values represent the offset in pixels (in the horizontal and vertical directions, respectively) from the upper left pixel of the vImage_Buffer to the upper left-hand pixel of the region of interest. (The region of interest is smaller or, at most, the same size as the original image.) The height and width of the region of interest are determined by the function, using other inputs. For example, for most functions, the height and width of the region of interest must be equal to the height and width of a destination buffer passed to the function. In this case the function can get the height and width of the region of interest from the corresponding values of the destination buffer.

Figure 1-2 illustrates a region of interest within a vImage_Buffer. The pointer in the data field of the vImage_Buffer still points to the pixel location (0,0), not to the upper-left pixel location of the region of interest. The region of interest and the destination buffer are the same size. In this illustration, for simplicity's sake, no image transformation has been applied to the source image. The portion of the source image within the region of interest has simply been copied over to the destination image.

Figure 1-2 A region of interest within a vImage_Buffer



Pixel Types

vImage defines the following pixel types for its four image types.

```
typedef uint8_t      Pixel_8;        /* 8 bit planar pixel value */
typedef float       Pixel_F;        /* floating point planar pixel value
                                         */
typedef uint8_t      Pixel_8888[4]; /* ARGB interleaved (8 bit/channel) pixel
                                         value. uint8_t[4] = { alpha, red, green, blue } */
typedef float       Pixel_FFFF[4]; /* ARGB interleaved (floating point) pixel
                                         value. float[4] = { alpha, red, green, blue } */
```

Flags

vImage defines the following flags. Every vImage function accepts a flags parameter, but only certain flags are used by each function.

```
typedef uint32_t vImage_Flags;
/* vImage flags */
enum
{
    kvImageNoFlags          = 0,
    kvImageLeaveAlphaUnchanged = 1, /* Operate on red, green and blue
                                    channels only. Alpha is copied from source to destination. For Interleaved
                                    formats only. */
    kvImageCopyInPlace       = 2, /* Copy edge pixels */
    kvImageBackgroundColorFill = 4, /* Use a background color. */
    kvImageEdgeExtend         = 8, /* Extend border data elements */
    kvImageDoNotTile         = 16, /* Pass to turn off internal tiling.
                                    Use this if you want to do your own tiling, or to use the Min/Max filters in
                                    place. */
    kvImageHighQualityResampling= 32 /* Use a higher quality, slower
                                    resampling filter for Geometry operations (shear, scale, rotate, affine
                                    transform, etc.) */

    kvImageTruncateKernel     = 64 /* Use only the part of
                                    the kernel that
                                    overlaps the image. For integer kernels, real_divisor = divisor
                                    * (sum of used kernel
                                    elements) / (sum of kernel elements). This should preserve image
                                    brightness at the
                                    edges. Convolution only. */
    kvImageHighGetTempBufferSize= 128 /* The function will return the number
                                    of bytes required for the temp buffer. If this value is negative, it is an
                                    error, per standard usage. */
};
```

You can pass multiple flags to a function by adding the flag values together. For example, to leave alpha unchanged and turn off tiling, you can pass the flag value `kvImageLeaveAlphaUnchanged + kvImageDoNotTile`.

Three of the flags are mutually exclusive: `kvImageCopyInPlace`, `kvImageBackgroundColorFill`, and `kvImageEdgeExtend`. Never pass more than one of these flag values in the same flag parameter.

When passing flags to a function, do not set values for flags that are not used by the function. If the function requires you to set certain flag values, do so. (For example, for the convolution function, you must set exactly one of `kvImageCopyInPlace`, `kvImageBackgroundColorFill`, and `kvImageEdgeExtend`.) Otherwise the function may return an error. These details are documented with each individual function. If you want to set no flag values, pass `kvImageNoFlags`.

Pixels Outside the Image Buffer

At certain times, some vImage image transformation functions—particularly the convolution, morphology, and geometry functions—need pixel values for pixel locations that are outside the image buffer being used. For example, when processing certain image pixels, a convolution kernel may extend partially beyond the edges of the image. Or, a rotation function may need to “rotate in” pixels that are not in the original image.

vImage handles this problem of “nonexistent pixels” in four different ways, controlled by flags (see [“Flags”](#) (page 19)) passed to the vImage functions:

- **Background color fill:** the caller supplies a background color (that is, a pixel value). If the `kvImageBackgroundColorFill` flag is set, all pixels outside the image are assigned this value.
- **Edge extend:** if the `kvImageEdgeExtend` flag is set, the edges of the image are replicated outward infinitely. The top row of the image is repeated infinitely up above the image, the bottom row infinitely down below the image, the first column is repeated infinitely often to the left of the image, and the last column infinitely often to the right. In the spaces that are “diagonally” off the image, the value of the corresponding corner pixel is used. For example, for all pixels that are both above and to the left of the image, the upper-leftmost pixel of the image (the pixel at row 0, column 0) supplies the value. In this way, every pixel location outside the image is assigned some value.
- **Copy in place (for convolutions only):** if the `kvImageCopyInPlace` flag is set, and the convolution is processing an image pixel for which some of the kernel extends beyond the image boundaries, the convolution is not computed; instead, the value of the particular pixel is used unchanged. This results in certain strips of pixels along the outer margins of the image being copied unchanged to the destination image.
- **Kernel truncation (for convolutions only):** if the `kvTruncateKernel` flag is set, only the part of the kernel that overlaps the image is used. The calculated pixel value is corrected by first multiplying by the sum of all the kernel elements, then dividing by the sum of the kernel elements that are actually used.

For the convolution functions, you can specify which of these four techniques you want vImage to use by setting a flag. The geometry functions only let you choose from the background color fill and edge extend techniques. (Some geometry functions do not offer a choice.)

The morphology functions do not use any of these techniques. Instead, they simply ignore pixels outside the source image, and do not use them in any of their calculations. See [“Edge Conditions for Morphology Functions”](#) (page 45).

64-Bit Processing

Mac OS X v10.4, Tiger, supports 64-bit addressing for those applications compiled for the ppc64 architecture. The vImage framework supports the ppc64 architecture, which means that all vImage APIs available in Mac OS X v10.4 and later are available for both 32-bit and 64-bit applications. 32-bit applications will continue to operate as always. For the ppc64 architecture, vImage accepts images with buffers larger than 4 Gigapixels wide or tall (or both) and passes data with 64 bit pointers.

Users writing shared source code that targets both the ppc and ppc64 architectures should be careful about their use of types with vImage. Some types in vImage change size between the two architectures, most notably `vImage_Error`, `size_t`, `vImagePixelCount`, anything with type `long` or `unsigned long`, and of course, pointers. These types are 64 bits for ppc64, and 32 bits for the ppc architecture. You should make sure that your own types grow and shrink accordingly to avoid truncation. Special care should be taken with data that might be transferred between architectures such as data stored to disk or sent over the network to make sure that truncation does not occur.

Performance Issues

This chapter discusses a number of different factors that can affect the performance of the vImage functions.

Tiling for Cache Utilization

In general, vImage functions have much better performance when the data they process (including input and output buffers) fit in the processor's data caches. For this reason it is best if the size of the data processed is less than 256 KB (or even less in some cases). However, many images are bigger than this. In this case, a common technique for achieving high performance is to operate on one small chunk, or "tile," of the image at a time. Perform all necessary operations on a given tile, then move on to the next.

Tiling (dividing your image into tiles) also makes it easier to parallelize your code to use multiple processors. Some vImage functions may be multithreaded internally; all of vImage is thread-safe and can be called reentrantly.

Many vImage functions use tiling and/or multithreading internally. If you want to do tiling or multithreading yourself, you should set the `kvImageDoNotTile` flag in the `flags` parameter when you call a function, which prevents the function from using tiling or multithreading itself.

Which functions do tiling or multithreading internally is subject to change from one release to another. If you are doing your own tiling or multithreading, you can probably improve performance by using the `kvImageDoNotTile` flag with all functions.

Multiprocessing

vImage is thread-safe and can be called reentrantly. If you tile your image, you can use separate threads for different tiles. It is recommended that if you use different processors to handle different tiles, you choose tiles that are not horizontally adjacent to each other. Otherwise the tile edges may share cachelines, potentially resulting in time-consuming cross talk between the two processors.

The state of a vImage output buffer is undefined while a vImage call is working on it. There may be times when the value of a pixel is neither the starting data or ending result, but the result of some intermediate calculation.

On Mac OS X v10.4 (Tiger) and later, some vImage functions are transparently multithreaded internally. They do their own parameter checking and only multithread in cases where it is expected that a performance benefit will be obtained. vImage maintains its own lazily allocated pool of threads to do this work. Thus, your code should just automatically be multithreaded without you doing anything for those vImage functions that have been internally multithreaded. (In Mac OS X v10.4.0, this is most functions in Geometry.h, and the gamma functionality, but this will likely grow.) The threads are not destroyed once created. They are reused. The calling thread may block while it is waiting for the secondary threads to finish their work. It is safe to call internally multithreaded APIs reentrantly.

While this is a win for the general case, in some cases, it is anticipated that there may be some applications that either do not benefit or cannot tolerate the locking required to multithread vImage functions internally. These applications should pass the kvImageDoNotTile flag to all vImage functions. This will signal vImage not to try to tile the image. Since tiling is required for multithreading, this will prevent vImage from multithreading within function calls, will prevent vImage from taking locks, and will prevent vImage from allocating additional threads.

Real-time Processing

In order to facilitate real-time usage, vImage avoids, as much as possible, use of the heap and other operations that block on a lock, such as memory allocation. Some functions require temporary buffers. You can provide these buffers yourself, often reusing the same buffer. If you do not provide a buffer, these functions will allocate memory for themselves (and, of course, deallocate it when they are finished); this can be time-consuming and interfere with real-time activities. For each group of functions that use temporary buffers, vImage provides a function that tells you how big the temporary buffer should be.

In Mac OS X 10.4 (Tiger) and later, some vImage functions are transparently multithreaded internally. This process use locks to maintain data coherency. If this is not desired, you may prevent vImage from multithreading and tiling by passing the kvImageDoNotTile flag. If you use this flag, your application will be responsible for tiling its own data and doing its own multithreading.

Planar Versus Interleaved

Most vImage functions come in four forms, depending on the format of the image data. The four formats are planar UInt8, planar floating point, ARGB interleaved (8 bits per channel), ARGB interleaved (floating point). The ARGB interleaved (8 bits per channel) format is identical to the traditional 32-bit pixel format used in Mac OS X.

A planar format is a single channel. In order to represent a full-color image, you would need four planar images, one each for the alpha, red, green, and blue channels. In an interleaved format, on the other hand, the data for each pixel contains the information for all four channels.

In most cases you will get significantly better performance by using a planar representation of your image, and performing the desired operations on each planar channel. Most vImage functions, when operating on an interleaved format, first separate the image into planar format; then operate on each of planar channel; then combine the results to create an interleaved result. By using a planar representation, you can avoid the cost of de-interleaving and re-interleaving in each function call. (In addition, in some cases you will not need to operate on the alpha channel, so you can skip that step.)

Buffer Size and Alignment

Buffer size and alignment can have significant effects on performance. Please note the following:

- All floating-point data must be aligned to 4-byte boundaries.
- vImage does not require that buffers be aligned in order to function or call vectorized code. However, for best performance, they should be 16-byte aligned. Especially for small images such as tiles, the difference can be dramatic: for images less than 64 bytes wide, 16-byte alignment can often make the difference between scalar and vector code, implying a speed difference of up to an order of magnitude.
- However, for images more than 256 bytes wide, alignment does not usually make a large difference in function performance.
- rowBytes should not be a power of 2. If it is, performance may be adversely affected for some functions. (This is a side effect of how some PowerPC machines handle address arithmetic internally.)
- For square tiles, a tile size of 128kB – 256kB gives the best overall throughput.
- Small (~16 kB) very wide (>256 bytes wide) rectangular tiles may in some cases outperform a 128kB – 256kB square tile. Typically this applies to functions that do not look at adjacent pixels to calculate the result.

Listing 2-1 contains a function for allocating a vImage_Buffer structure and a function for freeing one.

Listing 2-1 Initializing and freeing a vImage_Buffer structure

```
#include <stdlib.h>

vImage_Error MyInitBuffer( vImage_Buffer *result, int height, int width, size_t
bytesPerPixel )
{
    size_t rowBytes = width * bytesPerPixel;

    //Widen rowBytes out to a integer multiple of 16 bytes
    rowBytes = (rowBytes + 15) & ~15;

    //Make sure we are not an even power of 2 wide.
    //Will loop a few times for rowBytes <= 16.
    while( 0 == (rowBytes & (rowBytes - 1)) )
        rowBytes += 16;                      //grow rowBytes

    //Set up the buffer
    result->height = height;
    result->width = width;
    result->rowBytes = rowBytes;
    result->data = malloc( rowBytes * height );

    if (result->data == nil)
        return kvImageMemoryAllocationError;
    return kvImageNoError;
}
```

```

void MyFreeBuffer( vImage_Buffer *buffer )
{
    if( buffer && buffer->data )
        free( buffer->data );
}

```

Temporary Buffers for vImage Functions

Many of the vImage functions use temporary buffers to hold intermediate results while they process an image. You can supply the temporary buffer by passing a pointer to a buffer in the *tempBuffer* parameter of the function. Alternatively, if you pass `NULL` for the *tempBuffer* parameter, the function will allocate the temporary buffer itself.

If you are only going to call the function a small number of times, and the possibility of blocking on a lock for a short period of time is not a concern, it is sensible to let the function allocate the buffer itself. However, if you are going to call the function frequently, especially on images of the same size (for example, if you are tiling), or if real-time performance is an issue, then there are a number of advantages to allocating the buffer yourself and reusing it.

Allocating the buffer yourself gives you control over when and how often memory-allocating calls are made. It is faster to allocate the buffer once and use it in multiple function calls than it is to have each function call allocate and deallocate a buffer itself. In addition, memory-allocation calls may block on a lock, interfering with real-time performance. By allocating the buffer once yourself, you restrict that problem to a single call.

Previous to Mac OS X v10.4, a set of buffer-size functions were used to tell you how large the buffer must be—`vImageGetMinimumTempBufferSizeForConvolution`, `vImageGetMinimumGeometryTempBufferSize`, and so on. These functions are deprecated in Mac OS X v10.4.

Instead, to get the minimum buffer size for a particular vImage function that has a *tempBuffer* parameter, make a preliminary call to that function to get the minimum buffer size:

- Pass the `kvImageGetTempBufferSize` flag in the *flags* parameter.
- The *tempBuffer* parameter will be ignored.
- For the other parameters, pass the same values that you will pass in the “real” call.
- The function will return the minimum size as the function result (without doing anything else).

Allocate a buffer of that size, and call the function again, passing the buffer (and not the `kvImageGetTempBufferSize` flag). For details, see the function descriptions in “[vImage Reference](#)” (page 117).

The size of the buffer commonly depends on the size of the image to be processed and on the image format, among other things. Therefore, if you call a particular image-processing function repeatedly, using images of the same size every time, you will not have to reallocate a new temporary buffer each time. You can allocate a buffer once, and know that it is large enough for every call to the function.

More specifically, each function that uses a temporary buffer has a *src* and a *dest* parameter (both of type `vImage_Buffer`). The function only uses the `height` and `width` fields of these parameters; the `data` and `rowBytes` fields are ignored.

If, between two calls to a specific function, the following is true:

- the height field of the *src* parameter remains the same or decreases
- the width field of the *src* parameter remains the same or decreases
- the height field of the *dest* parameter remains the same or decreases
- the width field of the *dest* parameter remains the same or decreases

and all the other parameters to the buffer-size function besides *src* and *dest* remain the same, then the minimum size for the temporary buffer will not increase.

If other parameters change and you are concerned your existing temporary buffer is too small, you can call the function with the `kvImageGetTempBufferSize` flag as described above, check to see if the temporary buffer is large enough, and, if it is not, allocate a new one. You can still reuse the buffer if it is larger than necessary; there is no harm in passing an image-processing function a temporary buffer larger than the minimum size calculated by the corresponding buffer-size function.

If you allocate a temporary buffer, it is your responsibility to deallocate it when you are done with it. If you pass `NULL` in the *tempBuffer* parameter of an image-processing function, causing the function to allocate a temporary buffer itself, the function deallocates the buffer before returning.

Do not share a buffer between different threads of a multithreaded application unless you protect it with a lock or other mutual exclusion technique. Otherwise two vImage functions may overwrite each other's intermediate results and produce garbled results.

vImage does not keep any information in these temporary buffers between function calls.

Convolution Operations

This chapter describes the convolution operation and the vImage convolution functions. In addition, it describes some closely related functions including Richardson-Lucy deconvolution functions.

What Is Convolution?

Image convolution is a technique of smoothing an image or sharpening an image by replacing a pixel with a weighted sum of itself and nearby pixels. The size of the image remains the same.

Convolution uses a convolution kernel, which is a matrix of some size $M \times N$. M and N must both be odd. The source image is transformed as follows:

1. For each source pixel, place the kernel over the image so that the center element of the kernel lies over the source pixel.
2. For each element in the kernel, multiply that element by the value of the source pixel underneath it.
3. Add all the values calculated in Step 2.
4. In the integer formats only, divide by a given divisor.
5. The result is the value of the destination pixel.

In Mac OS X v10.4 (Tiger) you can specify a bias term for the basic convolution functions. The bias value will be added to the accumulated result during step 3. This is useful to prevent overflow or underflow in some kernels, for example, an emboss kernel, which may easily underflow. The user should note that the bias is added in before division, so for integer convolve, may need to be scaled accordingly. See “[Convolve with Bias](#)” (page 35) for details.

If the image is in a planar format, the single-channel values of the pixels are used directly in this operation. If the image is in an interleaved format, each channel (alpha, red, green, and blue) is operated on separately. In both the planar and interleaved format, the kernel itself is always planar.

Kernels for Convolution

A kernel for the convolution operation is *not* of type `vImage_Buffer`. It is a packed array, without any padding at the end of the rows. The elements of the array must be of type `int16_t` (for the Planar8 and ARGB8888 formats) or of type `float` (for the PlanarF and ARGBFFFF formats). The height and the width of the array must both be odd numbers.

For example, a 3 x 3 convolution kernel for a Planar8 image would consist of nine 16-bit (2-byte) values, arranged consecutively. The first three values would be the first row of the kernel, the next three values the second row, and the last three values the third row.

Normalization

For most purposes the convolution kernel should be normalized. For floating-point formats, this means the sum of the elements of the kernel should be 1.0f. For integer formats, the sum of the elements of the kernel, divided by the given divisor, should be 1. A non-normalized kernel will either lighten or darken the image.

Size Limits

For integer formats, the sum of any subset of elements of the kernel should be in the range $-(2^{**24})$ to $+(2^{**24}) - 1$, inclusive. Otherwise integer overflow may occur. If your kernel does not meet this restriction, either use a floating-point format or scale the kernel down to use smaller values.

Edge Conditions

When the pixel to be transformed is near the edge of the image—not merely the region of interest, but the entire image of which it is a part—the kernel may extend beyond the edge of the image, so that there are no existing pixels beneath some of the kernel’s elements. In these cases, it is necessary to have a technique for assigning values to these nonexistent pixels. `vImage` permits the caller to specify a desired technique by setting the `flag` parameter to the `Convolution` call. There are currently four possible techniques, corresponding to the following flags: `kvImageCopyInPlace`, `kvImageBackgroundColorFill`, `kvImageEdgeExtend`, and `kvImageTruncateKernel`. Exactly one of these four flags should be set in the `flag` parameter of the function call. See “[Pixels Outside the Image Buffer](#)” (page 21).

Convolution Functions

`vImage` provides the four principal convolution functions, some variations of those functions, and some special high-speed algorithms that are equivalent to certain convolutions.

Convolve

There are four principal convolve functions, one for each of the image representations supported by vImage: vImageConvolve_Planar8, vImageConvolve_PlanarF, vImageConvolve_ARGB8888, vImageConvolve_ARGBFFFF.

The caller supplies:

- A source buffer, and offsets to a point within the source image to define the upper left-hand point of the region of interest. Together with these offsets, the dimensions (number of rows and number of columns) of the destination buffer specify the size of the region of interest in the source buffer. Only the region of interest will be transformed.
- A destination buffer.
- A temporary buffer used by the function for intermediate results. If you provide `NULL`, the function will allocate the temporary buffer itself (and, of course, deallocate it when it is done), but this may decrease performance and interfere with real-time performance. See “[Temporary Buffers for vImage Functions](#)” (page 26)
- A kernel, specified by height, width, and data. The data is a packed array of either values of type `int16_t` (for the integer formats) or values of type `float` (for the floating-point formats).
- A background color pixel value to be used as a default pixel value for pixels beyond the edge of the image (only used if the `kvImageBackgroundColorFill` flag is set).
- For the integer image formats only (Planar8 and ARGB8888), a divisor for normalization.
- Flags. The `kvImageCopyInPlace`, `kvImageBackgroundColorFill`, `kvImageEdgeExtend`, `kvImageTruncateKernel`, and `kvImageDoNotTile` flags are honored. For interleaved formats, `kvImageLeaveAlphaUnchanged` is also honored.

For specific details on the parameters, see “[Convolution Functions](#)” (page 178).

Note: In Mac OS X v10.4, some of the parameter types for these functions have changed. For example, `unsigned int` has been changed to `uint_32_t` or to `size_t` (for byte counts) or `vImagePixelCount` (for pixel counts). All such changes are completely backward-compatible with code written and compiled for Mac OS X v10.3, but it is advisable to bring such code up to date and recompile.

The size (number of rows and number of columns) of the destination buffer is also used to specify the size of the region of interest in the source buffer.

For the integer image formats, the results of the convolution are divided by the given divisor for normalization. For the floating-point formats, the normalization factor is built into the kernel’s floating-point values.

To determine how pixel locations beyond the edge of the source image are handled, set exactly one of the following four flags: `kvImageCopyInPlace`, `kvImageBackgroundColorFill`, `kvImageEdgeExtend`, or `kvImageTruncateKernel`. Please see “[Pixels Outside the Image Buffer](#)” (page 21).

If you do your own tiling and/or multithreading, use the `kvImageDoNotTile` flag to prevent these functions from internally tiling or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Here are some examples of the effects that can be created with the convolution functions:

Figure 3-1 shows an unmodified image.

Figure 3-1 Facade unmodified

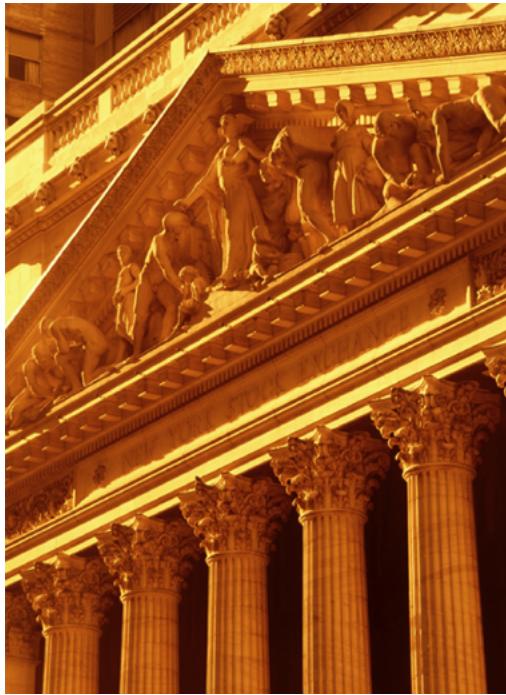
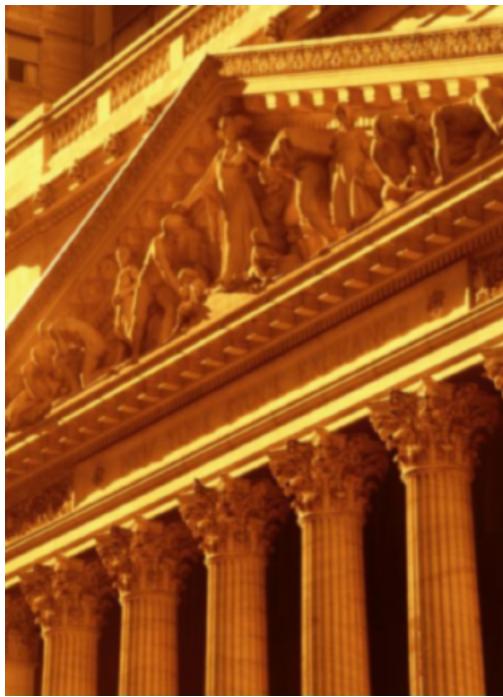


Figure 3-2 shows the same image “blurred” by a convolution with the kernel:

$$\begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 1 & 0 & 0 \\ 0 & 1 & 2 & 4 & 2 & 1 & 0 \\ 1 & 2 & 4 & 8 & 4 & 2 & 1 \\ 0 & 1 & 2 & 4 & 2 & 1 & 0 \\ 0 & 0 & 1 & 2 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

Figure 3-2 Facade blurred by convolution



In Figure 3-3, the image has been “embossed” by a convolution with the kernel:

$$\begin{pmatrix} 2 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -2 \end{pmatrix}$$

Figure 3-3 Facade embossed by convolution

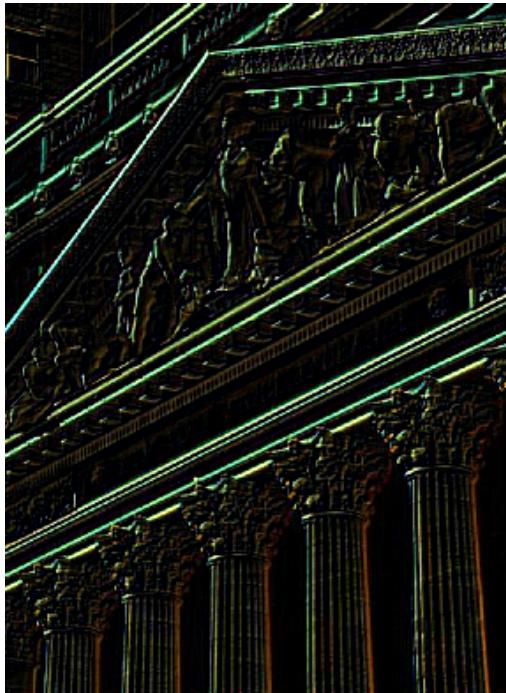


Figure 3-4 shows a different unmodified image.

Figure 3-4 Image before sharpening



In Figure 3-5, the image has been “sharpened” by a convolution with the kernel:

$$\begin{pmatrix} 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & -1 & -1 & 0 \\ -1 & -1 & 13 & -1 & -1 \\ 0 & -1 & -1 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 \end{pmatrix}$$

Figure 3-5 image sharpened by convolution

Convolve with Bias

Mac OS X v10.4 provides four variants of the principal convolve functions, which perform the same operations but also add a specified bias value to the convolution result. The bias is added before the divisor is applied or any clipping is done. These functions are `vImageConvolveWithBias_Planar8`, `vImageConvolveWithBias_PlanarF`, `vImageConvolveWithBias_ARGB8888`, and `vImageConvolveWithBias_ARGBFFFF`.

The caller supplies:

- A source buffer, and offsets to a point within the source image to define the upper left-hand point of the region of interest. Together with these offsets, the dimensions (number of rows and number of columns) of the destination buffer specify the size of the region of interest in the source buffer. Only the region of interest will be transformed.
- A destination buffer.
- A temporary buffer used by the function for intermediate results. If you provide `NULL`, the function will allocate the temporary buffer itself (and, of course, deallocate it when it is done), but this may decrease performance and interfere with real-time performance. See “[Temporary Buffers for vImage Functions](#)” (page 26)

- A kernel, specified by height, width, and data. The data is a packed array of either values of type `int16_t` (for the integer formats) or values of type `float` (for the floating-point formats).
- A bias value, of type `int_32` for integer formats or `float` for floating-point formats.
- A background color pixel value to be used as a default pixel value for pixels beyond the edge of the image (only used if the `kvImageBackgroundColorFill` flag is set).
- For the integer image formats only (Planar8 and ARGB8888), a divisor for normalization.
- Flags. The `kvImageCopyInPlace`, `kvImageBackgroundColorFill`, `kvImageEdgeExtend`, `kvImageTruncateKernel`, and `kvImageDoNotTile` flags are honored. For interleaved formats, `kvImageLeaveAlphaUnchanged` is also honored.

For specific details on the parameters, see “[Convolution Functions](#)” (page 178).

The size (number of rows and number of columns) of the destination buffer is also used to specify the size of the region of interest in the source buffer.

For the integer image formats, the results of the convolution are divided by the given divisor for normalization. For the floating-point formats, the normalization factor is built into the kernel’s floating-point values.

To determine how pixel locations beyond the edge of the source image are handled, set exactly one of the following four flags: `kvImageCopyInPlace`, `kvImageBackgroundColorFill`, `kvImageEdgeExtend`, or `kvImageTruncateKernel`. Please see “[Pixels Outside the Image Buffer](#)” (page 21).

If you do your own tiling and/or multithreading, use the `kvImageDoNotTile` flag to prevent these functions from internally tiling or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23)

Convolve with Multiple Kernels

Mac OS X v10.4 provides two variants of the principal convolve functions for the interleaved formats, which perform the same operations but use a different kernel for each channel. These functions are `vImageConvolveMultiKernel_ARGB8888` and `vImageConvolveMultiKernel_ARGBFFFF`.

The caller supplies:

- A source buffer, and offsets to a point within the source image to define the upper left-hand point of the region of interest. Together with these offsets, the dimensions (number of rows and number of columns) of the destination buffer specify the size of the region of interest in the source buffer. Only the region of interest will be transformed.
- A destination buffer.
- A temporary buffer used by the function for intermediate results. If you provide `NULL`, the function will allocate the temporary buffer itself (and, of course, deallocate it when it is done), but this may decrease performance and interfere with real-time performance. See “[Temporary Buffers for vImage Functions](#)” (page 26)
- Four kernels, each specified by height, width, and data. The data for each kernel is a packed array of either values of type `int16_t` (for the integer formats) or values of type `float` (for the floating-point formats).
- A pair of `uint32_t` values specifying the height and width of the kernels.

- For the ARGB8888 format only, a packed array of four divisors for normalization.
- A packed array of four bias values, of type `int_32` for the ARGB8888 format or `float` for the ARGBFFFF format.
- A background color pixel value to be used as a default pixel value for pixels beyond the edge of the image (only used if the `kvImageBackgroundColorFill` flag is set).
- Flags. The `kvImageCopyInPlace`, `kvImageBackgroundColorFill`, `kvImageEdgeExtend`, `kvImageTruncateKernel`, `kvImageDoNotTile`, and `kvImageLeaveAlphaUnchanged` flags are honored.

For specific details on the parameters, see “[Convolution Functions](#)” (page 178).

The size (number of rows and number of columns) of the destination buffer is also used to specify the size of the region of interest in the source buffer.

For the integer image formats, the results of the convolution are divided by the given divisor for normalization. For the floating-point formats, the normalization factor is built into the kernel’s floating-point values.

To determine how pixel locations beyond the edge of the source image are handled, set exactly one of the following four flags: `kvImageCopyInPlace`, `kvImageBackgroundColorFill`, `kvImageEdgeExtend`, or `kvImageTruncateKernel`. Please see “[Pixels Outside the Image Buffer](#)” (page 21).

If you do your own tiling and/or multithreading, use the `kvImageDoNotTile` flag to prevent these functions from internally tiling or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23)

High-Speed Box and Tent Filters

For the integer formats, Planar_8 and ARGB8888, Mac OS X v10.4 provides high-speed “box” and “tent” filter functions. They are equivalent to convolving with certain simple kernels, but no actual kernel is supplied. These functions can be orders of magnitude faster than the equivalent convolutions.

For the box functions `vImageBoxConvolve_Planar8` and `vImageBoxConvolveARGB8888`, each result pixel is the mean of surrounding pixels as defined by the kernel height and width. The operation is equivalent to applying a convolution kernel filled with all 1’s.

For the tent functions `vImageTentConvolve_Planar8` and `vImageTentConvolveARGB8888`, the operation is equivalent to applying a convolution kernel filled with values that decrease with distance from the center. Specifically, for an $M \times N$ kernel size, the kernel would be the product of an $M \times 1$ column matrix and a $1 \times N$ row matrix. Each would have 1 as the first element, then values increasing by 1 up to the middle element, then decreasing by 1 to the last element.

For example, suppose the kernel size is 3×5 . Then the first matrix is

$$\begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}$$

and the second is

$$(1 \ 2 \ 3 \ 2 \ 1)$$

The product is

$$\begin{pmatrix} 1 & 2 & 3 & 2 & 1 \\ 2 & 4 & 6 & 4 & 2 \\ 1 & 2 & 3 & 2 & 1 \end{pmatrix}$$

The 3×5 tent filter operation is equivalent to convolution with the above matrix.

The caller supplies

- A source buffer, and offsets to a point within the source image to define the upper left-hand point of the region of interest. Together with these offsets, the dimensions (number of rows and number of columns) of the destination buffer specify the size of the region of interest in the source buffer. Only the region of interest will be transformed.
- A destination buffer. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.
- A temporary buffer used by the function for intermediate results. If you provide `NULL`, the function will allocate the temporary buffer itself (and, of course, deallocate it when it is done), but this may decrease performance and interfere with real-time performance. See “[Temporary Buffers for vImage Functions](#)” (page 26)
- A pair of `uint32_t` values specifying the height and width of the “kernel.”
- A background color pixel value to be used as a default pixel value for pixels beyond the edge of the image (only used if the `kvImageBackgroundColorFill` flag is set).
- Flags. The `kvImageCopyInPlace`, `kvImageBackgroundColorFill`, `kvImageEdgeExtend`, `kvImageTruncateKernel`, and `kvImageDoNotTile` flags are honored. For ARGB8888 format, `kvImageLeaveAlphaUnchanged` is also honored.

For specific details on the parameters, see “[Convolution Functions](#)” (page 178).

The size (number of rows and number of columns) of the destination buffer is also used to specify the size of the region of interest in the source buffer.

The results of the convolution are divided by the given divisor for normalization. For the floating-point formats, the normalization factor is built into the kernel’s floating-point values.

To determine how pixel locations beyond the edge of the source image are handled, set exactly one of the following four flags: `kvImageCopyInPlace`, `kvImageBackgroundColorFill`, `kvImageEdgeExtend`, or `kvImageTruncateKernel`. Please see “[Pixels Outside the Image Buffer](#)” (page 21).

vImageGetMinimumTempBufferSizeForConvolution

This function is deprecated in Mac OS X v10.4. Instead, pass the `kvImageGetTempBufferSize` flag to any `vImage` function that takes a temporary buffer as a parameter; the function will return the minimum size, in bytes, of the temporary buffer (and will not do anything else). Please see “[Temporary Buffers for vImage Functions](#)” (page 26). Note that if the result is negative, it is an error code.

This utility function is used to obtain the minimum size, in bytes, of the temporary buffer used by the convolution functions.

The caller supplies:

- the source buffer that will be used in the Convolution call
- the destination buffer that will be used in the Convolution call
- the kernel height and width that will be used in the Convolution call
- the flags that will be used in the Convolution call
- The size of a pixel, in bytes (this depends on the image format you are using)

For specific details on the parameters, see “[Convolution Functions](#)” (page 178).

This function does not depend on the `data` or `rowBytes` fields of the source or destination buffer; it only uses the `height` and `width` fields from those parameters. If the size of the images you are processing stay the same, then the required size of the buffer will also stay the same. The function is designed to make it easy to reuse a buffer if the size of your images do not increase from one call to the next. See “[Temporary Buffers for vImage Functions](#)” (page 26).

Deconvolution Functions

Deconvolution is an operation that (approximately) undoes a previous convolution — typically a convolution that is physical in origin, such as diffraction effects in a lens. Usually, deconvolution is a “sharpening” operation.

There is a large family of deconvolution algorithms; `vImage` provides a set of functions to perform a particular deconvolution called the Richardson-Lucy deconvolution.

This is an iterative operation; the caller specifies the number of iterations desired. The more iterations are used, the greater the sharpening effect (and of course the time consumed). Note that like any sharpening operation, Richardson-Lucy amplifies noise, and at some number of iterations the noise will show up as noticeable artefacts.

The algorithm requires a “point spread function,” which specifies the convolution that is being undone. In this implementation, the point spread function is the kernel of that convolution; if the kernel is not symmetrical, a second kernel should be supplied which is the same as the first with the axes interchanged.

$$e_{i+1} = e_i \cdot \left(psf_0 * \left(\frac{O}{psf_1 * e_i} \right) \right)$$

If $psf_1 = \text{Null}$, psf_0 is used

where

- O is the starting image.
- e[i] is the current result and e[i+1] is being calculated; e[0] is O.
- psf[0] is the kernel representing the point spread function.
- psf[1] is a second kernel to be used if psf[0] is asymmetrical. A `NULL` value is used if psf[0] is symmetrical.
- The `*` operator indicates convolution.
- The division indicated in the innermost parentheses is division of each element in the upper matrix by the corresponding element in the lower matrix.
- The `.` operator indicates multiplication of each element in one matrix by the corresponding element in the other matrix.

Richardson-Lucy Deconvolution Functions

There are four Richardson-Lucy functions, one for each of vImage's major image formats: `vImageRichardsonLucyDeConvolve_Planar8`, `vImageRichardsonLucyDeConvolve_PlanarF`, `vImageRichardsonLucyDeConvolve_ARGB8888`, and `vImageRichardsonLucyDeConvolve_ARGBFFF`.

The caller supplies:

- A source buffer, and offsets to a point within the source image to define the upper left-hand point of the region of interest. Together with these offsets, the dimensions (number of rows and number of columns) of the destination buffer specify the size of the region of interest in the source buffer. Only the region of interest will be transformed.
- A destination buffer.
- A temporary buffer used by the function for intermediate results. If you provide `NULL`, the function will allocate the temporary buffer itself (and, of course, deallocate it when it is done), but this may decrease performance and interfere with real-time performance. See “[Temporary Buffers for vImage Functions](#)” (page 26)
- A kernel, specified by height, width, and data. The data is a packed array of either values of type `int16_t` (for the integer formats) or values of type `float` (for the floating-point formats). This kernel is the `psf[0]` discussed above.
- A second kernel, specified in the same way. This is the `psf[1]` discussed above. If this second kernel is not needed, pass `NULL`.
- For 8-bit image formats only, a divisor to be used in the convolution that involves the first kernel.

- For 8-bit image formats only, a divisor to be used in the convolution that involves the second kernel.
- A background color pixel value to be used as a default pixel value for pixels beyond the edge of the image (only used if the `kvImageBackgroundColorFill` flag is set).
- An iteration count.
- Flags. The `kvImageCopyInPlace`, `kvImageBackgroundColorFill`, `kvImageEdgeExtend`, `kvImageTruncateKernel`, and `kvImageDoNotTile` flags are honored. For interleaved formats, `kvImageLeaveAlphaUnchanged` is also honored.

For specific details on the parameters, see “[Convolution Functions](#)” (page 178).

To determine how pixel locations beyond the edge of the source image are handled, set exactly one of the following four flags: `kvImageCopyInPlace`, `kvImageBackgroundColorFill`, `kvImageEdgeExtend`, or `kvImageTruncateKernel`. Please see “[Pixels Outside the Image Buffer](#)” (page 21).

If you do your own tiling and/or multithreading, use the `kvImageDoNotTile` flag to prevent these functions from internally tiling or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23)

C H A P T E R 3
Convolution Operations

Morphological Operations

This chapter describes morphological operations and the morphology functions provided by vImage.

What Are Morphological Operations?

Morphological operations tend to make an object in an image bigger—expanding away from its original borders—or smaller—contracting within its original borders. The precise nature of the expanding or shrinking is determined by a kernel (also known as a *structure element*) provided by the caller. The size (number of rows and number of columns) of the entire image does not change.

The morphological operations provided by vImage can first be defined on grayscale images, where the source image is planar (single-channel). The definition can then be expanded to full-color images. (The kernel itself is always planar.)

vImage performs four morphological operations: dilate, erode, max, and min. Dilate expands objects; erode contracts them. Max and min are special cases of dilate and erode, respectively.

An “object” is not the same as the entire image. There is no precise definition for object. Relatively speaking, bright (high-intensity) pixels are considered part of the object, while darker pixels are considered part of the background.

If, instead, you consider darker pixels as part of the object, and lighter pixels as background, then dilate and erode will have the opposite effect from usual: dilate will shrink the object, erode will expand it.

Morphological Operations on Grayscale Images

In the case of greyscale morphological operations, the source image is planar (single-channel).

Definition of Dilation

Dilation of a source image (region of interest) by a kernel is defined in the following way:

1. For each source pixel, place the kernel over the image so that the center element of the kernel lies over the source pixel.

2. For each pixel in the kernel, subtract the value of that pixel from the value of the source pixel underneath it. Negative intermediate values are permitted.
3. Take the maximum of all the values calculated in Step 2.
4. Add the value of the center pixel of the kernel. The result is the value of the destination pixel.

The general effect of dilation is to take each bright pixel in source image and expand it into the shape of the kernel, flipped horizontally and vertically. The contribution of the source pixel to the kernel-shaped region depends on two things: the brightness of the source pixel (brighter pixels contribute more) and the values of the kernel pixels (pixels that are dark, relative to the center of the kernel, contribute more to their locations in the kernel-shaped region than pixels that are bright).

Definition of Erosion

Erosion of a source image (region of interest) by a kernel is defined in the a very similar manner:

1. For each source pixel, place the kernel over the image so that the center element of the kernel lies over the source pixel.
2. For each pixel in the kernel, subtract the value of that pixel from the value of the source pixel underneath it. Negative intermediate values are permitted.
3. Take the minimum of all the values calculated in Step 2.
4. Add the value of the center pixel of the kernel. The result is the value of the destination pixel.

Instead of expanding objects, erosion tends to spread dark pixels around, causing them to eat away at (erode) objects.

Other Morphological Operations

The morphological operation max is a special case of the dilate operation in which all the values of the kernel are the same. (It does not matter what specific value is used, so for convenience it is assumed to be 0.) Similarly, the min operation is a special case of the erode operation in which all the values of the kernel are the same.

Two other well-known morphological operations, open and close, are not performed directly by vImage. The open can be accomplished by an erode operation followed by a dilate operation. The close operation is a dilate operation followed by an erode operation. If all the values of the kernel are the same, then the open operation can be accomplished by a min operation followed by a max operation, and the close operation can be accomplished by a max operation followed by a min operation.

Morphological Operations for Full-color Images

Morphological operations on full-color images are defined in the following way:

Separate the image into four planar images each (one planar image for each of Alpha, Red, Green, and Blue)

Apply the desired morphological operation to each color plane separately, treating the planar images as greyscale

Recombine the results into a full-color image.

(If the `kvImageLeaveAlphaUnchanged` flag is set, the morphological operation is not performed on the alpha channel.)

Kernels for Morphological Operations

A kernel for a morphological operation is *not* of type `vImage_Buffer`. It is a packed array, without any padding at the end of the rows. The elements of the array must be of type `uint8_t` (for the Planar8 and ARGB8888 formats) or of type `float` (for the PlanarF and ARGBFFFF formats). The height and the width of the array must both be odd numbers.

For example, a 3×3 convolution kernel for a Planar8 image would consist of nine 8-bit (1-byte) values, arranged consecutively. The first three values would be the first row of the kernel, the next three values the second row, and the last three values the third row.

Edge Conditions for Morphology Functions

The morphology functions do saturated clipping to prevent overflow for the Planar8 and ARGB8888 formats. Saturated clipping maps all intensity levels above 255 to 255, all intensity levels below 0 to 0, and leaves intensity levels between 0 and 255, inclusive, unchanged.

When the pixel to be transformed is near the edge of the image—not merely the region of interest, but the entire image of which it is a part—the kernel may extend beyond the edge of the image, so that there are no existing pixels beneath some of the kernel’s elements. In this case the morphology functions only make use of that part of the kernel which overlaps the source buffer. The other kernel elements are ignored.

Morphological Functions

`vImage` provides the following morphological functions.

Dilate

There are four dilate functions, one for each of the image representations supported by `vImage`: `vImageDilate_Planar8`, `vImageDilate_PlanarF`, `vImageDilate_ARGB8888`, `vImageDilate_ARGBFFFF`.

The caller supplies:

- A source buffer, and offsets to a point within the source image to define the upper left-hand point of the region of interest (only the region of interest will be transformed).

- A destination buffer. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.
- A kernel, specified by height, width, and data. The data is a packed array of either values of type `uint8_t` (for the integer formats) or values of type `float` (for the floating-point formats).
- Flags.

For specific details on the parameters, see “[Morphological Functions](#)” (page 254).

The size (number of rows and number of columns) of the destination buffer is also used to specify the size of the region of interest in the source buffer.

Set the `kvImageDoNotTile` flag to prevent `vImage` from using tiling internally. (This is appropriate if you are doing tiling yourself.) Set the `kvImageLeaveAlphaUnchanged` flag to specify that the alpha channel should be copied to the destination image unchanged.

Erode

There are four erode functions, one for each of the image representations supported by `vImage`: `vImageErode_Planar8`, `vImageErode_PlanarF`, `vImageErode_ARGB8888`, `vImageErode_ARGBFFFF`.

The caller supplies:

- A source buffer, and offsets to a point within the source image to define the upper left-hand point of the region of interest (only the region of interest will be transformed).
- A destination buffer. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.
- A kernel, specified by height, width, and data. The data is a packed array of either values of type `uint8_t` (for the integer formats) or values of type `float` (for the floating-point formats).
- Flags.

For specific details on the parameters, see “[Morphological Functions](#)” (page 254).

The size (number of rows and number of columns) of the destination buffer is also used to specify the size of the region of interest in the source buffer.

Set the `kvImageDoNotTile` flag to prevent `vImage` from using tiling internally. (This is appropriate if you are doing tiling yourself.) (This is appropriate if you are doing tiling yourself.) Set the `kvImageLeaveAlphaUnchanged` flag to specify that the alpha channel should be copied to the destination image unchanged.

Max

There are max functions, one for each of the image representations supported by `vImage`: `vImageMax_Planar8`, `vImageMax_PlanarF`, `vImageMax_ARGB8888`, `vImageMax_ARGBFFFF`.

The caller supplies:

- a source buffer, and offsets to a point within the source image to define the upper left-hand point of the region of interest (only the region of interest will be transformed).
- a destination buffer. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.
- the height and width of the kernel (the actual kernel value is not necessary).
- a temporary buffer used by the function for intermediate results. If you provide `NULL`, the function will allocate the temporary buffer itself (and, of course, deallocate it when it is done), but this may decrease performance and interfere with real-time performance.
- flags.

For specific details on the parameters, see “[Morphological Functions](#)” (page 254).

The size (number of rows and number of columns) of the destination buffer is also used to specify the size of the region of interest in the source buffer.

Set the `kvImageDoNotTile` flag to prevent `vImage` from using tiling internally. (This is appropriate if you are doing tiling yourself.) Set the `kvImageLeaveAlphaUnchanged` flag to specify that the alpha channel should be copied to the destination image unchanged.

Min

There are four `min` function, one for each of the image representations supported by `vImage`: `vImageMin_Planar8`, `vImageMin_PlanarF`, `vImageMin_ARGB8888`, `vImageMin_ARGBFFFF`.

The caller supplies:

- A source buffer, and offsets to a point within the source image to define the upper left-hand point of the region of interest (only the region of interest will be transformed).
- A destination buffer. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.
- The height and width of the kernel (the actual kernel value is not necessary).
- A temporary buffer used by the function for intermediate results. If you provide `NULL`, the function will allocate the temporary buffer itself (and, of course, deallocate it when it is done), but this may decrease performance and interfere with real-time performance.
- Flags.

For specific details on the parameters, see “[Morphological Functions](#)” (page 254).

The size (number of rows and number of columns) of the destination buffer is also used to specify the size of the region of interest in the source buffer.

Set the `kvImageDoNotTile` flag to prevent `vImage` from using tiling internally. (This is appropriate if you are doing tiling yourself.) Set the `kvImageLeaveAlphaUnchanged` flag parameter to specify that the alpha channel should be copied to the destination image unchanged.

vImageGetMinimumTempBufferSizeForMinMax

This function is deprecated in Mac OS X v10.4. Instead, pass the `kvImageGetTempBufferSize` flag to any `vImage` function that takes a temporary buffer as a parameter; the function will return the minimum size, in bytes, of the temporary buffer (and will not do anything else). Please see “[Temporary Buffers for vImage Functions](#)” (page 26). Note that if the result is negative, it is an error code.

This utility function is used to obtain the necessary size for the temporary buffer used by the min and max functions.

The caller supplies:

- the source buffer that will be used in the min or max call.
- the destination buffer that will be used in the min or max call.
- the kernel height and width
- the flags that will be used with the min or max call
- The size of a pixel, in bytes (this depends on the image format you are using)

For specific details on the parameters, see “[Morphological Functions](#)” (page 254).

This function does not depend on the `data` or `rowBytes` fields of the source or destination buffer; it only uses the `height` and `width` fields from those parameters. If the size of the images you are processing stay the same, then the required size of the buffer will also stay the same. The function is designed to make it easy to reuse a buffer if your size of your images do not increase from one call to the next. See “[Temporary Buffers for vImage Functions](#)” (page 26).

Geometric Operations

This chapter describes the geometric operations provided by vImage.

What Are Geometric Operations?

Geometric operations rotate, resize, and distort the geometry of images. vImage provides both high-level and low-level geometric functions. The high-level functions are Rotate, Scale, and Affine Warp. The low-level functions are Reflect, Shear, and Rotate90.

Resampling

Most of the vImage geometric functions need to use resampling in order to avoid creating artifacts, such as interference patterns, in the destination image. Resampling is done using kernels, which combine data from a target pixel and other nearby pixels to calculate a value for the destination pixel. This is very similar to what is done in the convolution operation.

However, in the geometric operations, the resampling kernel must itself be resampled during the process of pairing kernel values against the sampled pixel data. The kernel must be evaluated at fractional pixel locations, in addition to integral pixel locations. Consequently, instead of using an M x N matrix for the kernel (as the convolution operations do), the operations use a kernel *function* that can be evaluated at both fractional and integral pixel locations. This resampling kernel function is called a **resampling filter**, or simply a filter.

For almost all geometric operations, vImage supplies a default resampling filter. There are actually two default filters to choose from. If you do not set the `kvImageHighQualityResampling` flag for the call, vImage will use a Lanczos3 filter. If you do set the flag, vImage will use a Lanczos5 filter. A Lanczos5 filter does higher-quality resampling than a Lanczos3 filter, but is slower to use.

Important: Throughout this document the default filters supplied by vImage are referred to as the Lanczos filters. This is currently the case. However, future implementations may use different default filters for improved performance and quality.

The rotate90 and reflect functions do not use resampling.

The shear functions are a special case. They can use a default filter, but they also permit you to specify a custom filter of your own. You can use this feature when you need increased control over the operation. Most callers will not need to use it.

Shear Functions and User-defined Resampling Filters

In some ways, the shear functions are the most basic of the geometric operations supplied by vImage. It is possible to duplicate most of the other geometric operations by using a small number of shears in sequence. (This document does not describe how to do this.) For this reason, vImage allows the user of the shear functions to provide a user-defined resampling filter for use with the shear functions. This permits a higher level of control than is possible with the other geometric functions.

This is done by passing the shear function a parameter of type `ResamplingFilter`. The `ResamplingFilter` parameter represents, or encapsulates, a resampling filter. The internal structure of the `ResamplingFilter` object is deliberately left undocumented. It may contain a pointer to your custom filter function, rows of calculated kernel values, flags, or anything else it needs to evaluate your filter. The structure may change in future releases. This should not affect the way your software uses it.

Because the shear functions operate on a line-by-line basis, a resampling kernel used with a shear function is one-dimensional, a $1 \times N$ matrix. Consequently the resampling filter for a shear function can be modeled as a one-dimensional function $y = f(x)$. When using a custom filter, you will need to provide a routine that evaluates your function $f(x)$ on an array of x values. At the same time you can, if desired, normalize the filter over the given array of x values. (This is the main reason your routine is required to evaluate your filter on an array of x values, rather than a single x value. It also improves performance.) To normalize the filter over an array of x values, you scale the filter values so that they add up to 1.0. If you do not do this, interference patterns may emerge in the resulting image.

Creating A `ResamplingFilter` Object

When you call a shear function, you must pass a parameter of type `ResamplingFilter`, regardless of whether you want to use a custom resampling filter or not.

If you want to use a default resampling filter, you create it by calling `vImageNewResamplingFilter`. This creates a new `ResamplingFilter` object that represents a default filter—a Lanczos3 filter if you do not set the `kvImageHighQualityResampling` flag, or a Lanczos5 filter if you do set that flag. You do not need to provide a filter function when you make this call.

The `ResamplingFilter` object can be used in multiple shear function calls. However, the filter includes a scaling factor. If you want to use a different scaling factor, you must create a new filter.

When you are done using a `ResamplingFilter` object created by a call to `vImageNewResamplingFilter`, dispose of it by calling `vImageDestroyResamplingFilter`. Do not attempt to directly deallocate the object's memory yourself.

If you want to create a `ResamplingFilter` object that represents your custom resampling filter, you create it by calling `vImageNewResamplingFilterForFunctionUsingBuffer`. You must pass a pointer to your custom filter function. `vImageNewResamplingFilterForFunctionUsingBuffer` creates the `ResamplingFilter` object in a buffer that you allocate directly. When you are done using the object, you must deallocate its memory yourself. Do not pass a `ResamplingFilter` object created by `vImageNewResamplingFilterForFunctionUsingBuffer` to `vImageDestroyResamplingFilter`.

To get the size of the buffer required for a custom ResamplingFilter object, call vImageGetResamplingFilterSize.

Both the default ResamplingFilter and the custom ResamplingFilter include a scale factor that you provide. This scale factor will be used to scale the transformed image in the shear operation.

A Sample User-Defined Resampling Filter

In this custom filter routine, the underlying filter function is $y = 1.0 - |x|$. The custom filter must evaluate that function on an array of x values. The filter may not be normalized over that range of x values. You can normalize it within the routine by dividing by the sum of the result values, as shown. This routine does not use userData.

```
void MyLinearInterpolationFilterFunc( const float *xArray, float *yArray, int
    count, void *userData )
{
    int i;
    float sum = 0.0f;
    //Calculate kernel values
    for( i = 0; i < count; i++ )
    {
        float unscaledResult = 1.0f - fabs( xArray[i] );      //LERP
        yArray[i] = unscaledResult;
        sum += unscaledResult;
    }
    //Make sure the kernel values sum to 1.0. You can use some other value here.
    //Values other than 1.0 will cause the image to lighten or darken.
    sum = 1.0f / sum;
    for( i = 0; i < count; i++ )
        yArray[i] *= sum;
}
```

Affine Transformations

The vImage affine warp functions perform affine transformations on an image. An affine transformation is a very general geometric transformation in which each pixel in the source image, given by its coordinates $[x, y]$, is mapped to a new location, $[x', y']$, in the destination image. The transformation is determined by an affine transform matrix

$$\begin{pmatrix} a & b \\ c & d \\ tx & ty \end{pmatrix}$$

according to the formula

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \times \begin{pmatrix} a & b \\ c & d \\ tx & ty \end{pmatrix}$$

(Note that tx and ty are each symbols for individual values. They do not represent multiplication.) The additional component “1” on the right-hand side is not part of the location of the source pixel; it is added simply to allow us to represent the affine transformation as a matrix multiplication. In fact, it is customary to add an extra coordinate to the result vector and represent the affine transformation as

$$\begin{bmatrix} x', y', 1 \end{bmatrix} = \begin{bmatrix} x, y, 1 \end{bmatrix} \times \begin{pmatrix} a & b & 0 \\ c & d & 0 \\ tx & ty & 1 \end{pmatrix}$$

This makes it easier to calculate the inverse transformation, since the 3×3 matrix can be inverted.

The affine transformation can be decomposed into two less general transformations: a linear transformation followed by a translation (in both the x and y directions). The linear transformation is defined as

$$\begin{bmatrix} x', y' \end{bmatrix} = \begin{bmatrix} x, y \end{bmatrix} \times \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

The translation simply adds tx to the x -coordinate and ty to the y -coordinate.

The `vImage_AffineTransform` structure, which contains an affine transformation matrix, is defined to be the same as the `CGAffineTransform`. Some functions for creating and manipulating matrixes of this form can be found in `CoreGraphics/CGAffineTransform.h`.

Edge Conditions

The `vImage` geometric functions may sometimes need pixel values for pixel locations that are outside of the source image. For example, a rotate function may “rotate in” an area that was outside the original image. Or, a resampling filter may extend beyond the edges of the image.

Most of the geometric functions allow you to choose between the background color fill technique and the edge extend technique to handle this problem. However, the `rotate90` and `reflect` functions currently only permit the background color fill technique. See “[Pixels Outside the Image Buffer](#)” (page 21).

Geometric Functions

High-Level Geometric Functions

Rotate

There are four `rotate` functions, one for each of the image representations supported by `vImage`: `vImageRotate_Planar8`, `vImageRotate_PlanarF`, `vImageRotate_ARGB8888`, `vImageRotate_ARGBFFFF`.

This function rotates a source image by a given angle and places the result in the destination buffer. The center point of the source image is mapped to the center point of the destination image. No scaling is done. Depending on the relative sizes of the source image and destination image, parts of the source image may be clipped, and areas outside the source image may appear in the destination image.

Resampling is done using a Lanczos filter.

The caller supplies:

- A source buffer.
- A destination buffer. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.
- A temporary buffer used by the function for intermediate results. If you provide `NULL`, the function will allocate the temporary buffer itself (and, of course, deallocate it when it is done), but this may decrease performance and interfere with real-time performance.
- The angle of rotation, in radians.
- A background color (only used if the `kvImageBackgroundColorFill` flag is set).
- Flags.

For specific details on the parameters, see “[Geometric Functions](#)” (page 275).

Set the `kvImageDoNotTile` field in the flags parameter to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.) To specify how to handle pixels beyond the edge of the source image are handled, set exactly one of the following two flags: `kvImageBackgroundColorFill`, or `kvImageEdgeExtend`. See “[Pixels Outside the Image Buffer](#)” (page 21). Set the `kvImageHighQualityResampling` field of the flags parameter to specify that you want the function to use a Lanczos5 kernel for resampling; otherwise a Lanczos3 kernel will be used. This function ignores the `kvImageLeaveAlphaUnchanged` flag field.

[Figure 5-1](#) (page 53) shows an unmodified picture of a bluejay. Figure 5-2 shows the same picture after a rotation.

Figure 5-1 Bluejay, unmodified



Figure 5-2 Bluejay, rotated

Scale

There are four scale functions, one for each of the image representations supported by vImage: `vImageScale_Planar8`, `vImageScale_PlanarF`, `vImageScale_ARGB8888`, `vImageScale_ARGBFFFF`.

This function scales a source image to fit a destination buffer.

Resampling is done using a Lanczos filter. If pixels outside the source image need to be examined, they are given values using the edge extend technique. See “[Pixels Outside the Image Buffer](#)” (page 21).

The caller supplies:

- A source buffer.
- A destination buffer. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.
- A temporary buffer used by the function for intermediate results. If you provide `NULL`, the function will allocate the temporary buffer itself (and, of course, deallocate it when it is done), but this may decrease performance and interfere with real-time performance.
- Flags.

For specific details on the parameters, see “[Geometric Functions](#)” (page 275).

Set the `kvImageDoNotTile` field in the flags parameter to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.) To specify how to handle pixels beyond the edge of the source image are handled, set exactly one of the following two flags: `kvImageBackgroundColorFill`, or `kvImageEdgeExtend`. Set the `kvImageHighQualityResampling` field of the flags parameter to specify that you want the function to use a Lanczos5 kernel for resampling; otherwise a Lanczos3 kernel will be used. This function ignores the `kvImageLeaveAlphaUnchanged` flag field.

Affine Warp

There are four affine warp functions, one for each of the image representations supported by vImage: `vImageAffineWarp_Planar8`, `vImageAffineWarp_PlanarF`, `vImageAffineWarp_ARGB8888`, `vImageAffineWarp_ARGBFFFF`.

This function performs an affine transform on a source image and places the result in the destination buffer. Depending on the particular affine transformation, and the relative sizes of the source image and destination image, parts of the source image may be clipped, and areas outside the source image may appear in the destination image.

Resampling is done using a Lanczos filter.

The caller supplies:

- A source buffer.
- A destination buffer.
- A temporary buffer used by the function for intermediate results. If you provide `NULL`, the function will allocate the temporary buffer itself (and, of course, deallocate it when it is done), but this may decrease performance and interfere with real-time performance.
- The affine transform matrix.
- A background color.
- Flags.

For specific details on the parameters, see “[Geometric Functions](#)” (page 275).

See “[Affine Transformations](#)” (page 51) for more information.

Set the `kvImageDoNotTile` field in the flags parameter to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.) To specify how to handle pixels beyond the edge of the source image are handled, set exactly one of the following two flags:

`kvImageBackgroundColorFill`, or `kvImageEdgeExtend`. See “[Pixels Outside the Image Buffer](#)” (page 21).

Set the `kvImageHighQualityResampling` field of the flags parameter to specify that you want the function to use a Lanczos5 kernel for resampling; otherwise a Lanczos3 kernel will be used. This function ignores the `kvImageLeaveAlphaUnchanged` flag field.

Figure 5-3 shows the bluejay of [Figure 5-1](#) (page 53) after transformation by an affine warp.

Figure 5-3 Bluejay, after affine warp.



vImageGetMinimumGeometryTempBufferSize

This function is deprecated in Mac OS X v10.4. Instead, pass the `kvImageGetTempBufferSize` flag to any `vImage` function that takes a temporary buffer as a parameter; the function will return the minimum size, in bytes, of the temporary buffer (and will not do anything else). Please see “[Temporary Buffers for vImage Functions](#)” (page 26). Note that if the result is negative, it is an error code.

This utility function is used to obtain the minimum size, in bytes, of the temporary buffer used by the rotate, scale, and affine warp functions.

The caller supplies:

- the source buffer that will be used in the function call
- the destination buffer that will be used in the function call
- the flags that will be used in the function call
- The size of a pixel, in bytes (this depends on the image format you are using)

For specific details on the parameters, see “[Geometric Functions](#)” (page 275).

This function does not depend on the `data` or `rowBytes` fields of the source or destination buffer; it only uses the `height` and `width` fields from those parameters. If the size of the images you are processing stay the same, then the required size of the buffer will also stay the same. The function is designed to make it easy to reuse a buffer if your size of your images do not increase from one call to the next. See “[Temporary Buffers for vImage Functions](#)” (page 26).

Low-level Geometric Functions

Horizontal Reflect

There are four horizontal reflect functions, one for each of the image representations supported by vImage: vImageHorizontalReflect_Planar8, vImageHorizontalReflect_PlanarF, vImageHorizontalReflect_ARGB8888, vImageHorizontalReflect_ARGBFFFF.

This function reflects the source image left to right across the center vertical line of the image, and puts the result in a destination buffer. The source and destination buffers must have the same height and the same width (in pixels). No scaling or resampling is done.

The caller supplies:

- A source buffer.
- A destination buffer. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.
- Flags.

For specific details on the parameters, see “[Geometric Functions](#)” (page 275).

Set the `kvImageDoNotTile` field in the flags parameter to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.)

Figure 5-4 shows the bluejay of [Figure 5-1](#) (page 53) after a horizontal reflection.

Figure 5-4 Bluejay, after a horizontal reflection



Vertical Reflect

There are four vertical reflect functions, one for each of the image representations supported by vImage: vImageVerticalReflect_Planar8, vImageVerticalReflect_PlanarF, vImageVerticalReflect_ARGB8888, vImageVerticalReflect_ARGBFFFF.

This function reflects the source image top to bottom across the center horizontal line of the image, and puts the result in a destination buffer. This causes the image to appear upside down as if seen from behind. The source and destination buffers must have the same height and the same width (in pixels). No scaling or resampling is done.

The caller supplies:

- A source buffer.
- A destination buffer. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.
- Flags.

For specific details on the parameters, see “[Geometric Functions](#)” (page 275).

Set the `kvImageDoNotTile` field in the flags parameter to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.)

Figure 5-5 shows the bluejay of [Figure 5-1](#) (page 53) after a vertical reflection.

Figure 5-5 Bluejay, after vertical reflection



Rotate90

There are four `rotate90` functions, one for each of the image representations supported by vImage: `vImageRotate90_Planar8`, `vImageRotate90_PlanarF`, `vImageRotate90_ARGB8888`, `vImageRotate90_ARGBFFFF`.

This function rotates a source image by either 0, 90, 180, or 270 degrees (depending on a value supplied by the caller) and places the result in the destination buffer. The center point of the source image is mapped to the center point of the destination image. No scaling or resampling is done. Depending on the relative sizes of the source image and destination image, parts of the source image may be clipped, and areas outside the source image (colored with a caller-supplied background color) may appear in the destination image.

The caller supplies:

- A source buffer.
- A destination buffer. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.
- A rotation constant. See “[Rotation Constants for Use With Rotate90 Function](#)” (page 329).
- A background color.
- Flags.

For specific details on the parameters, see “[Geometric Functions](#)” (page 275).

Because no resampling is done—instead, individual pixels are copied unchanged to new locations—this function places certain restrictions on the pixel height and widths of the source and destination buffers, so that it can map the center of the source to the center of the destination precisely. The restrictions are these:

If you are rotating 90 or 270 degrees, the height of the source image and the width of the destination image must both be even or both be odd; and the width of the source image and the height of the destination image must both be even or both be odd.

If you are rotating 0 or 180 degrees, the height of the source image and the destination image must both be even or both be odd; and the width of the source image and the destination image must both be even or both be odd.

If your images do not meet these restrictions, you can use the general (high-level) Rotate function instead.

Set the `kvImageDoNotTile` field in the flags parameter to prevent `vImage` from using tiling internally. (This is appropriate if you are doing tiling yourself.) This function ignores the `kvImageLeaveAlphaUnchanged` flag field.

Horizontal Shear

There are four horizontal shear functions, one for each of the image representations supported by `vImage`: `vImageHorizontalShear_Planar8`, `vImageHorizontalShear_PlanarF`, `vImageHorizontalShear_ARGB8888`, `vImageHorizontalShear_ARGBFFFF`.

This function performs a horizontal shearing operation on a source image and places the result in a destination buffer. In a simple horizontal shearing operation (no scaling or translation), a rectangular region of interest is mapped into a parallelogram whose top and bottom are parallel to the original rectangle, and whose sides are slanted at a particular slope. The height of the parallelogram is the same as the height of the original rectangle, and the width of the parallelogram (the length of any row line) is the same as the width of the original rectangle.

This is done by starting at the bottom of the rectangle and shifting each row of pixels to the right by an amount proportional to that row’s distance from the bottom of the rectangle. The shift amount may be fractional for some rows, and so resampling must be done to assign appropriate pixel values without introducing interference patterns or other artifacts.

`vImage` allows the user to specify the resampling filter. It can either be a default supplied by `vImage`—a Lanczos kernel—or a custom resampling filter supplied by the user. In either case the user passes a `ResamplingFilter` parameter to the horizontal shear function. The `ResamplingFilter` object contains

the information necessary to calculate the resampling filter. It also contains a scale factor, which scales the transformed image in the horizontal direction. See “[Shear Functions and User-defined Resampling Filters](#)” (page 50).

In addition to simple shearing and scaling, the vImage horizontal shearing function also allows the user to specify a horizontal translation value, which shifts the transformed image to the left or the right in the destination buffer.

As the region of interest is sheared, translated, and scaled, source pixels from outside the region of interest (but inside the source image) may appear in the destination buffer. In fact, vImage shears, translates, and scales as much of the source image as it needs to in order to attempt to fill the destination buffer. In addition, areas from outside the source image may appear in the destination image. These are assigned colors using either the background color fill technique or the edge extend technique, depending on a flag setting. See “[Pixels Outside the Image Buffer](#)” (page 21).

The caller supplies:

- A source buffer, and offsets to a point within the source image to define the upper left-hand point of the region of interest.
- A destination buffer. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.
- A translation value for the horizontal direction.
- A slope for the front edge of the shear.
- A resampling filter.
- A background color.
- Flags.

For specific details on the parameters, see “[Geometric Functions](#)” (page 275).

The size (number of rows and number of columns) of the destination buffer is also used to determine the size of the region of interest in the source buffer. The origin of both the region of interest and the destination buffer are assumed to be in the lower left-hand corner. If there is no translation, then the lower left corner of the region of interest (not necessarily of the source image, which may be different) is mapped to the lower left corner of the destination image. (The `vImage_Buffer` data pointer points to the top left corner of the image, as always.)

Set the `kvImageDoNotTile` flag to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.)

Set either the `kvImageBackgroundColorFill` or `kvImageEdgeExtend` flag to specify how the function should handle pixels from outside the source image. You should set the same flag when calling `vImageNewResamplingFilter` or `vImageNewResamplingFilterForFunctionUsingBuffer` to create the resampling filter to be used with this function.

[Figure 5-6](#) (page 61) shows the bluejay of [Figure 5-1](#) (page 53) after a horizontal shear operation with no translation component.

Figure 5-6 Bluejay, after horizontal shear

Vertical Shear

There are four vertical shear functions, one for each of the image representations supported by vImage: `vImageVerticalShear_Planar8`, `vImageVerticalShear_PlanarF`, `vImageVerticalShear_ARGB8888`, `vImageVerticalShear_ARGBFFFF`.

This function performs a vertical shearing operation on a source image and places the result in a destination buffer. In a simple Vertical shearing operation (no scaling or translation), a rectangular region of interest is mapped into a parallelogram whose left and right edges are parallel to the original rectangle, and whose top and bottom edges are slanted at a particular slope. The width of the parallelogram is the same as the width of the original rectangle, and the height of the parallelogram (the height of any column) is the same as the height of the original rectangle.

This is done by starting at the left edge of the rectangle and shifting each row of pixels up by an amount proportional to that row's distance from the bottom of the rectangle. The shift amount may be fractional for some rows, and so resampling must be done to assign appropriate pixel values without introducing interference patterns or other artifacts.

vImage allows the user to specify the resampling filter. It can either be a default supplied by vImage—a Lanczos kernel—or a custom resampling filter supplied by the user. In either case the user passes a `ResamplingFilter` parameter to the vertical shear function. The `ResamplingFilter` object contains the information necessary to calculate the resampling filter. It also contains a scale factor, which scales the transformed image in the vertical direction. See “[Shear Functions and User-defined Resampling Filters](#)” (page 50).

In addition to simple shearing and scaling, the vImage vertical shearing function also allows the user to specify a vertical translation value, which shifts the transformed image up or down in the destination buffer.

As the region of interest is sheared, translated, and scaled, source pixels from outside the region of interest (but inside the source image) may appear in the destination buffer. In fact, vImage shears, translates, and scales as much of the source image as it needs to in order to attempt to fill the destination

buffer. In addition, areas from outside the source image may appear in the destination image. These are assigned colors using either the background color fill technique or the edge extend technique, depending on a flag setting. See “[Pixels Outside the Image Buffer](#)” (page 21).

The caller supplies:

- A source buffer, and offsets to a point within the source image to define the upper left-hand point of the region of interest.
- A destination buffer. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.
- A translation value for the vertical direction.
- A slope for the top edge of the shear.
- A resampling filter.
- A background color.
- Flags.

For specific details on the parameters, see “[Geometric Functions](#)” (page 275).

The size (number of rows and number of columns) of the destination buffer is also used to determine the size of the region of interest in the source buffer. The origin of both the region of interest and the destination buffer are assumed to be in the lower left-hand corner. If there is no translation, then the lower left-hand corner of the region of interest (not necessarily of the source image, which may be different) is mapped to the lower left-hand corner of the destination image.

Set the `kvImageDoNotTile` flag to prevent `vImage` from using tiling internally. (This is appropriate if you are doing tiling yourself.)

Set either the `kvImageBackgroundColorFill` or `kvImageEdgeExtend` flag to specify how the function should handle pixels from outside the source image. You should set the same flag when calling `vImageNewResamplingFilter` or `vImageNewResamplingFilterForFunctionUsingBuffer` to create the resampling filter to be used with this function.

Figure 5-7 shows the bluejay of [Figure 5-1](#) (page 53) after a vertical shear operation.

Figure 5-7 Bluejay, after vertical shear

vImageNewResamplingFilter

This function creates an object of type `ResamplingFilter` that can be passed to a `Shear` function. The resampling filter it encapsulates is the default kernel for `vImage`—currently a Lanczos filter, although this may change in the future.

`vImageNewResamplingFilter` allocates any memory required for the `ResamplingFilter`; you do not need to pass it a buffer. When you are done with this `ResamplingFilter`, call `vImageDestroyResamplingFilter` to deallocate its memory. Do not directly deallocate its memory yourself.

The caller provides:

- A scale factor
- Flags

When the shear function is called, the scale factor is applied to the entire image (in a direction appropriate to the shear function, either horizontal or vertical).

Set the `kvImageHighQualityResampling` flag parameter to create a resampling filter for a Lanczos5 kernel; otherwise resampling filter for a Lanczos3 kernel will be created.

As long as you want to use the same kernel quality and the same scale factor, you can reuse this resampling filter.

vImageDestroyResamplingFilter

This function destroys a `ResamplingFilter` object created by `vImageNewResamplingFilter`, deallocating its memory.

Do not attempt to directly deallocate the memory of a `ResamplingFilter` created by `vImageNewResamplingFilter` yourself.

Do not pass this function a ResamplingFilter created by `vImageNewResamplingFilterForFunctionUsingBuffer`.

The caller provides:

- The ResamplingFilter object

`vImageNewResamplingFilterForFunctionUsingBuffer`

This function creates an object of type ResamplingFilter that can be passed to a shear function. The ResamplingFilter encapsulates a custom resampling filter that you provide.

Before calling this function, you must allocate a buffer large enough to hold the ResamplingFilter object. You can get the necessary size by calling the function `vImageGetResamplingFilterSize`. When you are done with this ResamplingFilter, you must deallocate its memory yourself (and properly dispose of any other objects associated with it).

Do not use the function `vImageDestroyResamplingFilter` on a resampling filter created with `vImageNewResamplingFilterForFunctionUsingBuffer`.

The caller provides:

- A buffer for the ResamplingFilter object (you can get the size of this buffer by calling the function `vImageGetResamplingFilterSize`)
- A scale factor
- A resampling filter function
- A kernel width, which specifies many pixels the kernel function can be required to sample at a time (given a source kernel at position 0, the function can look at pixels from position – kernel width to + kernel width)
- User data, which will be passed to your resampling filter function when it is called (this can be a table, a counter, or anything else that is useful in calculating your resampling filter function; it may also be `NULL`)
- Flags

When the shear function is called, the scale factor is applied to the entire image (in a direction appropriate to the shear function, either horizontal or vertical).

You can use the resampling filter created by this call repeatedly. When your requirements for a resampling filter (including the scale factor or other values) change, you should destroy this resampling filter and create a new one.

`vImageGetResamplingFilterSize`

This utility function is used to obtain the minimum size, in bytes, of a custom ResamplingFilter object created by `vImageNewResamplingFilterForFunctionUsingBuffer`.

The caller supplies:

- The scale factor that will be used in the `vImageNewResamplingFilterForFunctionUsingBuffer` call

- The *kernelFunc* pointer that will be used in the vImageNewResamplingFilterForFunctionUsingBuffer call
- The flags that will be used in the vImageNewResamplingFilterForFunctionUsingBuffer call

For specific details on the parameters, see “[Geometric Functions](#)” (page 275).

Histogram Operations

This chapter describes histograms and the histogram operations provided by vImage.

What Are Histograms?

A histogram is a chart that records the frequency with which pixels of various intensities appear in an image. The histogram gives you information about the spread of intensity values in a image. Is the image mainly dark, or mainly light? Does the image make use of most of the available intensity values, or are its pixels concentrated near just a few intensities? Are certain intensities underrepresented? The histogram represents the answers to these sorts of questions.

Histograms for Images in Integer Formats

The simplest example of a histogram in vImage corresponds to an image in the Planar8 format. Each pixel has exactly one integer value (its intensity), ranging from 0 to 255 inclusive. The histogram for the image would be an array of 256 integers, indexed from 0 to 255. The first element of the array would contain the number of pixels in the image with intensity value 0; the second element would contain the number of pixels with value 1; the third element would contain the number of pixels with value 2; and so on down to the last element, which would contain the number of pixels with value 255.

For images in ARGB8888 (interleaved integer) format, the imaged is separated into four Planar8 images. The histogram is calculated for each planar image separately. The histogram for the interleaved image is an array with four elements, each element containing the histogram for the corresponding planar image, in the order Alpha, Red, Green, Blue.

Histograms For Images In Floating-Point Formats

For images in floating-point formats, the situation is more complicated.

For PlanarF images, each pixel has some floating-point value, often (but not always) in the range 0.0f to 1.0f. It is neither practical or useful to have a histogram element that corresponds to each possible floating-point value. In this case, the caller is allowed to specify how many histogram array elements (or “bins”) there are, and how individual pixels are assigned to the array elements as they are being counted.

This is done by specifying the **histogram definition parameters**: *histogram_entries*, *minVal*, and *maxVal*. *histogram_entries* is an unsigned integer; *minVal* and *maxVal* are pixel values.

The histogram is created as an array with *histogram_entries* elements. The range of floating-point numbers from *minVal* to *maxVal* is divided into *histogram_entries* equal subintervals. When counting the pixels, a pixel whose value lies in the first subintervals is assigned to the first histogram element; a pixel whose value lies in the second subintervals is assigned to the second histogram element; and so on down to pixels whose values lies in the last subinterval, which are assigned to the last element.

Any pixel whose value is less than *minVal* is assigned to the first histogram element. Any pixel whose value is greater than *maxVal* is assigned to the last histogram element. This provides us with clipping outside the range from *minVal* to *maxVal*, for the purposes of calculating the histogram.

In this case, the histogram does not represents the number of pixels at each intensity level, but the number of pixels whose intensity level falls into each subrange. For the purposes of the histogram and all histogram operations, all intensity levels that fall into the same subrange are treated as though they were the same.

For images in ARGBFFFF format, the image is separated into four PlanarF images. The histogram is calculated for each planar image separately. The histogram for the interleaved image is an array with four elements, each element containing the histogram for the corresponding planar image, in the order Alpha, Red, Green, Blue.

Histogram Operations

It is possible to transform an image so that its new histogram has, or at least comes closer to having, certain desired properties. For example, given an image that has a very unequal distribution of pixel intensities, you can transform it so that its new histogram is more uniform, closer in shape to a histogram in which every pixel intensity occurs equally. In general, you can transform an image so that its histogram more closely resembles a desired histogram.

Why can this only be done approximately? There are two reasons. The first is trivial: in general, the two histograms will contain different numbers of total pixels. That is why such operations try to match the shape of the histogram, not the histogram itself. It is the percentage of pixels that appear at each intensity level that is important, not the actual number of pixels.

The second is more serious. All histogram operations are “point” operations: that is, the intensity of a destination pixel depends only on the intensity of the source pixel, modified by values that are the same over the entire image. Therefore, two pixels of the same intensity will always map to two pixels of the same (presumably altered) intensity. If the original image only had pixels with (just for example) 12 different intensity values, the transformed image will have at most 12 different intensity levels represented. You cannot force such an image to have a uniform histogram (or, in general, any specific histogram) by using a point operation.

However, there are point operations that will cause the transformed histogram to more closely resemble a desired histogram. vImage provides four of these: equalization, histogram specification, contrast stretch, and ends-in contrast stretch.

There are a number of reasons to use these operations. An image may not make full use of the possible range of intensity values—for example, most of its pixels may be fairly dark, making it difficult to see details. Changing it so that it has a more uniform histogram can improve contrast. Also, it may be easier to compare two images (with respect to texture or other aspects) if they have the same histogram. This is usually done by changing both images to match some standard histogram.

Lookup Tables (LUTs)

A lookup table (LUT) for a Planar8 image is a function $L(i)$ that maps pixel values into pixel values. That is, if $0 \leq i \leq 255$, then $0 \leq L(i) \leq 255$.

A LUT can be used as a mapping to transform a Planar8 image. Simply map each destination pixel with a value of i to a destination pixel with a value of $L(i)$. The size of the image does not change.

Note that a LUT transformation is a point operation. That is, the value of a destination pixel depends only on the value of the source pixel.

A histogram is not an LUT, because its values may be greater than 255. But LUTs are used in the definitions of several histogram operations.

Histogram Functions

These are the histogram-related functions provided by vImage.

Histogram Calculation

There are four histogram calculation functions, one for each of the image representations supported by vImage: `vImageHistogramCalculation_Planar8`, `vImageHistogramCalculation_PlanarF`, `vImageHistogramCalculation_ARGB8888`, `vImageHistogramCalculation_ARGBFFFF`.

These functions calculate the histogram for a given image.

The caller supplies:

- A source buffer
- A buffer for the calculated histogram
- For the floating-point image formats only (PlanarF and ARGBFFFF), a minimum value, a maximum value, and a number of histogram entries
- Flags

For specific details on the parameters, see “[Histogram Functions](#)” (page 231).

Equalization

There are four equalization functions, one for each of the image representations supported by vImage: vImageEqualization_Planar8, vImageEqualization_PlanarF, vImageEqualization_ARGB8888, vImageEqualization_ARGBFFFF.

These functions transform an image so that it has a more uniform histogram. (A truly uniform histogram is one in which each intensity level occurs with equal frequency. These functions try to approximate that histogram.)

The caller supplies:

- A source buffer.
- A destination buffer. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.
- For the floating-point image formats only (PlanarF and ARGBFFFF), a minimum value, a maximum value, a number of histogram entries, and a temporary buffer used by the function for intermediate results. If you provide NULL for the temporary buffer, the function will allocate the temporary buffer itself (and, of course, deallocate it when it is done), but this may decrease performance and interfere with real-time use.
- Flags.

For specific details on the parameters, see “[Histogram Functions](#)” (page 231).

You can set the `kvImageLeaveAlphaUnchanged` flag to have the alpha channel copied to the destination unchanged (interleaved formats only).

The equalization functions use the following algorithm in the Planar8 case:

1. Calculate the histogram of the source image.
2. Calculate the sum of the histogram. This is also a 256-element array, defined in the following way: element 0 contains histogram element 0. Element 1 contains the sum of histogram element 0 and histogram element 1. Element 2 contains the sum of histogram elements 0, 1, and 2. And so on, down to element 255, which contains the sum of histogram elements 0, 1, 2, 3, ..., 255.
3. Normalize the histogram sum array by multiplying each of its elements by 255/(the total number of pixels in the image). (Round to the nearest integer.)
4. The normalized histogram sum is a LUT. Use it as a mapping to transform the source image. The result is the destination image.

You can set the `kvImageLeaveAlphaUnchanged` flag to have the alpha channel copied to the destination unchanged (interleaved formats only).

The algorithm for the PlanarF format is very similar, although somewhat more complicated because it involves the histogram definition parameters `histogram_entries`, `minVal`, and `maxVal`. This document does not describe that algorithm in detail. For interleaved formats (ARGB8888 and ARGBFFFF), vImage separates the image into four planar formats, operates on each of them, and then recombines them back into an interleaved image.

Figure 6-1 shows an image with a good deal of unevenness in its intensity distribution. Figure 6-2 shows the same image after histogram equalization.

Figure 6-1 Image before histogram equalization



Figure 6-2 Image after histogram equalization



Histogram Specification

There are four histogram specification function, one for each of the image representations supported by vImage: vImageHistogramSpecification_Planar8, vImageHistogramSpecification_PlanarF, vImageHistogramSpecification_ARGB8888, vImageHistogramSpecification_ARGBFFFF.

These functions transform an image so that its histogram more closely resembles a given histogram.

The caller supplies:

- A source buffer.
- A destination buffer. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.
- A desired histogram.
- For the floating-point image formats only (PlanarF and ARGBFFFF), a minimum value, a maximum value, a number of histogram entries, and a temporary buffer used by the function for intermediate results. If you provide NULL, the function will allocate the temporary buffer itself (and, of course, deallocate it when it is done), but this may decrease performance and interfere with real-time use.
- Flags.

For specific details on the parameters, see “[Histogram Functions](#)” (page 231).

You can set the `kvImageLeaveAlphaUnchanged` flag to have the alpha channel copied to the destination unchanged (interleaved formats only).

The histogram specification functions use the following algorithm in the Planar8 case:

1. Equalize the source image. (see “[Equalization](#)” (page 70).)
2. Calculate the histogram equalization mapping (that is, the normalized histogram sum array) for the desired histogram. Call this mapping $H(i)$.
3. Invert $H(i)$. That is, for each value j ($0 \leq j \leq 255$), find the index i such that $H(i)$ is closest to j . (If there are several values of i that yield $H(i)$ values equally close to j , use the smallest value of i that qualifies.) This calculates the inverse equalization function $I(j)$.
4. The inverse equalization function $I(j)$ is a LUT. Use it as a mapping to transform the source image. The result is the destination image.

The algorithm for the PlanarF format is very similar, although somewhat more complicated because it involves the histogram definition parameters `histogram_entries`, `minVal`, and `maxVal`. This document does not describe that algorithm in detail. For interleaved formats (ARGB8888 and ARGBFFFF), vImage separates the image into four planar formats, operates on each of them, and then recombines them back into an interleaved image.

Contrast Stretch

There are four contrast stretch functions, one for each of the image representations supported by vImage: vImageContrastStretch_Planar8, vImageContrastStretch_PlanarF, vImageContrastStretch_ARGB8888, vImageContrastStretch_ARGBFFFF.

These functions transform an image so that its intensity values stretch out along the full range of intensity values. It is best used on images in which all the pixels are concentrated in one area of the intensity spectrum, and intensity values outside that area are not represented.

The caller supplies:

- A source buffer.
- A destination buffer. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.
- For the floating-point image formats only (PlanarF and ARGBFFFF), a minimum value, a maximum value, a number of histogram entries, and a temporary buffer used by the function for intermediate results. If you provide NULL, the function will allocate the temporary buffer itself (and, of course, deallocate it when it is done), but this may decrease performance and interfere with real-time use.
- Flags.

For specific details on the parameters, see “[Histogram Functions](#)” (page 231).

The Contrast Stretch functions in the Planar8 case map existing pixels to new ones according to the following formula, where i is the old pixel intensity:

```
new pixel intensity = 255 * (i - low)/(high - low)
```

`high` refers to the maximum intensity of any pixel in the image. `low` refers to the minimum intensity of any pixel in the image. By subtracting the `low` value, you shift the histogram all the way to the left of the intensity spectrum; then you stretch it out to cover the entire dynamic range.

The algorithm for the PlanarF format is very similar, although somewhat more complicated because it involves the histogram definition parameters `histogram_entries`, `minVal`, and `maxVal`. This document does not describe that algorithm in detail. For interleaved formats (ARGB8888 and ARGBFFFF), vImage separates the image into four planar formats, operates on each of them, and then recombines them back into an interleaved image.

Ends-In Contrast Stretch

There are four versions of the ends-in contrast stretch functions, one for each of the image representations supported by vImage: vImageEndsInContrastStretch_Planar8, vImageEndsInContrastStretch_PlanarF, vImageEndsInContrastStretch_ARGB8888, vImageEndsInContrastStretch_ARGBFFFF.

These functions are a more complex version of the contrast stretch functions. They are best used on images that have some pixels at or near the lowest and highest values of the intensity spectrum, but whose histogram is still mainly concentrated in one area. The ends-in contrast stretch functions map all intensities less than or equal to a certain level to 0; all intensities greater than or equal to a certain

level to 255; and perform a contrast stretch on all the values in between. The low and high levels are not defined directly by two given intensity values, but by percentages: the ends-in contrast stretch operation must find intensity levels such that a certain percent of pixels are below one of the intensity values, and a certain percent are above the other intensity value.

The caller supplies:

- A source buffer.
- A destination buffer.
- a *percent_low* and a *percent_high*.
- For the floating-point image formats only (PlanarF and ARGBFFFF), a minimum value, a maximum value, a number of histogram entries, and a temporary buffer used by the function for intermediate results. If you provide NULL, the function will allocate the temporary buffer itself (and, of course, deallocate it when it is done), but this may decrease performance and interfere with real-time use.
- Flags.

For specific details on the parameters, see “[Histogram Functions](#)” (page 231).

The ends-in contrast stretch functions use the following algorithm in the Planar8 case:

1. Calculate the histogram of the source image.
2. Calculate the sum of the histogram. (See “[Equalization](#)” (page 70).) Call this function $S(i)$.
3. Find the least number i such that $(S(i) * 100.0) / (\text{total number of pixels}) \geq \text{percent_low}$. Call this $i_{\text{low_threshold}}$.
4. Find the greatest number i such that $((\text{total number of pixels}) - S(i)) * 100.0 / (\text{total number of pixels}) \geq \text{percent_high}$. Call this $i_{\text{high_threshold}}$.
5. Create an LUT $E(i)$ according to the following formula:

$$\begin{aligned} E(i) &= 0 \text{ for } i \leq \text{low_threshold} \\ E(i) &= 255 \times (i - \text{low_threshold}) / (\text{high_threshold} - \text{low_threshold}) \text{ for } \text{low_Threshold} < i < \text{high_threshold} \\ E(i) &= 255 \text{ for } i \geq \text{high_threshold} \end{aligned}$$

6. Use $E(i)$ as a mapping to transform the source image. The result is the destination image.

The algorithm for the PlanarF format is very similar, although somewhat more complicated because it involves the histogram definition parameters *histogram_entries*, *minVal*, and *maxVal*. This document does not describe that algorithm in detail. For interleaved formats (ARGB8888 and ARGBFFFF), vImage separates the image into four planar formats, operates on each of them, and then recombines them back into an interleaved image.

You can set the `kvImageLeaveAlphaUnchanged` flag to have the alpha channel copied to the destination unchanged (interleaved formats only).

Figure 6-3 and Figure 6-4 show an image before and after transformation by an ends-in contrast stretch operation with `low_threshold = 20%`, `high_threshold = 0%`.

Figure 6-3 Space shuttle before ends-in contrast stretch



Figure 6-4 Space shuttle after ends-in contrast stretch



vImageGetMinimumBufferSizeForHistogram

This function is deprecated in Mac OS X v10.4. Instead, pass the `kvImageGetTempBufferSize` flag to any `vImage` function that takes a temporary buffer as a parameter; the function will return the minimum size, in bytes, of the temporary buffer (and will not do anything else). Please see “[Temporary Buffers for vImage Functions](#)” (page 26). Note that if the result is negative, it is an error code.

This utility function is used to obtain the necessary size for the temporary buffer used by some of the histogram functions (in particular, the floating-point versions of the histogram transformation functions).

The caller supplies:

- The source buffer. This should be the same as will be used in the function call.
- The destination buffer. These should be the same as will be used in the function call.
- The `histogram_entries` value that will be used in the function call.
- The size of a pixel, in bytes (this depends on the image format you are using).

For specific details on the parameters, see “[Histogram Functions](#)” (page 231).

This function does not depend on the `data` or `rowBytes` fields of the source or destination buffer; it only uses the `height` and `width` fields from those parameters. If the size of the images you are processing stay the same, then the required size of the buffer will also stay the same. The function is designed to make it easy to reuse a buffer if your size of your images do not increase from one call to the next. See “[Temporary Buffers for vImage Functions](#)” (page 26).

Alpha Compositing Operations

This chapter describes the vImage functions for alpha compositing.

What Is Alpha Compositing?

In many modern computer graphics systems, each pixel will have an associated alpha value that indicates how opaque the pixel is. Multiple images can be layered on top of each other, with the alpha value for a pixel in a given layer indicating what fraction of the colors from lower layers can be seen through the color at the given level.

The fundamental operation in this area is alpha compositing (also known as alpha blending). In alpha compositing, a top image (with alpha information) and a bottom image (with its own alpha information) are combined to create a composite image representing what would be seen if the top image were placed over the bottom image. The composite image should have alpha information of its own, derived from the alpha values of the two images.

The interleaved formats used by vImage—ARGB888 and ARGBFFFF—use one of their four channels to contain the alpha value for each pixel. The planar formats lack this extra channel. To do alpha compositing with planar images, you need to have alpha information for those images supplied separately (generally in the same format as the planar color channel).

All of the vImage alpha compositing functions work “in place.” That is, source and destination images may occupy the same memory if they are strictly aligned pixel for pixel.

All of the alpha compositing functions may do tiling and/or multithreading internally. If you do your own tiling and/or multithreading, use the `kVImageDoNotTile` flag to prevent this. Please see “[Tiling for Cache Utilization](#)” (page 23).

Premultiplied Alpha Versus Non-premultiplied Alpha

For computational efficiency, it is often useful to premultiply the non-alpha pixel values by the alpha value (expressed as a fraction from 0 to 1, inclusive), and use these premultiplied values as the values of the pixels themselves in the non-alpha channels. For floating-point formats, the two values can be multiplied directly. For integer formats, in which both values are in the range 0 to 255, they are multiplied, then the result is scaled down so that it lies in the 0 to 255 range. The scaling calculation is

```
scaledColor = (alpha * color + 127) / 255
```

This process makes compositing faster—especially multiple compositing, where there are more than two images involved. When using premultiplied alpha, you still want to maintain the original alpha information, so that you can retrieve the original, non-premultiplied values of the pixels when you need them. It is also required for the bottom layer in a compositing operation.

vImage provides functions for alpha compositing for both the premultiplied alpha case and the non-premultiplied alpha case. Mac OS X v10.4 adds some alpha compositing functions for common mixed cases.

Alpha Compositing Functions

Non-premultiplied Alpha Blend

There are four alpha blend (non-premultiplied) functions, one for each of the image representations supported by vImage: vImageAlphaBlend_Planar8, vImageAlphaBlend_PlanarF, vImageAlphaBlend_ARGB8888, vImageAlphaBlend_ARGBFFFF.

For floating-point formats, the non-premultiplied compositing calculation for each color channel is

```
resultAlpha = topAlpha + (1.0f - topAlpha) * bottomAlpha
resultColor = (topAlpha * topColor + (1.0f - topAlpha)
               * bottomAlpha * bottomColor) / resultAlpha
```

For 8-bit formats, the calculation is

```
resultAlpha = (topAlpha * 255 + (255 - topAlpha)
               * bottomAlpha + 127) / 255
resultColor = (topAlpha * topColor + (((255 - topAlpha)
               * bottomAlpha + 127) / 255) * bottomColor + 127)
               / resultAlpha
```

Alpha Blend—Planar

The vImageAlphaBlend_Planar8 and vImageAlphaBlend_PlanarF functions perform non-premultiplied alpha compositing of two planar images.

The caller supplies:

- A top image.
- Alpha information for the top image.
- A bottom image.
- Alpha information for the bottom image.
- Alpha information for the composite image.
- A destination buffer. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

- Flags.

For specific details on the parameters, see “[Alpha Compositing Functions](#)” (page 117).

In order to calculate the alpha information for the composite image—called the **composite alpha**—you must use `vImagePremultipliedAlphaBlend_Planar8` or `vImagePremultipliedAlphaBlend_PlanarF`, as appropriate. To calculate the composite alpha for the Planar8 case, call

```
vImagePremultipliedAlphaBlend_Planar8( srcTopAlpha, srcTopAlpha,
srcBottomAlpha, alpha, kvImageNoFlags );
```

Where `scrTopAlpha`, `srcBottomAlpha`, and `alpha` are the parameters that are going to be passed to `vImageAlphaBlend_Planar8`. (`alpha`, in this case, is a destination buffer that will be set to the calculated composite alpha by the `vImagePremultipliedAlphaBlend_Planar8` call, and then used as an input buffer in the `vImageAlphaBlend_Planar8` call.) To calculate the composite alpha for the PlanarF case, call

```
vImagePremultipliedAlphaBlend_PlanarF( srcTopAlpha, srcTopAlpha,
srcBottomAlpha, alpha, kvImageNoFlags );
```

Where `scrTopAlpha`, `srcBottomAlpha`, and `alpha` are the parameters that are going to be passed to `vImageAlphaBlend_PlanarF`. (`alpha` is a destination buffer to the `vImagePremultipliedAlphaBlend_PlanarF` call, then an input buffer to `vImageAlphaBlend_PlanarF`.)

Note that `scrTopAlpha` is used twice as an input in the `vImagePremultipliedAlphaBlend_Planar8` and `vImagePremultipliedAlphaBlend_PlanarF` calls used to calculate the composite alpha.

Why don’t the non-premultiplied planar alpha blend functions generate the composite alpha themselves, rather than requiring the caller to go through this extra step? The reason is simple: very often, when you do planar alpha compositing, you will in fact do it three times, once each for red, green, and blue. But the composite alpha is the same in all three cases. It would be very wasteful to have the function calculate the composite alpha every time. So the caller calculates it once, in advance.

Set the `kvImageDoNotTile` field in the flags parameter to prevent `vImage` from using tiling internally. (This is appropriate if you are doing tiling yourself.)

Alpha Blend—Interleaved

The `vImageAlphaBlend_ARGB8888` and `vImageAlphaBlend_ARGBFFFF` functions perform non-premultiplied alpha compositing on two interleaved (4-channel) images.

The caller supplies:

- A top image.
- A bottom image.
- A destination buffer. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.
- Flags.

For specific details on the parameters, see “[Alpha Compositing Functions](#)” (page 117).

Set the `kvImageDoNotTile` field in the `flags` parameter to prevent `vImage` from using tiling internally. (This is appropriate if you are doing tiling yourself.)

Premultiplied Alpha Blend

There are four premultiplied alpha blend functions, one for each of the image representations supported by `vImage`: `vImagePremultipliedAlphaBlend_Planar8`, `vImagePremultipliedAlphaBlend_PlanarF`, `vImagePremultipliedAlphaBlend_ARGB8888`, `vImagePremultipliedAlphaBlend_ARGBFFFF`.

For floating-point formats, the premultiplied compositing calculation for each color channel is

```
resultAlpha = topAlpha + (1.0f - topAlpha) * bottomAlpha
resultColor = topColor + (1.0f - topAlpha) * bottomColor
```

For 8-bit formats, the calculation is

```
resultAlpha = ((topAlpha * 255 + (255 - topAlpha)
               * bottomAlpha) + 127) / 255
resultColor = (topColor * 255 + (255 - topAlpha)
               * bottomColor + 127) / 255
```

Alpha Blend—Planar

The `vImagePremultipliedAlphaBlend_Planar8` and `vImagePremultipliedAlphaBlend_PlanarF` functions perform premultiplied alpha compositing of two planar images.

The caller supplies:

- A top image.
- The original alpha information for the top image.
- A bottom image.
- A destination buffer. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.
- Flags.

For specific details on the parameters, see “[Alpha Compositing Functions](#)” (page 117).

Set the `kvImageDoNotTile` field in the `flags` parameter to prevent `vImage` from using tiling internally. (This is appropriate if you are doing tiling yourself.)

Even though the alpha values have been premultiplied into the pixel values, the function also requires the original alpha information for the top image to do its calculations. There is no way to extract this information from the premultiplied planar values.

Premultiplied Alpha Blend—Interleaved

The `vImagePremultipliedAlphaBlend_ARGB8888` and `vImagePremultipliedAlphaBlend_ARGBFFFF` functions perform premultiplied alpha compositing on two interleaved (4-channel) images.

The caller supplies:

- A top image.
- A bottom image.
- A destination buffer. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.
- Flags.

For specific details on the parameters, see “[Alpha Compositing Functions](#)” (page 117).

Set the `kvImageDoNotTile` field in the flags parameter to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.)

Since the original alpha values are maintained in the interleaved data, the caller does not need to provide them.

Non-premultiplied to Premultiplied Alpha Blend

There are four non-premultiplied to premultiplied alpha blend functions, one for each of the image representations supported by vImage:

`vImageAlphaBlend_NonpremultipliedToPremultiplied_Planar8`,
`vImageAlphaBlend_NonpremultipliedToPremultiplied_PlanarF`,
`vImageAlphaBlend_NonpremultipliedToPremultiplied_ARGB8888`,
`vImageAlphaBlend_NonpremultipliedToPremultiplied_ARGBFFFF`.

The top image is non-premultiplied, the bottom image is premultiplied, and the result is premultiplied.

For floating-point formats, the non-premultiplied to premultiplied compositing calculation for each color channel is

```
resultAlpha = topAlpha + (1.0 - topAlpha) * bottomAlpha
resultColor = topAlpha * topColor + (1.0 - topAlpha) * bottomColor
```

For 8-bit formats, the calculation is

```
resultAlpha = (topAlpha * 255 + (255 - topAlpha)
               * bottomAlpha + 127) / 255
resultColor = (topAlpha * topColor + (255 - topAlpha)
                * bottomColor + 127) / 255
```

Non-premultiplied to Premultiplied Alpha Blend—Planar

The `vImageAlphaBlend_NonpremultipliedToPremultiplied_Planar8` and `vImageAlphaBlend_NonpremultipliedToPremultiplied_PlanarF` functions perform premultiplied alpha compositing of two planar images.

The caller supplies:

- A top image, non-premultiplied.
- The original alpha information for the top image.
- A bottom image, premultiplied.

- A destination buffer. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.
- Flags.

For specific details on the parameters, see “[Alpha Compositing Functions](#)” (page 117).

Set the `kvImageDoNotTile` field in the flags parameter to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.)

The top image is non-premultiplied and the bottom and destination images are premultiplied. These functions will work in place if the source and destination buffers overlap exactly.

The top and bottom source buffers must be at least as wide and at least as high (in pixels) as the destination buffer.

Non-premultiplied to Premultiplied Alpha Blend—Interleaved

The `vImageAlphaBlend_NonpremultipliedToPremultiplied_ARGB8888` and `vImageAlphaBlend_NonpremultipliedToPremultiplied_ARGBFFFF` functions perform non-premultiplied to premultiplied alpha compositing on two interleaved (4-channel) images.

The caller supplies:

- A top image, non-premultiplied.
- A bottom image, premultiplied.
- A destination buffer. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.
- Flags.

For specific details on the parameters, see “[Alpha Compositing Functions](#)” (page 117).

Set the `kvImageDoNotTile` field in the flags parameter to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.)

The top image is non-premultiplied and the bottom and destination images are premultiplied. These functions will work in place if the source and destination buffers overlap exactly.

The top and bottom source buffers must be at least as wide and at least as high (in pixels) as the destination buffer.

Premultiplied Constant Alpha Blend

There are four premultiplied constant alpha blend functions, one for each of the image representations supported by vImage: `vImagePremultipliedConstAlphaBlend_Planar8`, `vImagePremultipliedConstAlphaBlend_PlanarF`, `vImagePremultipliedConstAlphaBlend_ARGB8888`, `vImagePremultipliedConstAlphaBlend_ARGBFFFF`.

These functions are similar to the premultiplied alpha blend functions, but use a single alpha value over the whole image. For floating-point formats, the premultiplied compositing calculation for each channel is

```
resultAlpha = topAlpha * constantAlpha + (1.0f - topAlpha)
             * constantAlpha * bottomAlpha
resultColor = topColor * constantAlpha + (1.0f - topAlpha)
             * constantAlpha * bottomColor
```

For 8-bit formats, the calculation is

```
resultAlpha = (topAlpha * constantAlpha + (((255 - topAlpha)
                                         * constantAlpha + 127)/255) * bottomAlpha + 127) / 255
resultColor = (topColor * constantAlpha + (((255 - topAlpha)
                                         * constantAlpha + 127)/255) * bottomColor + 127) / 255
```

Constant Alpha Blend—Planar

The `vImagePremultipliedConstAlphaBlend_Planar8` and `vImagePremultipliedConstAlphaBlend_PlanarF` functions perform premultiplied constant alpha compositing of two planar images.

The caller supplies:

- A top image.
- An alpha pixel value.
- The original alpha information for the top image.
- A bottom image.
- A destination buffer. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.
- Flags.

For specific details on the parameters, see “[Alpha Compositing Functions](#)” (page 117).

Set the `kvImageDoNotTile` field in the flags parameter to prevent `vImage` from using tiling internally. (This is appropriate if you are doing tiling yourself.)

Even though the alpha values have been premultiplied into the pixel values, the function also requires the original alpha information for the top image to do its calculations. There is no way to extract this information from the premultiplied planar values.

Premultiplied Constant Alpha Blend—Interleaved

The `vImagePremultipliedConstAlphaBlend_ARGB8888` and `vImagePremultipliedConstAlphaBlend_ARGBFFFF` functions perform premultiplied constant alpha compositing on two interleaved (4-channel ARGB) images.

The caller supplies:

- A top image.
- An alpha pixel value.

- A bottom image.
- A destination buffer. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.
- Flags.

For specific details on the parameters, see “[Alpha Compositing Functions](#)” (page 117).

Set the `kvImageDoNotTile` field in the flags parameter to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.)

Since the original alpha values are maintained in the interleaved data, the caller does not need to provide them.

Unpremultiply Data

There are four unpremultiply data functions, one for each of the main image representations supported by vImage: `vImageUnpremultiplyData_Planar8`, `vImageUnpremultiplyData_PlanarF`, `vImageUnpremultiplyData_ARGB8888`, `vImageUnpremultiplyData_ARGBFFFF`.

For floating-point formats, the unpremultiply calculation is

```
resultAlpha = sourceAlpha
resultColor = sourceColor / sourceAlpha
```

For 8-bit formats, the calculation is

```
resultAlpha = sourceAlpha
resultColor = (255 * sourceColor) / sourceAlpha
```

(For interleaved formats, `sourceAlpha` is taken from the source image. For planar formats, an alpha plane is supplied as a parameter.)

In addition, the `vImageUnpremultiplyData_RGBA8888` and `vImageUnpremultiplyData_RGBAFFFF` functions support the RGBA8888 and RGBAFFFF formats.

These functions take an image in premultiplied form and transform it into an image in non-premultiplied form.

The caller supplies:

- A source image, in premultiplied form.
- For planar formats only, the alpha information for the image. The interleaved formats contain their own alpha channel.
- A destination buffer. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.
- Flags.

For specific details on the parameters, see “[Alpha Compositing Functions](#)” (page 117).

Set the `kvImageDoNotTile` field in the flags parameter to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.)

The planar formats require alpha information so that the premultiplied data can be divided by it, yielding the non-premultiplied data.

Premultiply Data

There are four premultiply data functions, one for each of the image representations supported by vImage: vImagePremultiplyData_Planar8, vImagePremultiplyData_PlanarF, vImagePremultiplyData_ARGB8888, vImagePremultiplyData_ARGBFFFF.

In addition, the vImagePremultiplyData_RGBA8888 and vImagePremultiplyData_RGBA8888 functions support the RGBA8888 and RGBAFFFF formats.

These functions take an image in non-premultiplied form and transform it into an image in premultiplied form.

For floating-point formats, the premultiply calculation is

```
resultAlpha = sourceAlpha
resultColor = sourceColor * sourceAlpha
```

For 8-bit formats, the calculation is

```
resultAlpha = sourceAlpha
resultColor = (sourceAlpha * sourceColor + 127) / 255
```

(For interleaved formats, sourceAlpha is taken from the source image. For planar formats, an alpha plane is supplied as a parameter.)

The caller supplies:

- A source image, in premultiplied form.
- For planar formats only, the alpha information for the image. The interleaved formats contain their own alpha channel.
- A destination buffer. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.
- Flags.

For specific details on the parameters, see “[Alpha Compositing Functions](#)” (page 117).

Set the `kvImageDoNotTile` field in the flags parameter to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.)

Clip Color Values to Alpha

There are four functions to clip color values to the corresponding alpha values, one for each of the image representations supported by vImage: vImageClipToAlpha_Planar8, vImageClipToAlpha_PlanarF, vImageClipToAlpha_ARGB8888, vImageClipToAlpha_ARGBFFFF.

These functions clip each color value in a pixel to the corresponding alpha value; that is, if the color value is greater than the alpha value, it is set to the alpha value. The calculation is

```
resultAlpha = sourceAlpha  
resultColor = MIN(sourceColor, sourceAlpha)
```

The caller supplies:

- A source image.
- For planar formats only, the alpha information for the image. The interleaved formats contain their own alpha channel.
- A destination buffer. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.
- Flags.

For specific details on the parameters, see “[Alpha Compositing Functions](#)” (page 117).

Set the `kvImageDoNotTile` field in the flags parameter to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.)

Image Transformation Operations

This chapter describes the vImage functions for image transformations.

What Are Image Transformations?

Like the convolution functions, the image transformation functions alter the values of the pixels in an image. Unlike the convolutions, these functions do not depend on the values of nearby pixels.

These functions fall into five broad groups:

- Functions to multiply the pixels by a matrix.
- Gamma correction functions.
- Functions to apply a set of piecewise polynomials to the pixels.
- Functions to apply the ratios of two sets of piecewise polynomials to the pixels.
- Functions to apply a lookup table to the pixels.

Matrix Multiplication Functions

There are four matrix multiplication functions, one for each of the image formats:
`vImageMatrixMultiply_Planar8`, `vImageMatrixMultiply_PlanarF`,
`vImageMatrixMultiply_ARGB8888`, and `vImageMatrixMultiply_ARGBFFFF`.

The caller supplies:

- A source image. Depending on the format, this is either an interleaved image or a set of image planes.
- A destination image. Depending on the format, this is either an interleaved image or a set of image planes.
- A matrix of floating-point or integer values. For M channels in the source image and N channels in the destination, this is an $N \times M$ matrix. For the interleaved ARGB formats, N and M must both be 4.

- For the integer image formats only (Planar8 and ARGB8888), a divisor for normalization.
- An array of integer “pre-bias” values which are added to the source values before the matrix multiplication. `NULL` may be passed, which case no pre-bias is added.
- An array of integer “post-bias” values which are added to the destination values after the matrix multiplication but before any clipping, rounding, or division. `NULL` may be passed, in which case the correct value for normal rounding will be used (divisor/2 for integer data, or 0.0 for floating point).
- Flags.

For specific details on the parameters, see [“Image Transformation Functions: Matrix Multiplication”](#) (page 218)

Be aware that 32-bit signed accumulators are used for integer code. If the sum over any matrix column is greater than ± 223 , then overflow may occur. Generally speaking this will not happen because the matrix elements are 16-bit integers, so it would take more than 256 source buffers before trouble could arise.

Matrix multiplication can be used for various purposes, such as color space conversion, color to greyscale conversion, and “color twisting.” As an example, to convert RGB to YUV, one might use the following formulas:

$$\begin{aligned}Y &= ((66 * R + 129 * G + 25 * B + 128) \gg 8) + 16 \\U &= ((-38 * R - 74 * G + 112 * B + 128) \gg 8) + 128 \\V &= ((112 * R - 94 * G - 18 * B + 128) \gg 8) + 128\end{aligned}$$

This translates to a matrix that looks like this:

$$\begin{pmatrix} 66 & -38 & 112 \\ 129 & -74 & -94 \\ 25 & 112 & -18 \end{pmatrix}$$

There is also the `>>8` operation and the extra terms to be dealt with. For integer data, you would use a divisor of 256 to account for the `>>8` operation. The divisor is applied last after all other multiplications and additions. (For floating point, there is no divisor. Just divide the whole matrix by the divisor if one is needed.)

Use the post-bias to handle the +128 and {+16, +128, +128} terms. Since the second set happen after the divisor in the formula above, but our `post_bias` is applied before the divide, you'll need to multiply those biases by the divisor. This will give a `post_bias` of:

$$\{16 * 256 + 128, 128 * 256 + 128, 128 * 256 + 128\} = \{4224, 32896, 32896\}$$

Finally, if there is an alpha component, such that you wish to convert ARGB to AYUV, leaving the alpha component unchanged then add another row and column:

$$\begin{pmatrix} \text{divisor} & 0 & 0 & 0 \\ 0 & 66 & -38 & 112 \\ 0 & 129 & 13 & -94 \\ 0 & -125 & 112 & -18 \end{pmatrix}$$

```
post_bias = { divisor/2, 4224, 32896, 32896 }
```

```
divisor = 256
```

Integer results out of range of 0...255 will be subject to saturated clipping just before writing to the destination buffer (after all other arithmetic).

Gamma Correction

A **gamma function** is used to correct the brightness profile of an image by multiplying each pixel by the value of the function. This is typically done in order to prepare the image for display or printing on a particular device.

vImage provides a set of “gamma correction functions” to apply gamma functions to planar images. Two other groups of functions are closely related to these functions: the “[Piecewise Polynomial Functions](#)” (page 91) and the “[Lookup Table Functions](#)” (page 93).

Types of Gamma Functions

The simplest gamma function is the power function

```
result = pow(fabs(pixelvalue), gamma) * sign
```

where `pixelvalue` is the original pixel value and `sign` is either 1 or -1 depending on the sign of `pixelvalue`.

In practice, this can be calculated in various ways, depending on the required precision and the input values. The value of `gamma` may be passed by the caller, or may be implicit in the `type` of the gamma function.

vImage supports several types of gamma function:

- The full-precision gamma function, using a caller-supplied gamma value (as described above). This guarantees full floating-point precision, and should produce results similar to calling the math library `pow` function, except perhaps 10 times faster.
- The half-precision gamma function, using a caller-supplied gamma value. This guarantees 12-bit precision and is intended for use with data that will eventually be converted to 8-bit integers. With a gamma value in the range from 0.1 to 10, the half-precision calculation is likely to be another order of magnitude faster than full precision.
- A set of gamma function types do half-precision calculation using predefined gamma values for common cases. These are faster still.

In half precision, the absolute value of the error due to lack of precision will be less than 1/4096. This means that for results close to 0, the error may be larger than the correct result. However, 1/4096 equals 0.24% of full white light, so this maximum error should not a problem for most image processing applications. If greater precision is needed, use the full-precision functions.

For further details on the behavior of the gamma function types, see “[Image Transformation Functions: Gamma Corrections](#)” (page 222).

Creating and Destroying Gamma Function Objects

The type `GammaFunction` is a pointer to an opaque object that encapsulate a gamma value, a gamma function type as described above, and vImage flags. Once created, a `GammaFunction` pointer can be used in one or more calls to the gamma correction functions, and then destroyed when no longer needed. It may be used by multiple threads concurrently.

For the `vImageCreateGammaFunction` function, the caller supplies:

- A gamma value of type `float`.
- An integer constant that selects the type of gamma function.
- An integer containing flags.

A `GammaFunction` pointer is returned.

For the `vImageDestroyGammaFunction`, the caller supplies:

- A `GammaFunction` pointer.

For specific details on the parameters, see “[Image Transformation Functions: Gamma Corrections](#)” (page 222).

The object is destroyed.

Gamma Correction Functions

There are three gamma correction functions:

- `vImageGamma_Planar8toPlanarF` takes a source image in Planar8 format and produces an output image in PlanarF format.
- `vImageGamma_PlanarFtoPlanar8` takes a source image in PlanarF format and produces an output image in Planar8 format.
- `vImageGamma_PlanarF` has PlanarF images for both input and output, and will work in place.

For each of these functions, the caller supplies:

- Source `vImage_Buffer` pointer.
- Destination `vImage_Buffer` pointer.
- A `GammaFunction` pointer.
- An integer containing flags.

For specific details on the parameters, see “[Image Transformation Functions: Gamma Corrections](#)” (page 222).

Note: These functions may be internally multithreaded. If you are doing your own multithreading, pass the `kvImageDoNotTile` flag to prevent internal multithreading.

Piecewise Polynomial Functions

The piecewise polynomial functions are similar to the gamma correction functions, but instead of applying a predefined gamma function they apply one or more polynomials supplied by the caller. *The number of polynomials must be an integer power of 2, and they must all be of the same order.*

This allows any reasonable correction function to be approximated by breaking up the range of input pixel values into contiguous subranges, and applying a different polynomial function to each segment. For example, if the range of input values is $0.0 \leq \text{value} \leq 1.0$, it could be broken into two contiguous subranges: $0.0 \leq \text{value} < 0.5$ and $0.5 \leq \text{value} \leq 1.0$. Two polynomials could be defined to process these subranges. Then a pixel value in the first subrange would be processed by the first polynomial, and a pixel in the second subrange by the second polynomial.

The subranges of values are specified by passing an array of boundary values, sorted by increasing value. If there are N subranges (and N polynomials), there are $N + 1$ boundaries.

The first boundary is the lowest value in the total range; input values lower than this will be clipped to this boundary.

The last boundary is the highest value in the total range; input values higher than this will be clipped to this boundary.

The boundaries between the first and last separate the subranges from each other. Each boundary is the lowest value in its subrange.

Each polynomial is specified as an array of floating-point coefficients; these arrays are passed to the function as an array of arrays. The first polynomial applies to the first subrange, the second to the second, and so forth.

There are three piecewise polynomial functions:

- `vImagePiecewisePolynomial_Planar8toPlanarF` takes a source image in Planar8 format and produces an output image in PlanarF format.
- `vImagePiecewisePolynomial_PlanarFtoPlanar8` takes a source image in PlanarF format and produces an output image in Planar8 format.
- `vImagePiecewisePolynomial_PlanarF` has PlanarF images for both input and output, and will work in place.

For each of these functions, the caller supplies:

- Source `vImage_Buffer` pointer.
- Destination `vImage_Buffer` pointer.
- An array of arrays of floating-point coefficients—one array of coefficients for each polynomial. The number of polynomials must be an integer power of 2.

- An array of floating-point boundary values, separating the range of possible pixel values into discrete subranges. If there are N polynomials, there are N + 1 boundaries to define N subranges.
- An integer giving the order of the polynomials.
- An integer giving the log base 2 of the number of polynomials.
- An integer containing flags. The kvImageDoNotTile flag is honored.

For specific details on the parameters, see “[Image Transformation Functions: Gamma Corrections](#)” (page 222).

The ordering of the coefficient and boundary arrays is important; see “[Image Transformation Functions: Gamma Corrections](#)” (page 222) for details.

Piecewise Rational Function

The piecewise rational function `vImagePiecewiseRational_PlanarF` similar to `vImagePiecewisePolynomial_PlanarF`, but instead of using one set of polynomials, it uses two sets, the “top” set and the “bottom” set. *The two sets must contain the same number of polynomials, which must be an integer power of 2, and all polynomials must be of the same order.*

Please see the description of `vImagePiecewisePolynomial_PlanarF` in “[Piecewise Polynomial Functions](#)” (page 91).

For each pixel, the new value is calculated by evaluating the appropriate polynomial from the top set (depending on which subrange the pixel value falls into), then evaluating the corresponding polynomial from the bottom set, then dividing the top value by the bottom value.

The caller supplies:

- Source `vImage_Buffer` pointer.
- Destination `vImage_Buffer` pointer.
- An array of arrays of floating-point coefficients—one array of coefficients for each polynomial. The number of polynomials must be an integer power of 2.
- An array of floating-point boundary values, separating the range of possible pixel values into discrete subranges. If there are N polynomials, there are N + 1 boundaries to define N subranges.
- An integer giving the order of the polynomials.
- An integer giving the log base 2 of the number of polynomials.
- An integer containing flags. The `kvImageDoNotTile` flag is honored.

For specific details on the parameters, see “[Image Transformation Functions: Gamma Corrections](#)” (page 222).

The ordering of the coefficient and boundary arrays is important; see “[Image Transformation Functions: Gamma Corrections](#)” (page 222) for details.

Lookup Table Functions

The lookup table functions are like the piecewise polynomial function, but instead of applying a polynomial they use a lookup table supplied by the caller. There are three lookup table functions:

- `vImageLookupTable_Planar8toPlanarF` uses a table of 256 floating-point values to map 8-bit pixels to floating-point pixels.
- `vImageLookupTable_PlanarFtoPlanar8` uses a table of 4096 8-bit values to map floating-point pixels to 8-bit pixels.
- `vImageInterpolatedLookupTable_PlanarF` uses a table of an arbitrary number of floating-point values to map floating-point pixels to floating-point pixels.

For `vImageLookupTable_Planar8toPlanarF`, the caller supplies:

- Source `vImageBuffer` pointer.
- Destination `vImage_Buffer` pointer.
- A pointer to an array of 256 floating-point values.
- An integer containing flags. No flags are honored; you should pass 0.

For specific details on the parameters, see “[Image Transformation Functions: Gamma Corrections](#)” (page 222).

For each pixel, the 8-bit value from the source Planar8 image is used as an index to get a floating-point value from the table. This value is used as the corresponding pixel in the PlanarF result image. For more details, see “[vImageLookupTable_Planar8ToPlanarF](#)” (page 229). This operation is about as fast as a 13th-order polynomial, when done on a G4 processor.

For `vImageLookupTable_Planarf8toPlanar8`, the caller supplies:

- Source `vImageBuffer` pointer.
- Destination `vImage_Buffer` pointer.
- A pointer to an array of 4096 8-bit values.
- An integer containing flags. No flags are honored; you should pass 0.

For specific details on the parameters, see “[Image Transformation Functions: Gamma Corrections](#)” (page 222).

For each pixel, the floating-point value from the source PlanarF image is first clipped to the range 0.0 ... 1.0, and then converted to an integer in the range 0 ... 4095. The conversion calculation is equivalent to

```
if (realValue < 0.0f) realValue = 0.0f;
if (realValue > 1.0f) realValue = 1.0f;
intValue = (int)(realValue * 4095.0f + 0.5f);
```

This integer is used as an index to get an 8-bit value from the table. This value is used as the corresponding pixel in the Planar8 result image.

For more details, see “[vImageLookupTable_PlanarFToPlanar8](#)” (page 229)

For `vImageInterpolatedLookupTable_PlanarF`, the caller supplies:

- Source `vImageBuffer` pointer.
- Destination `vImage_Buffer` pointer.
- A pointer to an array of floating-point values.
- A value of type `vImagePixelCount` giving the number of values in the array
- A floating-point value giving the minimum input value
- A floating-point value giving the maximum input value
- An integer containing flags. The `kvImageDoNotTile` flag is honored.

For specific details on the parameters, see “[Image Transformation Functions: Gamma Corrections](#)” (page 222).

The input pixel is first clipped to the range `minFloat ... maxFloat`. The result is then calculated as

```
float clippedPixel = MAX(MIN(src_pixel, maxFloat), minFloat);
float fIndex = (float) (tableEntries - 1) * (clippedPixel - minFloat)
    / (maxFloat - minFloat);
float fract = fIndex - floor(fIndex);
unsigned long i = fIndex;
float result = table[i] * (1.0f - fract) + table[i + 1] * fract;
```

For more details, see “[Image Transformation Functions: Gamma Corrections](#)” (page 222).

Conversion Operations

This chapter describes the vImage functions for changing an image from one format to another, for table lookup operations, and for clipping.

What Is Format Conversion?

vImage supports four primary formats: Planar8, PlanarF, ARGB8888, and ARGBFFFF. There are conversion functions for changing an image from any of these formats into several of the others. In addition, there are functions for changing images in these formats into certain other formats, or from these other formats into one of the primary formats. These other formats are otherwise unsupported.

Some of the conversion functions work “in place.” That is, source and destination images may occupy the same memory if they are strictly aligned pixel for pixel. Other conversion functions do not work in place.

All of the conversion functions may do tiling and/or multithreading internally. If you do your own tiling and/or multithreading, use the `kvImageDoNotTile` flag to prevent this. Please see “[Tiling for Cache Utilization](#)” (page 23).

Conversion Functions

vImageBufferFill

Two functions, `vImageBufferFill_ARGB8888` and `vImageBufferFill_ARGBFFFF`, can be used to fill an ARGB8888 or ARGBFFFF image buffer with a specified color.

The caller supplies:

- An ARGB8888 or ARGBFFFF destination buffer.
- The fill color as an 8-bit or floating-point pixel value.
- Flags.

For specific details on the parameters, see “[Conversion Functions](#)” (page 139).

To fill a planar buffer with a specified value, see “[vImageOverwriteChannelsWithScalar](#)” (page 96).

vImageOverwriteChannels

Two functions, `vImageOverwriteChannels_ARGB8888` and `vImageOverwriteChannels_ARGBFFFF`, can be used to overwrite one or more planes of an image buffer with a specified planar buffer. For each plane that is overwritten, each pixel value is replaced with the corresponding pixel value from the planar buffer.

The caller supplies:

- A planar buffer in Planar8 or PlanarF format.
- A source buffer in ARGB8888 or ARGBFFFF format.
- A destination buffer in ARGB8888 or ARGBFFFF format.
- The selection mask as an 8-bit integer. The bits of the mask are used to select planes to be overwritten with values from the planar buffer; a value of 0x8 selects the alpha channel, 0x4 the red channel, 0x2 the green channel, and 0x1 the blue channel. Add these values together to select multiple channels.
- Flags.

For specific details on the parameters, see “[Conversion Functions](#)” (page 139).

These functions work in place.

vImageOverwriteChannelsWithScalar

Four functions, `vImageOverwriteChannelsWithScalar_Planar8`, `vImageOverwriteChannelsWithScalar_PlanarF`, `vImageOverwriteChannelsWithScalar_PlanarARGB8888`, and `vImageOverwriteChannelsWithScalar_PlanarARGBFFFF`, can be used to fill an image buffer with a specified color.

The caller supplies:

- A destination buffer in Planar8, PlanarF, ARGB8888, or ARGBFFFF format.
- The fill color as an 8-bit or floating-point pixel value.
- For interleaved destination buffer formats, a mask value as an 8-bit integer. The bits of the mask are used to select planes to be overwritten with values from the planar buffer; a value of 0x8 selects the alpha channel, 0x4 the red channel, 0x2 the green channel, and 0x1 the blue channel. Add these values together to select multiple channels.
- Flags.

For specific details on the parameters, see “[Conversion Functions](#)” (page 139).

These functions work in place.

vImagePermuteChannels

Two functions, `vImagePermuteChannels_ARGB8888` and `vImagePermuteChannels_ARGBFFFF`, can be used to reorder the planes of an image buffer according to a specified mapping.

The caller supplies:

- A source buffer in ARGB8888 or ARGBFFFF format.
- A destination buffer in ARGB8888 or ARGBFFFF format.
- The permutation map as an array of 8-bit integers. The values in the array must be 0, 1, 2, and 3 in some order. For example, if the values are 0, 3, 1, 2, then the destination image will have the same alpha channel as the source, but the red will be the source's blue, the green will be the source's red, and the blue will be the source's green.
- Flags.

For specific details on the parameters, see “[Conversion Functions](#)” (page 139).

These functions work in place.

vImage_Flatten

Two functions, `vImage_FlattenARGB8888toRGB888` and `vImage_FlattenARGB8888toRGB888`, transform an ARGB image to an RGB image against an opaque background of a specified color.

The caller supplies:

- An ARGB8888 or ARGBFFFF source buffer.
- An RGB888 or RGBFFF destination buffer.
- The background color as an ARGB8888 or ARGBFFFF pixel value, assumed to have an alpha channel value of 255 for 8-bit data, or 1.0 for floating-point data.
- A boolean value: true if the image is premultiplied, false otherwise.
- Flags.

For specific details on the parameters, see “[Conversion Functions](#)” (page 139).

The two buffers must have the same number of rows and the same number of pixels per row. These functions work in place.

Set the `kvImageDoNotTile` field in the flags parameter to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.)

vImageClip_PlanarF

This function, given a PlanarF image buffer, clips each pixel value to specified maximum and minimum values. It will work in place, i.e., the source and destination buffers may be the same buffer.

The caller supplies:

- A PlanarF source buffer.
- A PlanarF destination buffer.
- The floating-point minimum value.
- The floating-point maximum value.
- Flags.

For specific details on the parameters, see “[Conversion Functions](#)” (page 139).

This function works in place.

vImageConvert_Planar8ToPlanarF

This function transforms a Planar8 image to a PlanarF image, using a *minFloat* value and a *maxFloat* value to specify the range of values for the PlanarF image. Each source pixel, which has a value from 0 to 255 inclusive, is mapped linearly into the range *minFloat* to *maxFloat*, using the mapping (where *i* is the old pixel value):

```
new pixel value = i * (maxFloat - minFloat)/255.0f + minFloat
```

The caller supplies:

- A Planar8 source buffer.
- A PlanarF destination buffer.
- The *minFloat* value.
- The *maxFloat* value.
- Flags.

For specific details on the parameters, see “[Conversion Functions](#)” (page 139).

The two buffers must have the same number of rows and the same number of columns. This function does not work in place.

Set the `kvImageDoNotTile` field in the flags parameter to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.)

vImageConvert_PlanarFToPlanar8

This function transforms a PlanarF image to a Planar8 image. A *minFloat* value and *maxFloat* value determine what range of intensity values should be mapped to 0 to 255 in the destination image. Values outside the *minFloat* to *maxFloat* range are clipped. The mapping is

```
if oldPixel < minFloat
    newPixel = 0

if minfloat <= oldPixel <= maxFloat
    newPixel = (oldPixel - minFloat) * 255.0f / (maxFloat - minFloat)
```

```
if oldPixel > maxFloat
newPixel = 255
```

The caller supplies:

- A PlanarF source buffer.
- A Planar8 destination buffer.
- The *minFloat* value.
- The *maxFloat* value.
- Flags.

For specific details on the parameters, see “[Conversion Functions](#)” (page 139).

The two buffers must have the same number of rows and the same number of columns. This function works in place.

Set the `kvImageDoNotTile` field in the flags parameter to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.)

vImageConvert_Planar8toARGB8888

This function combines four planar8 images—one each for alpha, red, green, and blue—into a single ARGB8888 image.

The caller supplies:

- Source buffers for the alpha, red, green, and blue channels.
- An ARGB8888 destination buffer.
- Flags.

For specific details on the parameters, see “[Conversion Functions](#)” (page 139).

All four source buffers and the destination buffer must have the same number of rows and the same number of columns. This function does not work in place.

Set the `kvImageDoNotTile` field in the flags parameter to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.)

vImageConvert_ARGB8888toPlanar8

This function separates an ARGB8888 image into four planar8 images—one each for alpha, red, green, and blue.

The caller supplies:

- An ARGB8888 source buffer.
- Destination buffers for the alpha, red, green, and blue channels.

- Flags.

For specific details on the parameters, see “[Conversion Functions](#)” (page 139).

The source buffer and all four destination buffers must have the same number of rows and the same number of columns. This function works in place for one destination buffer; the others must be separately allocated.

Set the `kvImageDoNotTile` field in the flags parameter to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.)

[vImageConvert_PlanarFtoARGBFFF](#)

This function combines four planarF images—one each for alpha, red, green, and blue—into a single ARGBFFFF image.

The caller supplies:

- Source buffers for the alpha, red, green, and blue channels.
- An ARGBFFFF destination buffer.
- Flags.

For specific details on the parameters, see “[Conversion Functions](#)” (page 139).

All four source buffers and the destination buffer must have the same number of rows and the same number of columns. This function does not work in place.

Set the `kvImageDoNotTile` field in the flags parameter to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.)

[vImageConvert_ARGBFFFtoPlanarF](#)

This function separates an ARGBFFFF image into four planarF images—one each for alpha, red, green, and blue.

The caller supplies:

- An ARGBFFFF source buffer.
- Destination buffers for the alpha, red, green, and blue channels.
- Flags.

For specific details on the parameters, see “[Conversion Functions](#)” (page 139).

The source buffer and all four destination buffers must have the same number of rows and the same number of columns. This function works in place for one destination buffer; the others must be separately allocated.

Set the `kvImageDoNotTile` field in the flags parameter to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.)

vImageConvert_16SToF

This function takes an image in a special planar format—in which each pixel value is a 16-bit signed integer—and converts it to a PlanarF format. Each pixel value is changed to a floating-point value, then scaled and offset by values supplied by the caller. The calculation is

```
resultPixel = (float) sourcePixel * scale + offset
```

The caller supplies:

- A source buffer. This is an object of type `vImage_Buffer`, but it is not in one of the four standard vImage formats. Each pixel value is a 16-bit signed integer. The image is planar. The fields of the `vImage_Buffer` have their usual meaning.
- A destination buffer. This is an object of type `vImage_Buffer`, in PlanarF format.
- An offset value, to be added to each pixel (after scaling).
- A scale value. Each pixel value is multiplied by the scale value.
- flags.

For specific details on the parameters, see “[Conversion Functions](#)” (page 139).

This function does not work in place.

Set the `kvImageDoNotTile` field in the flags parameter to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.)

Note: `vImageConvert_16SToF` and `vImageConvert_FTo16S` are designed to be inverse transformations if you use the same offset value and the same scale value in the two functions. (The inversion is not precise due to round-off error.) This requires the two functions to use these values differently (and in a different order). In `vImageConvert_16SToF`, each pixel value is multiplied by `scale`, and then added to `offset`. In `vImageConvert_FTo16S`, each pixel value has `offset` subtracted from it, then is divided by `scale`.

vImageConvert_16UToF

This function takes an image in a special planar format—in which each pixel value is a 16-bit unsigned integer—and converts it to a PlanarF format. Each pixel value is changed to a floating-point value, then scaled and offset by values supplied by the caller. The calculation is

```
resultPixel = SATURATED_CLIP_SHRT_MIN_to_SHRT_MAX( (sourcePixel  
- offset) / scale + 0.5f)
```

The caller supplies:

- A source buffer. This is an object of type `vImage_Buffer`, but it is not in one of the four standard vImage formats. Each pixel value is a 16-bit unsigned integer. The image is planar. The fields of the `vImage_Buffer` have their usual meaning.
- A destination buffer. This is an object of type `vImage_Buffer`, in PlanarF format.
- An offset value, to be added to each pixel (after scaling).

- A scale value. Each pixel value is multiplied by the scale value.
- flags.

For specific details on the parameters, see “[Conversion Functions](#)” (page 139).

This function does not work in place.

Set the `kvImageDoNotTile` field in the flags parameter to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.)

Note: `vImageConvert_16UToF` and `vImageConvert_FTo16U` are designed to be inverse transformations if you use the same offset value and the same scale value in the two functions. (The inversion is not precise due to round-off error.) This requires the two functions to use these values differently (and in a different order). In `vImageConvert_16UToF`, each pixel value is multiplied by `scale`, and then added to `offset`. In `vImageConvert_FTo16U`, each pixel value has `offset` subtracted from it, then is divided by `scale`.

vImageConvert_16UToPlanar8

This function takes an image in a special planar format—in which each pixel value is a 16-bit unsigned integer—and converts it to a Planar8 format. Each 8-bit pixel value is calculated as

```
uint8_t result = (srcPixel * 255 + 32767) / 65535
```

The caller supplies:

- A source buffer. This is an object of type `vImage_Buffer`, but it is not in one of the four standard vImage formats. Each pixel value is a 16-bit unsigned integer. The image is planar. The fields of the `vImage_Buffer` have their usual meaning.
- A destination buffer. This is an object of type `vImage_Buffer`, in Planar8 format.
- flags.

For specific details on the parameters, see “[Conversion Functions](#)” (page 139).

This function works in place.

Set the `kvImageDoNotTile` field in the flags parameter to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.)

Note: You can use this function to convert interleaved 16-bit data to an ARGB8888 image as well; simply multiply the `vImage_Buffer.width` by 4.

vImageConvert_FTo16S

This function converts a PlanarF image into a special format in which each pixel is a 16-bit signed integer. Each pixel value is first offset and scaled by user-supplied values, and then changed to a 16-bit signed integer (rounded and clipped as necessary). The calculation is

```
resultPixel = SATURATED_CLIP_0_to_USHRT_MAX( (srcPixel - offset)
                                             / scale + 0.5f)
```

The caller supplies:

- A PlanarF source buffer.
- A destination buffer. This is an object of type `vImage_Buffer`, but it is not in one of the four standard `vImage` formats. Each pixel value is a 16-bit signed integer.
- An offset value, to be subtracted from each pixel.
- A scale value. Each pixel value is divided by the scale value (after the offset).
- flags.

For specific details on the parameters, see “[Conversion Functions](#)” (page 139).

This function works in place.

Set the `kvImageDoNotTile` field in the flags parameter to prevent `vImage` from using tiling internally. (This is appropriate if you are doing tiling yourself.)

Note: `vImageConvert_16SToF` and `vImageConvert_FTo16S` are designed to be inverse transformations if you use the same offset value and the same scale value in the two functions. (The inversion is not precise due to round-off error.) This requires the two functions to use these values differently (and in a different order). In `vImageConvert_16SToF`, each pixel value is multiplied by `scale`, and then added to `offset`. In `vImageConvert_FTo16S`, each pixel value has `offset` subtracted from it, then is divided by `scale`.

vImageConvert_FTo16U

This function converts a PlanarF image into a special format in which each pixel is a 16-bit unsigned integer. Each pixel value is first offset and scaled by user-supplied values, and then changed to a 16-bit signed integer (rounded and clipped as necessary). The calculation is

```
resultPixel = SATURATED_CLIP_0_to_USHRT_MAX( (sourcePixel - offset)
                                             / scale + 0.5f)
```

The caller supplies:

- A PlanarF source buffer.
- A destination buffer. This is an object of type `vImage_Buffer`, but it is not in one of the four standard `vImage` formats. Each pixel value is a 16-bit unsigned integer.
- An offset value, to be subtracted from each pixel.
- A scale value. Each pixel value is divided by the scale value (after the offset).
- flags.

For specific details on the parameters, see “[Conversion Functions](#)” (page 139).

This function works in place.

Set the `kvImageDoNotTile` field in the `flags` parameter to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.)

Note: `vImageConvert_16UToF` and `vImageConvert_FTo16U` are designed to be inverse transformations if you use the same offset value and the same scale value in the two functions. (The inversion is not precise due to round-off error.) This requires the two functions to use these values differently (and in a different order). In `vImageConvert_16UToF`, each pixel value is multiplied by `scale`, and then added to `offset`. In `vImageConvert_FTo16U`, each pixel value has `offset` subtracted from it, then is divided by `scale`.

vImageConvert_Planar8To16U

This function converts a Planar8 image into a special format in which each pixel is a 16-bit unsigned integer. Each 16-bit pixel value is calculated as

```
uint16_t result = (srcPixel * 65535 + 127) / 255
```

The caller supplies:

- A Planar8 source buffer.
- A destination buffer. This is an object of type `vImage_Buffer`, but it is not in one of the four standard vImage formats. Each pixel value is a 16-bit unsigned integer.
- `flags`.

For specific details on the parameters, see “[Conversion Functions](#)” (page 139).

This function does not work in place.

Set the `kvImageDoNotTile` field in the `flags` parameter to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.)

Note: You can use this function to convert an ARGB8888 image to interleaved 16-bit data as well; simply multiply the `vImage_Buffer.width` by 4.

vImageConvert_Planar16FtoPlanarF

This function converts a Planar16F image into a PlanarF image.

Note: The Planar16F pixels are 16-bit floating-point numbers, conformant to the OpenEXR standard. Denormals, NaNs, and +/- Infinity are supported. In conformance with IEEE-754, all signaling NaNs are quieted during the conversion (OpenEXR-1.2.1 does not do this.)

The caller supplies:

- A Planar16F source buffer.
- A PlanarF destination buffer. This is an object of type `vImage_Buffer`.

- flags.

For specific details on the parameters, see “[Conversion Functions](#)” (page 139).

This function does not work in place.

Set the `kvImageDoNotTile` field in the `flags` parameter to prevent `vImage` from using tiling internally. (This is appropriate if you are doing tiling yourself.)

`vImageConvert_PlanarFtoPlanar16F`

This function converts a `PlanarF` image into a `Planar16F` image.

Note: The `Planar16F` pixels are 16-bit floating-point numbers, conformant to the OpenEXR standard. Denormals, NaNs, and +/- Infinity are supported. In conformance with IEEE-754, all signaling NaNs are quieted during the conversion (`OpenEXR-1.2.1` does not do this.)

The caller supplies:

- A `PlanarF` source buffer.
- A `Planar16F` destination buffer. This is an object of type `vImage_Buffer`.
- flags.

For specific details on the parameters, see “[Conversion Functions](#)” (page 139).

This function works in place.

Set the `kvImageDoNotTile` field in the `flags` parameter to prevent `vImage` from using tiling internally. (This is appropriate if you are doing tiling yourself.)

`vImageConvert_ARGB1555toARGB8888`

This function converts an `ARGB1555` image into an `ARGB8888` image. (`ARGB1555` has a 1-bit alpha channel and 5-bit red, green, and blue channels.) To convert to 8-bit pixels, the alpha channel value is calculated as

```
alpha = sourceAlpha * 255
```

and each of the red, green, and blue values is calculated as

```
color = (sourceColor * 255 + 15) / 31.
```

The caller supplies:

- An `ARGB1555` source buffer.
- An `ARGB8888` destination buffer. This is an object of type `vImage_Buffer`.
- flags.

For specific details on the parameters, see “[Conversion Functions](#)” (page 139).

This function does not work in place.

Set the `kvImageDoNotTile` field in the `flags` parameter to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.)

vImageConvert_ARGB8888toARGB1555

This function converts an ARGB8888 image into an ARGB1555 image. (ARGB1555 has a 1-bit alpha channel and 5-bit red, green, and blue channels.) To convert from 8-bit pixels, the alpha channel value is calculated as

```
alpha = (sourceAlpha + 127) / 255
```

and each of the red, green, and blue values is calculated as

```
color = (sourceColor * 31 + 127) / 255
```

The caller supplies:

- An ARGB8888 source buffer.
- An ARGB1555 destination buffer. This is an object of type `vImage_Buffer`.
- `flags`.

For specific details on the parameters, see “[Conversion Functions](#)” (page 139).

This function works in place.

Set the `kvImageDoNotTile` field in the `flags` parameter to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.)

vImageConvert_ARGB1555toPlanar8

This function converts an ARGB1555 image into four Planar8 images. (ARGB1555 has a 1-bit alpha channel and 5-bit red, green, and blue channels.) To convert to 8-bit pixels, the alpha value is calculated as

```
alpha = sourceAlpha * 255
```

and each of the red, green, and blue values is calculated as

```
color = (sourceColor * 255 + 15) / 31
```

The caller supplies:

- An ARGB1555 source buffer.
- Four planar8 destination buffers, one for each of the four channels. These are objects of type `vImage_Buffer`.
- `flags`.

For specific details on the parameters, see “[Conversion Functions](#)” (page 139).

This function works in place for one destination channel; the others must be allocated separately.

Set the `kvImageDoNotTile` field in the `flags` parameter to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.)

vImageConvert_Planar8toARGB1555

This function converts four Planar8 images into an ARGB1555 image. (ARGB1555 has a 1-bit alpha channel and 5-bit red, green, and blue channels.) To convert from 8-bit pixels, the alpha channel value is calculated as

$$(\text{sourceValue} + 127) / 255$$

and each of the red, green, and blue values is calculated as

$$(\text{sourceValue} * 31 + 127) / 255$$

The caller supplies:

- Four planar8 source buffers, one for each of the four channels.
- An ARGB1555 destination buffer. This is an object of type `vImage_Buffer`.
- `flags`.

For specific details on the parameters, see “[Conversion Functions](#)” (page 139).

This function does not work in place.

Set the `kvImageDoNotTile` field in the `flags` parameter to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.)

vImageConvert_RGB565toARGB8888

This function converts an RGB565 image into an ARGB8888 image using a specified alpha value for all pixels. (RGB565 has a 5-bit red channel, a 6-bit green channel, and a 5-bit blue channel.) To convert to 8-bit pixels, the alpha value taken from the value passed by the caller. The red value is calculated as

$$(\text{redValue} * 255 + 15) / 31$$

The green values is calculated as

$$(\text{greenValue} * 255 + 31) / 63$$

The blue value is calculated as

$$(\text{blueValue} * 255 + 15) / 31$$

The caller supplies:

- An 8-bit alpha value.

- An RGB565 source buffer.
- An ARGB8888 destination buffer. This is an object of type `vImage_Buffer`.
- flags.

For specific details on the parameters, see “[Conversion Functions](#)” (page 139).

This function does not work in place.

Set the `kvImageDoNotTile` field in the flags parameter to prevent `vImage` from using tiling internally. (This is appropriate if you are doing tiling yourself.)

vImageConvert_ARGB8888toRGB565

This function converts an ARGB8888 image into an RGB565 image. (RGB565 has a 5-bit red channel, a 6-bit green channel, and a 5-bit blue channel.) To convert from 8-bit pixels, the red value is calculated as

```
redValue = (sourceRedValue * 31 + 127) / 255
```

The green value is calculated as

```
greenValue = (sourceGreenValue * 63 + 127) / 255
```

The blue value is calculated as

```
blueValue = (sourceBlueValue * 31 + 127) / 255
```

The source alpha value is discarded.

The RGB565 pixel is calculated as

```
pixel = (redValue << 11) | (greenValue << 5) | blueValue
```

The caller supplies:

- An ARGB8888 source buffer.
- An ARGB1555 destination buffer. This is an object of type `vImage_Buffer`.
- flags.

For specific details on the parameters, see “[Conversion Functions](#)” (page 139).

This function works in place.

Set the `kvImageDoNotTile` field in the flags parameter to prevent `vImage` from using tiling internally. (This is appropriate if you are doing tiling yourself.)

vImageConvert_RGB565toPlanar8

This function converts an RGB565 image into three Planar8 images. (RGB565 has a 5-bit red channel, a 6-bit green channel, and a 5-bit blue channel.) To convert to 8-bit pixels, the alpha value taken from the value passed by the caller. The red value is calculated as

$$(\text{redValue} * 255 + 15) / 31$$

The green value is calculated as

$$(\text{greenValue} * 255 + 31) / 63$$

The blue value is calculated as

$$(\text{blueValue} * 255 + 15) / 31$$

The caller supplies:

- An RGB565 source buffer.
- Three planar8 destination buffers, one for each of the three channels. These are objects of type vImage_Buffer.
- flags.

For specific details on the parameters, see “[Conversion Functions](#)” (page 139).

This function works in place for one destination buffer; the others must be allocated separately.

Set the `kvImageDoNotTile` field in the flags parameter to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.)

vImageConvert_Planar8toRGB565

This function converts three Planar8 images into an RGB565 image. (RGB565 has a 5-bit red channel, a 6-bit green channel, and a 5-bit blue channel.) To convert from 8-bit pixels, the red value is calculated as

$$\text{redValue} = (\text{sourceRedValue} * 31 + 127) / 255$$

The green value is calculated as

$$\text{greenValue} = (\text{sourceGreenValue} * 63 + 127) / 255$$

The blue value is calculated as

$$\text{blueValue} = (\text{sourceBlueValue} * 31 + 127) / 255$$

The RGB565 pixel is calculated as

$$\text{pixel} = (\text{redValue} \ll 11) | (\text{greenValue} \ll 5) | \text{blueValue}$$

The caller supplies:

- Three planar8 source buffers, one for each of the three channels.

- An RGB565 destination buffer. This is an object of type `vImage_Buffer`.
- `flags`.

For specific details on the parameters, see “[Conversion Functions](#)” (page 139).

This function does not work in place.

Set the `kvImageDoNotTile` field in the `flags` parameter to prevent `vImage` from using tiling internally. (This is appropriate if you are doing tiling yourself.)

`vImageConvert_RGB888toARGB8888`

This function converts an RGB888 image into an ARGB8888 image using either a Planar8 alpha channel or a single specified alpha value. Premultiplication can be specified in the call; if it is, each destination channel value is calculated as

$$(\text{alpha} * \text{sourceValue} + 127) / 255$$

The caller supplies:

- An RGB888 source buffer.
- A Planar8 buffer containing an alpha channel to be copied into the destination image. If this is `NULL`, the single alpha value specified in the next parameter is used instead.
- An 8-bit alpha value to be used for all pixels in the destination image; this is ignored if the preceding parameter is not `NULL`.
- An ARGB8888 destination buffer. This is an object of type `vImage_Buffer`.
- A Boolean to indicate whether premultiplication is desired.
- `flags`.

For specific details on the parameters, see “[Conversion Functions](#)” (page 139).

This function does not work in place.

Set the `kvImageDoNotTile` field in the `flags` parameter to prevent `vImage` from using tiling internally. (This is appropriate if you are doing tiling yourself.)

`vImageConvert_ARGB8888toRGB888`

This function converts an ARGB8888 image into an RGB888 image, discarding the alpha channel. The red, green, and blue channels are simply copied from the source to the destination.

The caller supplies:

- An ARGB888 source buffer.
- An RGB888 destination buffer. This is an object of type `vImage_Buffer`.
- `flags`.

For specific details on the parameters, see “[Conversion Functions](#)” (page 139).

This function works in place.

Set the `kvImageDoNotTile` field in the `flags` parameter to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.)

vImageConvert_Planar8toRGB888

This function converts three Planar8 images into an RGB888 image.

The caller supplies:

- Three planar8 source buffers, one for each of the three channels.
- An RGB888 destination buffer. This is an object of type `vImage_Buffer`.
- `flags`.

For specific details on the parameters, see “[Conversion Functions](#)” (page 139).

This function does not work in place.

Set the `kvImageDoNotTile` field in the `flags` parameter to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.)

vImageConvert_RGB888toPlanar8

This function converts an RGB888 image into three Planar8 images.

The caller supplies:

- An RGB888 source buffer.
- Three planar8 destination buffers, one for each of the three channels. These are objects of type `vImage_Buffer`.
- `flags`.

For specific details on the parameters, see “[Conversion Functions](#)” (page 139).

This function works in place for one destination buffer; the others must be allocated separately.

Set the `kvImageDoNotTile` field in the `flags` parameter to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.)

vImageConvert_PlanarFtoRGBFFF

This function converts three PlanarF images into an RGBFFF image.

The caller supplies:

- Three planarF source buffers, one for each of the three channels.
- An RGBFFF destination buffer. This is an object of type vImage_Buffer.
- flags.

For specific details on the parameters, see “[Conversion Functions](#)” (page 139).

This function does not work in place.

Set the `kvImageDoNotTile` field in the flags parameter to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.)

vImageConvert_RGBFFFtoPlanarF

This function converts an RGBFFF image into three PlanarF images.

The caller supplies:

- An RGBFFF source buffer.
- Three planarF destination buffers, one for each of the three channels. These are objects of type vImage_Buffer.
- flags.

For specific details on the parameters, see “[Conversion Functions](#)” (page 139).

This function works in place for one destination buffer; the others must be allocated separately.

Set the `kvImageDoNotTile` field in the flags parameter to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.)

vImageTableLookUp_Planar8

This function transforms a Planar8 image by replacing all pixels of a given intensity value with pixels of a new intensity value. The mapping of old values to new values is given by a 256-element lookup table (LUT).

The caller supplies:

- A Planar8 source buffer.
- A Planar8 destination buffer.
- The lookup table.
- Flags.

For specific details on the parameters, see “[Conversion Functions](#)” (page 139).

This function works in place.

The two buffers must have the same number of rows and the same number of columns.

Set the `kvImageDoNotTile` field in the `flags` parameter to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.)

vImageTableLookUp_ARGB8888

This function transforms an ARGB8888 image by replacing all pixels of a given color (including alpha value) with pixels of a new color.

The color mapping is done by separating each color into its alpha, red, green, and blue components (each one of which is a value from 0 to 255); using four separate lookup tables to replace each of the four components; and combining the new components into a 4-channel color.

The caller supplies:

- An ARGB8888 source buffer.
- An ARGB8888 destination buffer.
- Four lookup tables (one each for alpha, red, green, and blue). If you pass NULL for a lookup table, the corresponding channel will be copied to the destination buffer unchanged.
- Flags.

For specific details on the parameters, see “[Conversion Functions](#)” (page 139).

This function works in place.

The two buffers must have the same number of rows and the same number of columns.

Set the `kvImageDoNotTile` field in the `flags` parameter to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.)

Note that you can not use this function to do a completely arbitrary color mapping. If two different source colors have the same green component (just for example), they must be mapped to two destination colors whose green components are equal. They can not be mapped to two arbitrary colors.

General Conversions

These functions described here are convenience functions only. They are not fast or vectorized.

Four of the conversion functions—specifically, `vImageConvert_ChunkyToPlanar8`, `vImageConvert_ChunkyToPlanarF`, `vImageConvert_PlanarToChunky8`, and `vImageConvert_PlanarToChunkyF`—enable you to deal with images in a very general way, called the “general representation.”

This makes it possible to convert between various interleaved formats that vImage does not explicitly support (and that may have less than or more than four channels) and the formats that vImage does explicitly support. (*Chunky* is sometimes used as a synonym for *interleaved*.) Some non-interleaved formats can be represented as well.

The conversion functions change images described in the general representation into one of the primary vImage formats, or from some of the primary vImage formats into some format described in the general representation.

The general representation describes an image as follows: there are *channelCount* channels. There is an array of pointers of length *channelCount*, one for each channel. Each pointer points to the first pixel value of the corresponding channel.

All the channels share the following values:

- *stride*: the number of bytes from one pixel value of the channel to the next pixel value of the channel (within a row)
- *width*: the number of pixel values in a row
- *height*: the number of rows
- *rowBytes*: the number of bytes from the beginning of one row of the channel to the beginning of the next row of the channel

All the pixel values must be of the same type—either 8 bits (for `vImageConvert_ChunkyToPlanar8` and `vImageConvert_PlanarToChunky8`) or floating point (for `vImageConvert_ChunkToPlanarF` and `vImageConvert_PlanarToChunkyF`).

Convert Chunky to Planar

This function separates an image into a collection of corresponding planar images, one for each channel of the original image. There are two versions of this function, one for images with 8-bit pixel values, one for images with floating-point pixel values: `vImageConvert_ChunkToPlanar8` and `vImageConvert_ChunkToPlanarF`, respectively. The type of pixel in each channel of the original image must be the same as the type of pixel in the destination buffers. The destination buffers must be planar.

The caller supplies:

- An array of pointers, one for each channel of the source image. Each pointer points to the first pixel value of the corresponding channel.
- An array of planar destination buffers, one for each channel of the source image. These are objects of type `vImage_Buffer`. The destination buffers must all have the same width and height (in pixels), but they can have different *rowBytes* values. For each destination buffer, you must set the fields of the structure yourself, and allocate memory for its data. When you are done with the destination buffers, you must deallocate their memory.
- The number of channels.
- A stride value, which is the number of bytes from one pixel value of a given channel to the next pixel value of that channel (within a row). This value must be the same for all channels.
- The number of pixel values in a channel's row. This value must be the same for all channels.
- The number of rows in each channel. This value must be the same for all channels.
- The number of bytes from the beginning of a channel's row to the beginning of that channel's next row. This value must be the same for all channels.
- flags.

For specific details on the parameters, see “[Conversion Functions](#)” (page 139).

These functions do not work in place.

Set the `kvImageDoNotTile` field in the `flags` parameter to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.)

Convert Planar to Chunky

This function takes a collection of planar images and combines them into a single interleaved image, with one channel for each planar image. There are two versions of this function, one for images with 8-bit pixel values, one for images with floating-point pixel values: `vImageConvert_PlanarToChunky8` and `vImageConvert_PlanarToChunkyF`, respectively. The planar images must all have the same width and height (in pixels); they may have different `rowBytes` values. They must all have the same pixel type (8-bit or floating-point), which will be the pixel type for the resulting image.

The caller supplies:

- An array of planar source buffers, objects of type `vImage_Buffer`. The number of elements in this array will be the number of channels in the resulting image. The planar buffers must all have the same width and height (in pixels). They may have different `rowBytes` values.
- An array of pointers, one for each channel of the resulting image. Each pointer points to the start of the data area for the corresponding channel. You must allocate the memory for each pointer (that is, the data area for each channel) yourself. The pixel values will be filled in by the function call. When you are done with the resulting image, you must deallocate all the memory for the pointers.
- The number of channels (which is the same as the number of planar source buffers).
- A stride value, which is the number of bytes from one pixel value of a given channel to the next pixel value of that channel (within a row). This value will be used for all channels.
- The number of pixel values in a channel’s row. This value will be used for all channels. It must be the same as the width of each of the planar source buffers.
- The number of rows in each channel. This value will be used for all channels. It must be the same as the height of each of the planar source buffers.
- The number of bytes from the beginning of a channel’s row to the beginning of that channel’s next row. This value will be used for all channels. It does not have to be the same as the `rowBytes` value of the planar source buffers (which, in any case, do not have to share the same `rowBytes` value themselves).
- Flags.

For specific details on the parameters, see “[Conversion Functions](#)” (page 139).

These functions do not work in place.

Set the `kvImageDoNotTile` field in the `flags` parameter to prevent vImage from using tiling internally. (This is appropriate if you are doing tiling yourself.)

vImage Functions

Alpha Compositing Functions

For conceptual introduction to these functions, please see “[Alpha Compositing Operations](#)” (page 77).

vImageAlphaBlend_ARGBFFFF

This function performs non-premultiplied alpha compositing of a top image over a bottom image, placing the result in a destination buffer. The images must be in ARGBFFFF format.

```
vImage_Error vImageAlphaBlend_ARGBFFFF (
    const vImage_Buffer *srcTop,
    const vImage_Buffer *srcBottom,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters

srcTop

A pointer to a structure of type `vImage_Buffer` containing the top image.

srcBottom

A pointer to a structure of type `vImage_Buffer` containing the bottom image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The top image, bottom image, and destination buffer must all have the same height and width (in pixels). This function works in place.

vImage Functions

For details on the calculation, see “[Non-premultiplied Alpha Blend](#)” (page 78).

vImageAlphaBlend_ARGB8888

This function performs non-premultiplied alpha compositing of a top image over a bottom image, placing the result in a destination buffer. The images must be in ARGB8888 format.

```
vImage_Error vImageAlphaBlend_ARGB8888 (
    const vImage_Buffer *srcTop,
    const vImage_Buffer *srcBottom,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters

srcTop

A pointer to a structure of type `vImage_Buffer` containing the top image.

srcBottom

A pointer to a structure of type `vImage_Buffer` containing the bottom image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The top image, bottom image, and destination buffer should all have the same height and width (in pixels). This function works in place.

For details on the calculation, see “[Non-premultiplied Alpha Blend](#)” (page 78).

vImageAlphaBlend_PlanarF

This function performs non-premultiplied alpha compositing of a top image over a bottom image, placing the result in a destination buffer. The images and alpha values must be in PlanarF format.

```
vImage_Error vImageAlphaBlend_PlanarF (
    const vImage_Buffer *srcTop,
    const vImage_Buffer *srcTopAlpha,
    const vImage_Buffer *srcBottom,
    const vImage_Buffer *srcBottomAlpha,
    const vImage_Buffer *alpha,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

vImage Functions

Parameters*srcTop*

A pointer to a structure of type `vImage_Buffer` containing the top image.

srcTopAlpha

A pointer to a structure of type `vImage_Buffer` containing the alpha values for the top image.

srcBottom

A pointer to a structure of type `vImage_Buffer` containing the bottom image.

srcBottomAlpha

A pointer to a structure of type `vImage_Buffer` containing the alpha values for the bottom image.

alpha

A pointer to a structure of type `vImage_Buffer` containing the composite alpha values (that is, alpha values for the composite image). This input must be pre-calculated (see Discussion).

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The top image, alpha values for the top image, bottom image, alpha values for the bottom image, destination buffer, and composite alpha values (alpha values for the composite image) should all have the same height and width (in pixels). This function works in place.

alpha, the alpha values for the composite image, is not calculated by this call. It must be pre-calculated using the function `vPremultipliedAlphaBlend_PlanarF` with the following call:

```
vImagePremultipliedAlphaBlend_PlanarF( srcTopAlpha, srcTopAlpha, srcBottomAlpha,
                                         alpha, kvImageNoFlags );
```

alpha is a destination buffer in the `vImagePremultipliedAlphaBlend_PlanarF` call, which will be set the calculated composite alpha. Then it can be used as an input buffer for the `vImageAlphaBlend_PlanarF` call. Note that `srcTopAlpha` is used as an input twice in the `vImagePremultipliedAlphaBlend_PlanarF` call here.

`vImageAlphaBlend_PlanarF` could have been designed to calculate *alpha*. However, when this function is called, it is often called three times, one for each color channel (Red, Green, Blue), with the same values for *alpha* each time. It was deemed more efficient to have the caller precalculate *alpha*, rather than have the calculation repeated three times by the function.

For details on the calculation, see “[Non-premultiplied Alpha Blend](#)” (page 78).

vImageAlphaBlend_Planar8

This function performs non-premultiplied alpha compositing of a top image over a bottom image, placing the result in a destination buffer. The images and alpha values must be in Planar8 format.

```
vImage_Error vImageAlphaBlend_Planar8 (
    const vImage_Buffer *srcTop,
    const vImage_Buffer *srcTopAlpha,
    const vImage_Buffer *srcBottom,
    const vImage_Buffer *srcBottomAlpha,
    const vImage_Buffer *alpha,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters*srcTop*

A pointer to a structure of type `vImage_Buffer` containing the top image.

srcTopAlpha

A pointer to a structure of type `vImage_Buffer` containing the alpha values for the top image.

srcBottom

A pointer to a structure of type `vImage_Buffer` containing the bottom image.

srcBottomAlpha

A pointer to a structure of type `vImage_Buffer` containing the alpha values for the bottom image.

alpha

A pointer to a structure of type `vImage_Buffer` containing the composite alpha values (that is, alpha values for the composite image). This input must be pre-calculated (see Discussion).

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The top image, alpha values for the top image, bottom image, alpha values for the bottom image, destination buffer, and composite alpha values (alpha values for the composite image) should all have the same height and width (in pixels). This function works in place.

alpha, the alpha values for the composite image, is not calculated by this call. It must be pre-calculated using the function `vPremultipliedAlphaBlend_Planar8` with the following call:

```
vImagePremultipliedAlphaBlend_Planar8( srcTopAlpha, srcTopAlpha, srcBottomAlpha,
                                         alpha, kvImageNoFlags );
```

vImage Functions

alpha is a destination buffer in the vImagePremultipliedAlphaBlend_Planar8 call, which will be set the calculated composite alpha. Then it can be used as an input buffer for the vImageAlphaBlend_Planar8 call. Note that *srcTopAlpha* is used as an input twice in the vImagePremultipliedAlphaBlend_Planar8 call here.

vImageAlphaBlend_PlanarF could have been designed to calculate alpha. However, when this function is called, it is often called three times, one for each color channel (Red, Green, Blue), with the same values for alpha each time. It was deemed more efficient to have the caller precalculate alpha, rather than have the calculation repeated three times by the function.

For details on the calculation, see “[Non-premultiplied Alpha Blend](#)” (page 78).

vImagePremultipliedAlphaBlend_ARGBFFFF

This function performs premultiplied alpha compositing of a top image over a bottom image, placing the result in a destination buffer. The images must be in ARGBFFFF format.

```
vImage_Error vImagePremultipliedAlphaBlend_ARGBFFFF (
    const vImage_Buffer *srcTop,
    const vImage_Buffer *srcBottom,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters

srcTop

A pointer to a structure of type vImage_Buffer containing the top image.

srcBottom

A pointer to a structure of type vImage_Buffer containing the bottom image.

dest

A pointer to a structure of type vImage_Buffer. On return, it contains the destination image.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The top image, bottom image, and destination buffer must all have the same height and width (in pixels). This function works in place.

Since the original alpha values are maintained in the interleaved data, you do not need to provide them separately.

For details on the calculation, see “[Premultiplied Alpha Blend](#)” (page 80).

vImagePremultipliedAlphaBlend_ARGB8888

This function performs premultiplied alpha compositing of a top image over a bottom image, placing the result in a destination buffer. The images must be in ARGB8888 format.

```
vImage_Error vImagePremultipliedAlphaBlend_ARGB8888 (
    const vImage_Buffer *srcTop,
    const vImage_Buffer *srcBottom,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters*srcTop*

A pointer to a structure of type `vImage_Buffer` containing the top image.

srcBottom

A pointer to a structure of type `vImage_Buffer` containing the bottom image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The top image, bottom image, and destination buffer must all have the same height and width (in pixels). This function works in place.

Since the original alpha values are maintained in the interleaved data, you do not need to provide them separately.

For details on the calculation, see “[Premultiplied Alpha Blend](#)” (page 80).

vImagePremultipliedAlphaBlend_PlanarF

This function performs premultiplied alpha compositing of a top image over a bottom image, placing the result in a destination buffer. The images must be in PlanarF format.

```
vImage_Error vImagePremultipliedAlphaBlend_Planar8 (
    const vImage_Buffer *srcTop,
    const vImage_Buffer *srcTopAlpha,
    const vImage_Buffer *srcBottom,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

vImage Functions**Parameters***srcTop*

A pointer to a structure of type `vImage_Buffer` containing the top image.

srcTopAlpha

A pointer to a structure of type `vImage_Buffer` containing the alpha information for the top image.

srcBottom

A pointer to a structure of type `vImage_Buffer` containing the bottom image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The top image, top image alpha information, bottom image, and destination buffer must all have the same height and width (in pixels). This function works in place.

Even though the alpha values have been premultiplied into the pixel values, the function also requires the original alpha information for the top image to do its calculations. There is no way to extract this information from the premultiplied planar values, so you must provide it.

For details on the calculation, see “[Premultiplied Alpha Blend](#)” (page 80).

`vImagePremultipliedAlphaBlend_Planar8`

This function performs premultiplied alpha compositing of a top image over a bottom image, placing the result in a destination buffer. The images must be in Planar8 format.

```

vImage_Error vImagePremultipliedAlphaBlend_Planar8 (
    const vImage_Buffer *srcTop,
    const vImage_Buffer *srcTopAlpha,
    const vImage_Buffer *srcBottom,
    const vImage_Buffer *dest,
    vImage_Flags flags
);

```

Parameters*srcTop*

A pointer to a structure of type `vImage_Buffer` containing the top image.

srcTopAlpha

A pointer to a structure of type `vImage_Buffer` containing the alpha information for the top image.

vImage Functions*srcBottom*

A pointer to a structure of type `vImage_Buffer` containing the bottom image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The top image, top image alpha information, bottom image, and destination buffer must all have the same height and width (in pixels). This function works in place.

Even though the alpha values have been premultiplied into the pixel values, the function also requires the original alpha information for the top image to do its calculations. There is no way to extract this information from the premultiplied planar values, so you must provide it.

For details on the calculation, see “[Premultiplied Alpha Blend](#)” (page 80).

`vImagePremultipliedConstAlphaBlend_ARGBFFFF`

This function performs premultiplied alpha compositing of a top image over a bottom image, using a single alpha value for the whole image and placing the result in a destination buffer. The images must be in ARGBFFFF format.

```
vImage_Error vImagePremultipliedConstAlphaBlend_ARGBFFFF (
    const vImage_Buffer *srcTop,
    Pixel_F constAlpha,
    const vImage_Buffer *srcBottom,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters*srcTop*

A pointer to a structure of type `vImage_Buffer` containing the top image.

constAlpha

A floating-point alpha value of type `Pixel_F`.

srcBottom

A pointer to a structure of type `vImage_Buffer` containing the bottom image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image.

vImage Functions*flags*

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The top image, bottom image, and destination buffer must all have the same height and width (in pixels). This function works in place.

For details on the calculation, see “[Premultiplied Constant Alpha Blend](#)” (page 82).

`vImagePremultipliedConstAlphaBlend_ARGB8888`

This function performs premultiplied alpha compositing of a top image over a bottom image, using a single alpha value for the whole image and placing the result in a destination buffer. The images must be in ARGB8888 format.

```
vImage_Error vImagePremultipliedConstAlphaBlend_ARGB8888 (
    const vImage_Buffer *srcTop,
    Pixel_8 constAlpha,
    const vImage_Buffer *srcBottom,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters*srcTop*

A pointer to a structure of type `vImage_Buffer` containing the top image.

constAlpha

An 8-bit integer alpha value of type `Pixel_8`.

srcBottom

A pointer to a structure of type `vImage_Buffer` containing the bottom image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The top image, bottom image, and destination buffer must all have the same height and width (in pixels). This function works in place.

For details on the calculation, see “[Premultiplied Constant Alpha Blend](#)” (page 82).

vImagePremultipliedConstAlphaBlend_PlanarF

This function performs premultiplied alpha compositing of a top image over a bottom image, using a single alpha value for the whole image and placing the result in a destination buffer. The images must be in PlanarF format.

```
vImage_Error vImagePremultipliedConstAlphaBlend_Planar8 (
    const vImage_Buffer *srcTop,
    Pixel_F constAlpha,
    const vImage_Buffer *srcTopAlpha,
    const vImage_Buffer *srcBottom,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters

srcTop

A pointer to a structure of type `vImage_Buffer` containing the top image.

constAlpha

A floating-point alpha value of type `Pixel_F`.

srcTopAlpha

A pointer to a structure of type `vImage_Buffer` containing the alpha information for the top image.

srcBottom

A pointer to a structure of type `vImage_Buffer` containing the bottom image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The top image, top image alpha information, bottom image, and destination buffer must all have the same height and width (in pixels). This function works in place.

Even though the alpha values have been premultiplied into the pixel values, the function also requires the original alpha information for the top image to do its calculations. There is no way to extract this information from the premultiplied planar values, so you must provide it.

For details on the calculation, see “[Premultiplied Constant Alpha Blend](#)” (page 82).

vImagePremultipliedConstAlphaBlend_Planar8

This function performs premultiplied alpha compositing of a top image over a bottom image, using a single alpha value for the whole image and placing the result in a destination buffer. The images must be in Planar8 format.

```
vImage_Error vImagePremultipliedConstAlphaBlend_Planar8 (
    const vImage_Buffer *srcTop,
    Pixel_8 constAlpha,
    const vImage_Buffer *srcTopAlpha,
    const vImage_Buffer *srcBottom,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters

srcTop

A pointer to a structure of type `vImage_Buffer` containing the top image.

constAlpha

An 8-bit integer alpha value of type `Pixel_8`.

srcTopAlpha

A pointer to a structure of type `vImage_Buffer` containing the alpha information for the top image.

srcBottom

A pointer to a structure of type `vImage_Buffer` containing the bottom image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The top image, top image alpha information, bottom image, and destination buffer must all have the same height and width (in pixels). This function works in place.

Even though the alpha values have been premultiplied into the pixel values, the function also requires the original alpha information for the top image to do its calculations. There is no way to extract this information from the premultiplied planar values, so you must provide it.

For details on the calculation, see “[Premultiplied Constant Alpha Blend](#)” (page 82).

vImageAlphaBlend_NonpremultipliedToPremultiplied_ARGBFFFF

This function performs mixed alpha compositing of a non-premultiplied top image over a premultiplied bottom image, placing the premultiplied result in a destination buffer. The images must be in ARGBFFFF format.

```
vImage_Error vImageAlphaBlend_NonpremultipliedToPremultiplied_ARGBFFFF (
    const vImage_Buffer *srcTop,
    const vImage_Buffer *srcBottom,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters*srcTop*

A pointer to a structure of type `vImage_Buffer` containing the non-premultiplied top image.

srcBottom

A pointer to a structure of type `vImage_Buffer` containing the premultiplied bottom image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the premultiplied destination image.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

This function will work in place if the source and destination images overlap exactly. The top and bottom source buffers must be at least as wide and at least as high (in pixels) as the destination buffer.

For details on the calculation, see “[Non-premultiplied to Premultiplied Alpha Blend](#)” (page 81).

vImageAlphaBlend_NonpremultipliedToPremultiplied_ARGB8888

This function performs mixed alpha compositing of a non-premultiplied top image over a premultiplied bottom image, placing the premultiplied result in a destination buffer. The images must be in ARGB8888 format.

```
vImage_Error vImageAlphaBlend_NonpremultipliedToPremultiplied_ARGB8888 (
    const vImage_Buffer *srcTop,
    const vImage_Buffer *srcBottom,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

vImage Functions**Parameters***srcTop*

A pointer to a structure of type `vImage_Buffer` containing the non-premultiplied top image.

srcBottom

A pointer to a structure of type `vImage_Buffer` containing the premultiplied bottom image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the premultiplied destination image.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

This function will work in place if the source and destination images overlap exactly. The top and bottom source buffers must be at least as wide and at least as high (in pixels) as the destination buffer.

For details on the calculation, see “[Non-premultiplied to Premultiplied Alpha Blend](#)” (page 81).

`vImageAlphaBlend_NonpremultipliedToPremultiplied_PlanarF`

This function performs mixed alpha compositing of a non-premultiplied top image over a premultiplied bottom image, placing the premultiplied result in a destination buffer. The images and alpha values must be in PlanarF format.

```
vImage_Error vImageAlphaBlend_NonpremultipliedToPremultiplied_PlanarF (
    const vImage_Buffer *srcTop,
    const vImage_Buffer *srcTopAlpha,
    const vImage_Buffer *srcBottom,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters*srcTop*

A pointer to a structure of type `vImage_Buffer` containing the non-premultiplied top image.

srcTopAlpha

A pointer to a structure of type `vImage_Buffer` containing the alpha values for the top image.

srcBottom

A pointer to a structure of type `vImage_Buffer` containing the premultiplied bottom image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the premultiplied destination image.

vImage Functions*flags*

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

This function will work in place if the source and destination images overlap exactly. The top and bottom source buffers must be at least as wide and at least as high (in pixels) as the destination buffer.

For details on the calculation, see “[Non-premultiplied to Premultiplied Alpha Blend](#)” (page 81).

vImageAlphaBlend_NonpremultipliedToPremultiplied_Planar8

This function performs mixed alpha compositing of a non-premultiplied top image over a premultiplied bottom image, placing the premultiplied result in a destination buffer. The images and alpha values must be in Planar8 format.

```
vImage_Error vImageAlphaBlend_NonpremultipliedToPremultiplied_Planar8 (
    const vImage_Buffer *srcTop,
    const vImage_Buffer *srcTopAlpha,
    const vImage_Buffer *srcBottom,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters*srcTop*

A pointer to a structure of type `vImage_Buffer` containing the non-premultiplied top image.

srcTopAlpha

A pointer to a structure of type `vImage_Buffer` containing the alpha values for the top image.

srcBottom

A pointer to a structure of type `vImage_Buffer` containing the premultiplied bottom image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the premultiplied destination image.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

vImage Functions**Discussion**

This function will work in place if the source and destination images overlap exactly. The top and bottom source buffers must be at least as wide and at least as high (in pixels) as the destination buffer.

For details on the calculation, see “[Non-premultiplied to Premultiplied Alpha Blend](#)” (page 81).

vImagePremultiplyData_ARGBFFFF

This function takes an image in non-premultiplied alpha format and transforms it into an image in premultiplied alpha format. The image must be in ARGBFFFF format.

```
vImage_Error vImagePremultiplyData_ARGBFFFF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters

src

The source image, in non-premultiplied alpha format.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image, in premultiplied alpha format.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The function gets the required alpha information from the alpha channel of the original image. The alpha channel is copied over unchanged to the destination image. This function works in place.

For details on the calculation, see “[Premultiply Data](#)” (page 84).

vImagePremultiplyData_RGBAFFF

This function takes an image in non-premultiplied alpha format and transforms it into an image in premultiplied alpha format. The image must be in RGBAFFF format.

```
vImage_Error vImagePremultiplyData_RGBAFFF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

vImage Functions**Parameters***src*

The source image, in non-premultiplied alpha format.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image, in premultiplied alpha format.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The function gets the required alpha information from the alpha channel of the original image. The alpha channel is copied over unchanged to the destination image. This function works in place.

For details on the calculation, see “[Premultiply Data](#)” (page 84).

vImagePremultiplyData_ARGB8888

This function takes an image in non-premultiplied alpha format and transforms it into an image in premultiplied alpha format. The image must be in ARGB8888 format.

```
vImage_Error vImagePremultiplyData_ARGB8888 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters*src*

The source image, in non-premultiplied alpha format.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image, in premultiplied alpha format.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The function gets the required alpha information from the alpha channel of the original image. The alpha channel is copied over unchanged to the destination image. This function works in place.

For details on the calculation, see “[Premultiply Data](#)” (page 84).

vImagePremultiplyData_RGBA8888

This function takes an image in non-premultiplied alpha format and transforms it into an image in premultiplied alpha format. The image must be in RGBA8888 format.

```
vImage_Error vImagePremultiplyData_RGBA8888 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters

src

The source image, in non-premultiplied alpha format.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image, in premultiplied alpha format.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The function gets the required alpha information from the alpha channel of the original image. The alpha channel is copied over unchanged to the destination image. This function works in place.

For details on the calculation, see “[Premultiply Data](#)” (page 84).

vImagePremultiplyData_PlanarF

This function takes an image in non-premultiplied alpha format, along with alpha information, and transforms it into an image in premultiplied alpha format. The image must be in PlanarF format.

```
vImage_Error vImagePremultiplyData_PlanarF (
    const vImage_Buffer *src,
    const vImage_Buffer *alpha,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

vImage Functions**Parameters***src*

The source image, in non-premultiplied alpha format.

alpha

The alpha information to be used with the image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image, in premultiplied alpha format.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

Since the planar source image does not contain its own alpha information, the alpha information must be supplied explicitly. This function works in place.

For details on the calculation, see “[Premultiply Data](#)” (page 84).

`vImagePremultiplyData_Planar8`

This function takes an image in non-premultiplied alpha format, along with alpha information, and transforms it into an image in premultiplied alpha format. The image must be in Planar8 format.

```
vImage_Error vImagePremultiplyData_Planar8 (
    const vImage_Buffer *src,
    const vImage_Buffer *alpha,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters*src*

The source image, in non-premultiplied alpha format.

alpha

The alpha information to be used with the image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image, in premultiplied alpha format.

vImage Functions*flags*

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

Since the planar source image does not contain its own alpha information, the alpha information must be supplied explicitly. This function works in place.

For details on the calculation, see “[Premultiply Data](#)” (page 84).

`vImageUnpremultiplyData_ARGBFFFF`

This function takes an image in premultiplied alpha format and transforms it into an image in non-premultiplied alpha format. The image must be in ARGBFFFF format.

```
vImage_Error vImageUnpremultiplyData_ARGBFFFF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters*src*

The source image, in premultiplied alpha format.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image, in non-premultiplied alpha format.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The function gets the required alpha information from the alpha channel of the original image. The alpha channel is copied over unchanged to the destination image. This function works in place.

For details on the calculation, see “[Unpremultiply Data](#)” (page 84).

vImageUnpremultiplyData_RGBAFFF

This function takes an image in premultiplied alpha format and transforms it into an image in non-premultiplied alpha format. The image must be in RGBAFFF format.

```
vImage_Error vImageUnpremultiplyData_RGBAFFF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters*src*

The source image, in premultiplied alpha format.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image, in non-premultiplied alpha format.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The function gets the required alpha information from the alpha channel of the original image. The alpha channel is copied over unchanged to the destination image. This function works in place.

For details on the calculation, see “[Unpremultiply Data](#)” (page 84).

vImageUnpremultiplyData_ARGB8888

This function takes an image in premultiplied alpha format and transforms it into an image in non-premultiplied alpha format. The image must be in ARGB8888 format.

```
vImage_Error vImageUnpremultiplyData_ARGB8888 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters*src*

The source image, in premultiplied alpha format.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image, in non-premultiplied alpha format.

vImage Functions*flags*

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The function gets the required alpha information from the alpha channel of the original image. The alpha channel is copied over unchanged to the destination image. This function works in place.

For details on the calculation, see “[Unpremultiply Data](#)” (page 84).

vImageUnpremultiplyData_RGBA8888

This function takes an image in premultiplied alpha format and transforms it into an image in non-premultiplied alpha format. The image must be in RGBA8888 format.

```
vImage_Error vImageUnpremultiplyData_RGBA8888 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters

src

The source image, in premultiplied alpha format.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image, in non-premultiplied alpha format.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The function gets the required alpha information from the alpha channel of the original image. The alpha channel is copied over unchanged to the destination image. This function works in place.

For details on the calculation, see “[Unpremultiply Data](#)” (page 84).

vImageUnpremultiplyData_PlanarF

This function takes an image in premultiplied alpha format, along with alpha information, and transforms it into an image in non-premultiplied alpha format. The image must be in PlanarF format.

```
vImage_Error vImageUnpremultiplyData_PlanarF (
    const vImage_Buffer *src,
    const vImage_Buffer *alpha,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters*src*

The source image, in premultiplied alpha format.

alpha

The alpha information to be used with the image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image, in non-premultiplied alpha format.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

Since the planar source image does not contain its own alpha information, the alpha information must be supplied explicitly. This function works in place.

For details on the calculation, see “[Unpremultiply Data](#)” (page 84).

vImageUnpremultiplyData_Planar8

This function takes an image in premultiplied alpha format, along with alpha information, and transforms it into an image in non-premultiplied alpha format. The image must be in Planar8 format.

```
vImage_Error vImageUnpremultiplyData_Planar8 (
    const vImage_Buffer *src,
    const vImage_Buffer *alpha,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters*src*

The source image, in premultiplied alpha format.

vImage Functions*alpha*

The alpha information to be used with the image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image, in non-premultiplied alpha format.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

Since the planar source image does not contain its own alpha information, the alpha information must be supplied explicitly. This function works in place.

For details on the calculation, see “[Unpremultiply Data](#)” (page 84).

Conversion Functions

For conceptual introduction to these functions, please see “[Conversion Operations](#)” (page 95).

vImageBufferFill_ARGB8888

This function fills an ARGB8888 buffer with a specified color. (Note: to fill a Planar8 buffer with a specified value, use `vImageOverwriteChannelsWithScalar_Planar8` (page 174).)

```
vImage_Error vImageBufferFill_ARGB8888 (
    const vImage_Buffer* dest,
    Pixel_8888 color,
    vImage_Flags flags
);
```

Parameters*dest*

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image.

color

An 8-bit interleaved pixel value.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

vImage Functions**vImageBufferFill_ARGBFFFF**

This function fills an ARGBFFFF buffer with a specified color. (Note: to fill a PlanarF buffer with a specified value, use [vImageOverwriteChannelsWithScalar](#) (page 96).)

```
vImage_Error vImageBufferFill_ARGBFFFF (
    const vImage_Buffer* dest,
    Pixel_FFFF color,
    vImage_Flags flags
);
```

Parameters*dest*

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image.

color

A floating-point interleaved pixel value.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see [“Tiling for Cache Utilization”](#) (page 23).

vImageClip_PlanarF

This function clips the pixel values of an image, using a caller-supplied minimum and maximum value, and puts the resulting image in a destination buffer. The image must be in PlanarF format.

```
vImage_Error vImageClip_PlanarF (
    const vImage_Buffer* src,
    const vImage_Buffer* dest,
    Pixel_F maxFloat,
    Pixel_F minFloat,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image.

maxFloat

A maximum pixel value. Any pixel value greater than this will be clipped to this value in the destination image.

minFloat

A minimum pixel value. Any pixel value less than this will be clipped to this value in the destination image.

vImage Functions*flags*

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Discussion

This function works in place.

`vImageConvert_16SToF`

This function takes an image in a special planar format—in which each pixel value is a 16-bit signed integer—and converts it to a PlanarF format. Each pixel value is changed to a floating-point value, then scaled and offset by values supplied by the caller.

```
vImage_Error vImageConvert_16SToF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    float offset,
    float scale,
    vImage_Flags flags
);
```

Parameters*src*

A source buffer. This is an object of type `vImage_Buffer`, but it is not in one of the four standard `vImage` formats. Each pixel value is a 16-bit signed integer. The image is planar. The fields of the `vImage_Buffer` have their usual meaning.

dest

A destination buffer. This is an object of type `vImage_Buffer`, in PlanarF format.

offset

An offset value, to be added to each pixel (after scaling).

scale

A scale value. Each pixel value is multiplied by the scale value.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

`vImageConvert_16SToF` and `vImageConvert_FTo16S` are designed to be inverse transformations if you use the same offset value and the same scale value in the two functions. (The inversion is not precise due to round-off error.) This requires the two functions to use these values differently (and

vImage Functions

in a different order). In `vImageConvert_16SToF`, each pixel value is multiplied by `scale`, and then added to `offset`. In `vImageConvert_FTo16S`, each pixel value has `offset` subtracted from it, then is divided by `scale`.

This function does not work in place.

`vImageConvert_16UToF`

This function takes an image in a special planar format—in which each pixel value is a 16-bit unsigned integer—and converts it to a PlanarF format. Each pixel value is changed to a floating-point value, then scaled and offset by values supplied by the caller.

```
vImage_Error vImageConvert_16UToF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    float offset,
    float scale,
    vImage_Flags flags
);
```

Parameters

src

A source buffer. This is an object of type `vImage_Buffer`, but it is not in one of the four standard `vImage` formats. Each pixel value is a 16-bit unsigned integer. The image is planar. The fields of the `vImage_Buffer` have their usual meaning.

dest

A destination buffer. This is an object of type `vImage_Buffer`, in PlanarF format.

offset

An offset value, to be added to each pixel (after scaling).

scale

A scale value. Each pixel value is multiplied by the scale value.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

`vImageConvert_16UToF` and `vImageConvert_FTo16U` are designed to be inverse transformations if you use the same offset value and the same scale value in the two functions. (The inversion is not precise due to round-off error.) This requires the two functions to use these values differently (and in a different order). In `vImageConvert_16UToF`, each pixel value is multiplied by `scale`, and then added to `offset`. In `vImageConvert_FTo16U`, each pixel value has `offset` subtracted from it, then is divided by `scale`.

This function does not work in place.

vImage Functions**vImageConvert_16UtoPlanar8**

This function converts from 16U format to Planar8. 16U is a planar format in which the pixels are 16-bit unsigned integer values.

```
vImage_Error vImageConvert_16UtoPlanar8 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing a 16U image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains a Planar8 image.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The conversion from 16-bit to 8-bit values is

```
uint8_t result = (srcPixel * 255 + 32767) / 65535
```

Note that this function can also be used to convert a 4-channel interleaved 16U image to ARGB8888. Simply multiply the width of the destination buffer by four.

This function works in place.

vImageConvert_ARGB1555toARGB8888

This function takes an ARGB1555 image and converts it to an ARGB8888 image. (The ARGB1555 format has 16-bit pixels with 1 bit for alpha and 5 bits each for red, green, and blue.)

```
vImage_Error vImageConvert_ARGB1555toPlanar8 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing an ARGB1555 image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains an ARGB8888 image.

vImage Functions*flags*

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The 8-bit pixels in the destination image are calculated as

```
Pixel8 alpha = 1bitAlphaChannel * 255
Pixel8 red   = (5bitRedChannel * 255 + 15) / 31
Pixel8 green = (5bitGreenChannel * 255 + 15) / 31
Pixel8 blue  = (5bitBlueChannel * 255 + 15) / 31
```

This function does not work in place.

vImageConvert_ARGB1555toPlanar8

This function takes an ARGB1555 image and separates it into four Planar8 images, one each for each of the four channels alpha, red, green, and blue. (The ARGB1555 format has 16-bit pixels with 1 bit for alpha and 5 bits each for red, green, and blue.)

```
vImage_Error vImageConvert_ARGB1555toPlanar8 (
    const vImage_Buffer *src,
    const vImage_Buffer *destA,
    const vImage_Buffer *destR,
    const vImage_Buffer *destG,
    const vImage_Buffer *destB,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing an ARGB1555 image.

destA

A pointer to a structure of type `vImage_Buffer`. On return, it contains a Planar8 image equivalent to the alpha channel of the source image.

destR

A pointer to a structure of type `vImage_Buffer`. On return, it contains a Planar8 image equivalent to the red channel of the source image.

destG

A pointer to a structure of type `vImage_Buffer`. On return, it contains a Planar8 image equivalent to the green channel of the source image.

destB

A pointer to a structure of type `vImage_Buffer`. On return, it contains a Planar8 image equivalent to the blue channel of the source image.

vImage Functions*flags*

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The 8-bit pixels in the destination image are calculated as

```
Pixel8 alpha = 1bitAlphaChannel * 255
Pixel8 red   = (5bitRedChannel * 255 + 15) / 31
Pixel8 green = (5bitGreenChannel * 255 + 15) / 31
Pixel8 blue  = (5bitBlueChannel * 255 + 15) / 31
```

This function works in place for one destination buffer; the others must be allocated separately.

`vImageConvert_ARGB8888toARGB1555`

This function takes an ARGB8888 image and converts it into an ARGB1555 image. (The ARGB1555 format has 16-bit pixels with 1 bit for alpha and 5 bits each for red, green, and blue.)

```
vImage_Error vImageConvert_ARGB8888toARGB1555 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing an ARGB8888 image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image in ARGB1555 format.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The pixels in the destination image are calculated as

```
uint32_t alpha = (8bitAlphaChannel      + 127) / 255
uint32_t red   = (8bitRedChannel       * 31 + 127) / 255
uint32_t green = (8bitGreenChannel     * 31 + 127) / 255
```

vImage Functions

```

uint32_t blue  = (8bitBlueChannel * 31 + 127) / 255
uint16_t ARGB1555pixel = (alpha << 15) | (red << 10) | (green << 5) | blue

```

This function works in place.

vImageConvert_ARGB8888toPlanar8

This function takes an ARGB8888 image and separates it into four Planar8 images, one each for each of the four channels alpha, red, green, and blue.

```

vImage_Error vImageConvert_ARGB8888toPlanar8 (
    const vImage_Buffer *srcARGB,
    const vImage_Buffer *destA,
    const vImage_Buffer *destR,
    const vImage_Buffer *destG,
    const vImage_Buffer *destB,
    vImage_Flags flags
);

```

Parameters

srcARGB

A pointer to a structure of type `vImage_Buffer` containing an ARGB8888 image.

destA

A pointer to a structure of type `vImage_Buffer`. On return, it contains a Planar8 image equivalent to the alpha channel of the source image.

destR

A pointer to a structure of type `vImage_Buffer`. On return, it contains a Planar8 image equivalent to the red channel of the source image.

destG

A pointer to a structure of type `vImage_Buffer`. On return, it contains a Planar8 image equivalent to the green channel of the source image.

destB

A pointer to a structure of type `vImage_Buffer`. On return, it contains a Planar8 image equivalent to the blue channel of the source image.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The source image, and the *destA*, *destR*, *destG*, and *destB* destination buffers, must have the same height and the same width (in pixels). However, the source buffer contains an ARGB8888 image, while the destination buffers each contain a Planar8 image.

This function works in place for one destination buffer. The others must be allocated separately.

vImage Functions**vImageConvert_ARGB8888toRGB565**

This function takes an ARGB8888 image and converts it into an RGB565 image. The alpha channel in the ARGB8888 image is ignored. (The RGB565 format has 16-bit pixels with 5 bits for red, 6 for green, and 5 for blue.)

```
vImage_Error vImageConvert_ARGB8888toRGB565 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing an ARGB8888 image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains an RGB565.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The pixels in the destination image are calculated as

```
uint32_t red    = (8bitRedChannel    * (31*2) + 255) / (255*2)
uint32_t green = (8bitGreenChannel * 63 + 127) / 255
uint32_t blue  = (8bitBlueChannel  * 31 + 127) / 255
uint16_t RGB565pixel = (red << 11) | (green << 5) | blue
```

This function works in place.

vImageConvert_ARGB8888toRGB888

This function takes an ARGB8888 image and converts it into an RGB888 image by removing the alpha channel. The red, green, and blue channels are simply copied.

```
vImage_Error vImageConvert_ARGB8888toRGB888 (
    const vImage_Buffer *argbSrc,
    const vImage_Buffer *rgbDest,
    vImage_Flags flags
);
```

Parameters*argbSrc*

A pointer to a structure of type `vImage_Buffer` containing an ARGB8888 image.

rgbDest

A pointer to a structure of type `vImage_Buffer`. On return, it contains an RGB888 image.

vImage Functions*flags*

The following flag is used:

kvImageDoNotTile

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

This function works in place.

vImageConvert_ARGBFFFFtoPlanarF

This function takes an ARGBFFFF image and separates it into four PlanarF images, one each for each of the four channels alpha, red, green, and blue.

```
vImage_Error vImageConvert_ARGBFFFFtoPlanarF (
    const vImage_Buffer *srcARGB,
    const vImage_Buffer *destA,
    const vImage_Buffer *destR,
    const vImage_Buffer *destG,
    const vImage_Buffer *destB,
    vImage_Flags flags
);
```

Parameters*srcARGB*

A pointer to a structure of type `vImage_Buffer` containing an ARGBFFFF image.

destA

A pointer to a structure of type `vImage_Buffer`. On return, it contains a PlanarF image equivalent to the alpha channel of the source image.

destR

A pointer to a structure of type `vImage_Buffer`. On return, it contains a PlanarF image equivalent to the red channel of the source image.

destG

A pointer to a structure of type `vImage_Buffer`. On return, it contains a PlanarF image equivalent to the green channel of the source image.

destB

A pointer to a structure of type `vImage_Buffer`. On return, it contains a PlanarF image equivalent to the blue channel of the source image.

flags

The following flag is used:

kvImageDoNotTile

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

vImage Functions**Return value**

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The source image, and the `destA`, `destR`, `destG`, and `destB` destination buffers, must have the same height and the same width (in pixels). However, the source buffer contains an ARGBFFFF image, while the destination buffers each contain a PlanarF image.

This function works in place for one destination buffer. The others must be allocated separately.

vImageConvert_ChunkyToPlanar8

This function separates a source image into a collection of corresponding planar destination images, one for each channel of the original image. The pixel values for each channel of the source image and for each of the destination images must be 8-bit values.

```
vImage_Error vImageConvert_ChunkyToPlanar8 (
    const void *srcChannels[],
    const vImage_Buffer *destPlanarBuffers[],
    unsigned int channelCount,
    size_t srcStrideBytes,
    unsigned int srcWidth,
    unsigned int srcHeight,
    unsigned int srcRowBytes,
    vImage_Flags flags
);
```

Parameters*srcChannels*

An array of pointers. The array is of length `channelCount`. Each pointer points to the first pixel value of a channel in the source image. There is one pointer for each channel.

destPlanarBuffers

An array of objects of type `vImage_Buffer`. The array is of length `channelCount`. The buffers are in Planar8 format. On return, each buffer contains a planar image equivalent to the corresponding channel of the source image. Each of the buffers must have the same height (in pixels) and the same width (in pixels), but they may have different `rowBytes` values.

channelCount

The number of channels in the source image.

srcStrideBytes

The number of bytes from one pixel value in a given channel to the next pixel of that channel (within a row). This value must be the same for all channels.

srcWidth

The number of pixels in a row. This value must be the same for all channels in the source image, and for all the destination buffers.

srcHeight

The number of rows. This value must be the same for all channels in the source image, and for all the destination buffers.

vImage Functions***srcRowBytes***

The number of bytes from the beginning of a channel's row to the beginning of that channel's next row. This value must be the same for all channels of the source image. (It does not have to be the same as the *rowBytes* values of the destination buffers. Each destination buffer can have its own *rowBytes* value.)

flags

The following flag is used:

kvImageDoNotTile

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

See “[General Conversions](#)” (page 113).

This function does not work in place.

vImageConvert_ChunkToPlanarF

This function separates a source image into a collection of corresponding planar destination images, one for each channel of the original image. The pixel values for each channel of the source image and for each of the destination images must be floating-point values.

```
vImage_Error vImageConvert_ChunkToPlanarF (
    const void **srcChannels[],
    const vImage_Buffer *destPlanarBuffers[],
    unsigned int channelCount,
    size_t srcStrideBytes,
    unsigned int srcWidth,
    unsigned int srcHeight,
    unsigned int srcRowBytes,
    vImage_Flags flags
);
```

Parameters***srcChannels***

An array of pointers. The array is of length *channelCount*. Each pointer points to the first pixel value of a channel in the source image. There is one pointer for each channel.

destPlanarBuffers

An array of objects of type `vImage_Buffer`. The array is of length *channelCount*. The buffers are in PlanarF format. On return, each buffer contains a planar image equivalent to the corresponding channel of the source image. Each of the buffers must have the same height (in pixels) and the same width (in pixels), but they may have different *rowBytes* values.

channelCount

The number of channels in the source image.

srcStrideBytes

The number of bytes from one pixel value in a given channel to the next pixel of that channel (within a row). This value must be the same for all channels.

vImage Functions*srcWidth*

The number of pixels in a row. This value must be the same for all channels in the source image, and for all the destination buffers.

srcHeight

The number of rows. This value must be the same for all channels in the source image, and for all the destination buffers.

srcRowBytes

The number of bytes from the beginning of a channel's row to the beginning of that channel's next row. This value must be the same for all channels of the source image. (It does not have to be the same as the *rowBytes* values of the destination buffers. Each destination buffer can have its own *rowBytes* value.)

flags

The following flag is used:

kvImageDoNotTile

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

See “[General Conversions](#)” (page 113).

This function does not work in place.

vImageConvert_FTo16S

This function converts a PlanarF image into a special format in which each pixel is a 16-bit signed integer. Each pixel value is first offset and scaled by user-supplied values, and then changed to a 16-bit signed integer (rounded and clipped as necessary).

```
vImage_Error vImageConvert_FTo16S (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    float offset,
    float scale,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the PlanarF source image.

dest

A destination buffer. This is an object of type `vImage_Buffer`, but it is not in one of the four standard vImage formats. Each pixel value is a 16-bit signed integer.

offset

An offset value. This value is subtracted from every pixel.

vImage Functions*scale*

A scale value. Every pixel is divided by this value (after offset).

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

`vImageConvert_16SToF` and `vImageConvert_FTo16S` are designed to be inverse transformations if you use the same offset value and the same scale value in the two functions. (The inversion is not precise due to round-off error.) This requires the two functions to use these values differently (and in a different order). In `vImageConvert_16SToF`, each pixel value is multiplied by *scale*, and then added to *offset*. In `vImageConvert_FTo16S`, each pixel value has *offset* subtracted from it, then is divided by *scale*.

This function works in place.

vImageConvert_FTo16U

This function converts a PlanarF image into a special format in which each pixel is a 16-bit unsigned integer. Each pixel value is first offset and scaled by user-supplied values, and then changed to a 16-bit unsigned integer (rounded and clipped as necessary).

```
vImage_Error vImageConvert_FTo16U (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    float offset, float scale,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the PlanarF source image.

dest

A destination buffer. This is an object of type `vImage_Buffer`, but it is not in one of the four standard vImage formats. Each pixel value is a 16-bit unsigned integer.

offset

An offset value. This value is subtracted from every pixel.

scale

A scale value. Every pixel is divided by this value (after offset).

vImage Functions*flags*

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

`vImageConvert_16UToF` and `vImageConvert_FTo16U` are designed to be inverse transformations if you use the same offset value and the same scale value in the two functions. (The inversion is not precise due to round-off error.) This requires the two functions to use these values differently (and in a different order). In `vImageConvert_16UToF`, each pixel value is multiplied by `scale`, and then added to `offset`. In `vImageConvert_FTo16U`, each pixel value has `offset` subtracted from it, then is divided by `scale`.

This function works in place.

vImageConvert_Planar16FtoPlanarF

This function converts a Planar16F image to a PlanarF image, placing the new PlanarF image in a destination buffer. The Planar16F format uses 16-bit floating-point numbers (it is identical to OpenEXR).

```
vImage_Error vImageConvert_Planar8toPlanarF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image in Planar16F format.

dest

A pointer to a structure of type `vImage_Buffer` in PlanarF format. On return, it contains the destination image.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

In conformance with IEEE-754, all signaling NaNs are quieted during the conversion (OpenEXR-1.2.1 does not do this.).

vImage Functions

This function does not work in place.

vImageConvert_Planar8to16U

This function converts from Planar8 format to 16U. 16U is a planar format in which the pixels are 16-bit unsigned integer values.

```
vImage_Error vImageConvert_Planar8_to16U (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing a Planar8 image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains a 16U image.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The conversion from 8-bit to 16-bit values is

```
uint16_t result = (srcPixel * 65535 + 127) / 255
```

Note that this function can also be used to convert an ARGB8888 image to a 4-channel interleaved 16U image. Simply multiply the width of the destination buffer by four.

This function does not work in place.

vImageConvert_Planar8toARGB1555

This function takes four Planar8 images and combines them into an ARGB1555 image, using each Planar8 image as one channel of the ARGB1555: alpha, red, green, and blue. (The ARGB1555 format has 16-bit pixels with 1 bit for alpha and 5 bits each for red, green, and blue.)

```
vImage_Error vImageConvert_Planar8toARGB1555 (
    const vImage_Buffer *srcA,
    const vImage_Buffer *srcR,
    const vImage_Buffer *srcG,
    const vImage_Buffer *srcB,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

vImage Functions**Parameters***srcA*

A pointer to a structure of type `vImage_Buffer` containing a Planar8 image to be used as the alpha channel of the destination image.

srcR

A pointer to a structure of type `vImage_Buffer` containing a Planar8 image to be used as the red channel of the destination image.

srcG

A pointer to a structure of type `vImage_Buffer` containing a Planar8 image to be used as the green channel of the destination image.

srcB

A pointer to a structure of type `vImage_Buffer` containing a Planar8 image to be used as the blue channel of the destination image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image in ARGB1555 format.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The 8-bit pixels in the destination image are calculated as

```
Pixel8 alpha = 1bitAlphaChannel * 255
Pixel8 red   = (5bitRedChannel * 255 + 15) / 31
Pixel8 green = (5bitGreenChannel * 255 + 15) / 31
Pixel8 blue  = (5bitBlueChannel * 255 + 15) / 31
```

This function does not work in place.

`vImageConvert_Planar8toARGB8888`

This function takes four Planar8 images and combines them into an ARGB8888 image, using each Planar8 image as one channel of the ARGB8888: alpha, red, green, and blue.

```
vImage_Error vImageConvert_Planar8toARGB8888 (
    const vImage_Buffer *srcA,
    const vImage_Buffer *srcR,
    const vImage_Buffer *srcG,
    const vImage_Buffer *srcB,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

vImage Functions**Parameters***srcA*

A pointer to a structure of type `vImage_Buffer` containing a Planar8 image to be used as the alpha channel of the destination image.

srcR

A pointer to a structure of type `vImage_Buffer` containing a Planar8 image to be used as the red channel of the destination image.

srcG

A pointer to a structure of type `vImage_Buffer` containing a Planar8 image to be used as the green channel of the destination image.

srcB

A pointer to a structure of type `vImage_Buffer` containing a Planar8 image to be used as the blue channel of the destination image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The source and destination buffers must have the same height and the same width (in pixels). However, the source buffers contain Planar8 images, while the destination buffer will contain an ARGB8888 image. The destination buffer must be large enough, in bytes, to contain the resulting image.

This function does not work in place.

vImageConvert_Planar8toPlanarF

This function converts a Planar8 image to a PlanarF image, placing the new PlanarF image in a destination buffer. The Planar8 pixel values 0 to 255 are mapped linearly into the floating-point values `minFloat` to `maxFloat`. clips the pixel values of an image, using a caller-supplied minimum and maximum value, and puts the resulting image in a destination buffer. The source image must be in Planar8 format.

```
vImage_Error vImageConvert_Planar8toPlanarF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    Pixel_F maxFloat,
    Pixel_F minFloat,
    vImage_Flags flags
);
```

vImage Functions**Parameters***src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image.

maxFloat

The maximum pixel value for the destination image.

minFloat

The minimum pixel value for the destination image.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The source and destination buffers must have the same height and the same width (in pixels). However, the destination buffer must be large enough, in bytes, to contain the PlanarF destination image.

This function does not work in place.

`vImageConvert_Planar8toRGB565`

This function takes three Planar8 images and combines them into an RGB565 image, using each Planar8 image as one channel of the RGB565 image: red, green, and blue. (The RGB565 format has 16-bit pixels with 5 bits for red, 6 for green, and 5 for blue.)

```
vImage_Error vImageConvert_Planar8toRGB565 (
    const vImage_Buffer *srcR,
    const vImage_Buffer *srcG,
    const vImage_Buffer *srcB,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters*srcR*

A pointer to a structure of type `vImage_Buffer` containing a Planar8 image to be used as the red channel of the destination image.

srcG

A pointer to a structure of type `vImage_Buffer` containing a Planar8 image to be used as the green channel of the destination image.

srcB

A pointer to a structure of type `vImage_Buffer` containing a Planar8 image to be used as the blue channel of the destination image.

vImage Functions*dest*

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image in RGB565 format.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The pixels in the destination image are calculated as

```
uint32_t red    = (8bitRedChannel * (31*2) + 255) / (255*2)
uint32_t green = (8bitGreenChannel * 63 + 127) / 255
uint32_t blue  = (8bitBlueChannel * 31 + 127) / 255
uint16_t RGB565pixel = (red << 11) | (green << 5) | blue
```

This function does not work in place.

vImageConvert_Planar8toRGB888

This function takes three Planar8 images and combines them into an RGB888 image, using each Planar8 image as one channel of the RGB888 image: red, green, and blue.

```
vImage_Error vImageConvert_Planar8toARGB8888 (
const vImage_Buffer *planarRed,
const vImage_Buffer *planarGreen,
const vImage_Buffer *planarBlue,
const vImage_Buffer *rgbDest,
vImage_Flags flags
);
```

Parameters*planarRed*

A pointer to a structure of type `vImage_Buffer` containing a Planar8 image to be used as the red channel of the destination image.

planarGreen

A pointer to a structure of type `vImage_Buffer` containing a Planar8 image to be used as the green channel of the destination image.

planarBlue

A pointer to a structure of type `vImage_Buffer` containing a Planar8 image to be used as the blue channel of the destination image.

rgbDest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image.

vImage Functions*flags*

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The source and destination buffers must have the same height and the same width (in pixels). However, the source buffers contain Planar8 images, while the destination buffer will contain an RGB888 image. The destination buffer must be large enough, in bytes, to contain the resulting image.

This function does not work in place.

`vImageConvert_PlanarFtoRGBFFF`

This function takes three PlanarF images and combines them into an RGBFFF image, using each PlanarF image as one channel of the RGBFFF image: red, green, and blue.

```
vImage_Error vImageConvert_Planar8toARGB8888 (
    const vImage_Buffer *planarRed,
    const vImage_Buffer *planarGreen,
    const vImage_Buffer *planarBlue,
    const vImage_Buffer *rgbDest,
    vImage_Flags flags
);
```

Parameters*planarRed*

A pointer to a structure of type `vImage_Buffer` containing a PlanarF image to be used as the red channel of the destination image.

planarGreen

A pointer to a structure of type `vImage_Buffer` containing a PlanarF image to be used as the green channel of the destination image.

planarBlue

A pointer to a structure of type `vImage_Buffer` containing a PlanarF image to be used as the blue channel of the destination image.

rgbDest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

vImage Functions**Discussion**

The source and destination buffers must have the same height and the same width (in pixels). However, the source buffers contain PlanarF images, while the destination buffer will contain an ARGBFFF image. The destination buffer must be large enough, in bytes, to contain the resulting image.

This function does not work in place.

vImageConvert_PlanarFtoARGBFFF

This function takes four PlanarF images and combines them into an ARGBFFFF image, using each PlanarF image as one channel of the ARGBFFFF: alpha, red, green, and blue.

```
vImage_Error vImageConvert_PlanarFtoARGBFFF (
    const vImage_Buffer *srcA,
    const vImage_Buffer *srcR,
    const vImage_Buffer *srcG,
    const vImage_Buffer *srcB,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters

srcA

A pointer to a structure of type `vImage_Buffer` containing a PlanarF image to be used as the alpha channel of the destination image.

srcR

A pointer to a structure of type `vImage_Buffer` containing a PlanarF image to be used as the red channel of the destination image.

srcG

A pointer to a structure of type `vImage_Buffer` containing a PlanarF image to be used as the green channel of the destination image.

srcB

A pointer to a structure of type `vImage_Buffer` containing a PlanarF image to be used as the blue channel of the destination image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

vImage Functions**Discussion**

The *srcA*, *srcR*, *srcG*, and *srcB* buffers, and destination buffer, must have the same height and the same width (in pixels). However, the source buffers contain PlanarF images, while the destination buffer will contain an ARGBFFFF image. The destination buffer must be large enough, in bytes, to contain the resulting image.

This function does not work in place.

vImageConvert_PlanarFtoPlanar16F

This function converts a PlanarF image to a Planar16F image, placing the new Planar16F image in a destination buffer. The Planar16F format uses 16-bit floating-point numbers (it is identical to OpenEXR).

```
vImage_Error vImageConvert_Planar8toPlanarF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters

src

A pointer to a structure of type `vImage_Buffer` containing the source image in PlanarF format.

dest

A pointer to a structure of type `vImage_Buffer` in Planar16F format. On return, it contains the destination image.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

In conformance with IEEE-754, all signaling NaNs are quieted during the conversion (OpenEXR-1.2.1 does not do this).

This function works in place.

vImageConvert_PlanarFtoPlanar8

This function converts a PlanarF image to a Planar8 image, placing the new Planar8 image in a destination buffer. The PlanarF pixel values are clipped to minFloat and maxFloat; then, the range from minFloat to maxFloat in the source image is mapped linearly to the range 0 to 255 in the destination image. The source image must be in PlanarF format.

```
vImage_Error vImageConvert_PlanarFtoPlanar8 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    Pixel_F minFloat,
    Pixel_F maxFloat,
```

vImage Functions

```
Pixel_F minFloat,
vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image.

maxFloat

A maximum pixel value. Pixel values greater than this in the source image are clipped to this value before being mapped to the destination image.

minFloat

A minimum pixel value. Pixel values less than this in the source image are clipped to this value before being mapped to the destination image.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The source and destination buffers must have the same height and the same width (in pixels). However, the source buffer contains a PlanarF image, but the destination buffer will contain a Planar8 image. Therefore, under most circumstances, the destination buffer will be considerably smaller, in bytes, than the source buffer.

This function works in place.

vImageConvert_PlanarToChunky8

This function takes a collection of planar source images and combines them into a single interleaved destination image, with one channel for each planar image. The source images must all be in Planar8 format.

```
vImage_Error vImageConvert_PlanarToChunky8 (
    const vImage_Buffer *srcPlanarBuffers[],
    void *destChannels[],
    unsigned int channelCount,
    size_t destStrideBytes,
    unsigned int destWidth,
    unsigned int destHeight,
    unsigned int destRowBytes,
    vImage_Flags flags
);
```

Parameters*srcPlanarBuffers*

An array of objects of type `vImage_Buffer`. The array is of length `channelCount`. The buffers are in Planar8 format. Each buffer contains one planar source image. The buffers must all have the same width (in pixels) and the same height (in pixels). They may have different `rowBytes` values.

destChannels

An array of pointers. The array is of length `channelCount`. Each pointer points to the start of the data area for a channel of the destination image. The pixel values in the channel will be filled in by the function call, using the corresponding source image for each channel.

channelCount

The number of source images in the `srcPlanarBuffers` array. This is also the number of channels in the destination image.

destStrideBytes

The number of bytes from one pixel value in a given channel to the next pixel of that channel (within a row). This value will be used for all channels.

destWidth

The number of pixels in a row. This value will be used for all channels. It must be the same as the width of each of the planar source images.

destHeight

The number of rows. This value will be used for all channels. It must be the same as the height of each of the planar source images.

destRowBytes

The number of bytes from the beginning of a channel's row to the beginning of that channel's next row. This value will be used for all channels. (It does not have to be the same as the `rowBytes` values of the source buffers. Each source buffer can have its own `rowBytes` value.)

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

See “[General Conversions](#)” (page 113).

This function does not work in place.

`vImageConvert_PlanarToChunkyF`

This function takes a collection of planar source images and combines them into a single interleaved destination image, with one channel for each planar image. The source images must all be in PlanarF format.

```
vImage_Error vImageConvert_PlanarToChunkyF (
```

vImage Functions

```
const vImage_Buffer *srcPlanarBuffers[],
void *destChannels[],
unsigned int channelCount,
size_t destStrideBytes,
unsigned int destWidth,
unsigned int destHeight,
unsigned int destRowBytes,
vImage_Flags flags
);
```

Parameters*srcPlanarBuffers*

An array of objects of type `vImage_Buffer`. The array is of length `channelCount`. The buffers are in PlanarF format. Each buffer contains one planar source image. The buffers must all have the same width (in pixels) and the same height (in pixels). They may have different `rowBytes` values.

destChannels

An array of pointers. The array is of length `channelCount`. Each pointer points to the start of the data area for a channel of the destination image. The pixel values in the channels will be filled in by the function call, using the corresponding source image for each channel.

channelCount

The number of source images in the `srcPlanarBuffers` array. This is also the number of channels in the destination image.

destStrideBytes

The number of bytes from one pixel value in a given channel to the next pixel of that channel (within a row). This value will be used for all channels.

destWidth

The number of pixels in a row. This value will be used for all channels. It must be the same as the width of each of the planar source images.

destHeight

The number of rows. This value will be used for all channels. It must be the same as the height of each of the planar source images.

destRowBytes

The number of bytes from the beginning of a channel's row to the beginning of that channel's next row. This value will be used for all channels. (It does not have to be the same as the `rowBytes` values of the source buffers. Each source buffer can have its own `rowBytes` value.)

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

See “[General Conversions](#)” (page 113).

This function does not work in place.

vImageConvert_RGB565toPlanar8

This function takes an RGB565 image and converts it into three Planar8 images, one each for each of the three channels red, green, and blue. (The RGB565 format has 16-bit pixels with 5 bits for red, 6 for green, and 5 for blue.)

```
vImage_Error vImageConvert_RGB565toPlanar8 (
    const vImage_Buffer *src,
    const vImage_Buffer *destR,
    const vImage_Buffer *destG,
    const vImage_Buffer *destB,
    vImage_Flags flags
);
```

Parameters

src

A pointer to a structure of type `vImage_Buffer` containing an RGB565 image.

destR

A pointer to a structure of type `vImage_Buffer`. On return, it contains a Planar8 image equivalent to the red channel of the source image.

destG

A pointer to a structure of type `vImage_Buffer`. On return, it contains a Planar8 image equivalent to the green channel of the source image.

destB

A pointer to a structure of type `vImage_Buffer`. On return, it contains a Planar8 image equivalent to the blue channel of the source image.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The 8-bit pixels in the destination image are calculated as

```
Pixel8 red    = (5bitRedChannel * 255 + 15) / 31
Pixel8 green = (6bitGreenChannel * 255 + 31) / 63
Pixel8 blue   = (5bitBlueChannel * 255 + 15) / 31
```

This function works in place for one destination buffer. The others must be allocated separately.

vImageConvert_RGB565toARGB8888

This function takes an RGB565 image and converts it into an ARGB8888 image, using a specified 8-bit alpha value. (The RGB565 format has 16-bit pixels with 5 bits for red, 6 for green, and 5 for blue.)

```
vImage_Error vImageConvert_RGB565toARGB8888 (
    Pixel_8 alpha,
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters*alpha*

A value of type `Pixel_8` to be used as the alpha value for all pixels in the destination image.

src

A pointer to a structure of type `vImage_Buffer` containing an RGB565 image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains an ARGB8888 image.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The 8-bit pixels in the destination image are calculated as

```
Pixel8 alpha = alpha
Pixel8 red   = (5bitRedChannel * 255 + 15) / 31
Pixel8 green = (6bitGreenChannel * 255 + 31) / 63
Pixel8 blue  = (5bitBlueChannel * 255 + 15) / 31
```

This function does not work in place.

vImageConvert_RGB888toARGB8888

This function takes an RGB888 image and converts it into an ARGB8888 image, using either a specified Planar8 alpha plane or a specified 8-bit alpha value.

```
vImage_Error vImageConvert_RGB888toARGB8888 (
    const vImage_Buffer *rgbSrc,
    const vImage_Buffer *aSrc,
    Pixel_8 alpha,
    const vImage_Buffer *argbDest,
    vImage_Flags flags
);
```

vImage Functions**Parameters***rgbSrc*

A pointer to a structure of type `vImage_Buffer` containing an RGB888 image.

aSrc

A pointer to a structure of type `vImage_Buffer` containing a Planar8 alpha plane to be used in the destination image. If this pointer is `NULL`, the single alpha value in the next parameter is used for all pixels in the destination image.

alpha

A value of type `Pixel_8` to be used as the alpha value for all pixels in the destination image, if and only if the preceding parameter is `NULL`.

argbDest

A pointer to a structure of type `vImage_Buffer`. On return, it contains an ARGB8888 image.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

This function does not work in place.

`vImageConvert_RGB888toPlanar8`

This function takes an RGB888 image and separates it into three Planar8 images, one each for each of the three channels red, green, and blue.

```
vImage_Error vImageConvert_RGB888toPlanar8 (
    const vImage_Buffer *rgbSrc,
    const vImage_Buffer *redDest,
    const vImage_Buffer *greenDest,
    const vImage_Buffer *blueDest,
    vImage_Flags flags
);
```

Parameters*rgbSrc*

A pointer to a structure of type `vImage_Buffer` containing an RGB888 image.

redDest

A pointer to a structure of type `vImage_Buffer`. On return, it contains a Planar8 image equivalent to the red channel of the source image.

greenDest

A pointer to a structure of type `vImage_Buffer`. On return, it contains a Planar8 image equivalent to the green channel of the source image.

vImage Functions*blueDest*

A pointer to a structure of type `vImage_Buffer`. On return, it contains a Planar8 image equivalent to the blue channel of the source image.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The source image and the destination buffers, must all have the same height and the same width (in pixels). However, the source buffer contains an RGB888 image, while the destination buffers each contain a Planar8 image.

This function works in place for one destination buffer. The others must be allocated separately.

`vImageConvert_RGBFFFtoPlanarF`

This function takes an RGBFFF image and separates it into three PlanarF images, one each for each of the three channels red, green, and blue.

```
vImage_Error vImageConvert_RGBFFFtoPlanarF (
    const vImage_Buffer *rgbSrc,
    const vImage_Buffer *redDest,
    const vImage_Buffer *greenDest,
    const vImage_Buffer *blueDest,
    vImage_Flags flags
);
```

Parameters*rgbSrc*

A pointer to a structure of type `vImage_Buffer` containing an RGBFFF image.

redDest

A pointer to a structure of type `vImage_Buffer`. On return, it contains a PlanarF image equivalent to the red channel of the source image.

greenDest

A pointer to a structure of type `vImage_Buffer`. On return, it contains a PlanarF image equivalent to the green channel of the source image.

blueDest

A pointer to a structure of type `vImage_Buffer`. On return, it contains a PlanarF image equivalent to the blue channel of the source image.

vImage Functions*flags*

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The source image and the destination buffers, must all have the same height and the same width (in pixels). However, the source buffer contains an RGBFFF image, while the destination buffers each contain a PlanarF image.

This function works in place for one destination buffer. The others must be allocated separately.

`vImage_FlattenARGB8888ToRGB888`

This functions transforms an ARGB8888 image to an RGB888 image against an opaque background of a specified color.

```
vImage_Error vImage_FlattenARGB8888ToRGB888 (
    const vImage_Buffer *argb8888Src,
    const vImage_Buffer *rgb888dest,
    Pixel_8888 backgroundColor,
    bool isImagePremultiplied,
    vImage_Flags flags
);
```

Parameters*argb8888Src*

A source buffer in ARGB8888 format.

rgb888dest

A destination buffer. This is an object of type `vImage_Buffer`, in RGB888 format.

backgroundColor

An 8-bit interleaved pixel value.

isImagePremultiplied

True if the source image is premultiplied, false otherwise..

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

vImage Functions**Discussion**

If the source image is premultiplied, each channel value for a pixel in the destination image is calculated as follows (where i is the source value for the channel):

```
new value = (i * 255 + (255 - alpha) * backgroundColor + 127) / 255
```

If the source image is not premultiplied, each channel value for a pixel in the destination image is calculated as follows (where i is the source value for the channel):

```
new value = (i * alpha + (255 - alpha) * backgroundColor + 127) / 255
```

This function works in place.

vImage_FlattenARGBFFFFToRGBFFF

This functions transforms an ARGBFFFF image to an RGBFFF image against an opaque background of a specified color.

```
vImage_Error vImage_FlattenARGBFFFFToRGBFFF (
    const vImage_Buffer *argbFFFFSrc,
    const vImage_Buffer *rgbFFFdest,
    Pixel_FFFF backgroundColor,
    bool isImagePremultiplied,
    vImage_Flags flags
);
```

Parameters

argbFFFFSrc

A source buffer in ARGBFFFF format.

rgbFFFdest

A destination buffer. This is an object of type `vImage_Buffer`, in RGBFFF format.

backgroundColor

A floating-point interleaved pixel value.

isImagePremultiplied

True if the source image is premultiplied, false otherwise..

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

If the source image is premultiplied, each channel value for a pixel in the destination image is calculated as follows (where i is the source value for the channel):

```
newcolor = i + (1.0f - alpha) * backgroundColor
```

vImage Functions

If the source image is not premultiplied, each channel value for a pixel in the destination image is calculated as follows (where i is the source value for the channel):

```
newcolor = i * alpha + (1.0f - alpha) * backgroundColor
```

This function works in place.

vImageOverwriteChannels_ARGB8888

This function can be used to overwrite one or more planes of an image buffer with a specified planar buffer. For each plane that is overwritten, each pixel value is replaced with the corresponding pixel value from the planar buffer.

```
vImage_Error vImageOverwriteChannels_ARGB8888 (
    const vImage_Buffer *newSrc,
    const vImage_Buffer *origSrc,
    const vImage_Buffer *dest,
    uint8_t copyMask,
    vImage_Flags flags
);
```

Parameters

newSrc

A source buffer. This is an object of type `vImage_Buffer` in Planar8 format.

origSrc

A source buffer. This is an object of type `vImage_Buffer` in ARGB8888 format.

dest

A destination buffer. This is an object of type `vImage_Buffer`, in ARGB8888 format.

copyMask

An output value that selects the plane(s) from the ARGB8888 source buffer to be replaced with data from the Planar8 source buffer. 0x8 selects the alpha channel, 0x4 the red channel, 0x2 the green channel, and 0x1 the blue channel. Add these values together to select multiple channels.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

This function works in place (but the planar buffer must be allocated separately).

vImageOverwriteChannels_ARGBFFFF

This function can be used to overwrite one or more planes of an image buffer with a specified planar buffer. For each plane that is overwritten, each pixel value is replaced with the corresponding pixel value from the planar buffer.

vImage Functions

```
vImage_Error vImageOverwriteChannels_ARGBFFFF (
    const vImage_Buffer *newSrc,
    const vImage_Buffer *origSrc,
    const vImage_Buffer *dest,
    uint8_t copyMask,
    vImage_Flags flags
);
```

Parameters*newSrc*

A source buffer. This is an object of type `vImage_Buffer` in PlanarF format.

origSrc

A source buffer. This is an object of type `vImage_Buffer` in ARGBFFFF format.

dest

A destination buffer. This is an object of type `vImage_Buffer`, in ARGBFFFF format.

copyMask

An output value that selects the plane(s) from the ARGBFFFF source buffer to be replaced with data from the PlanarF source buffer. 0x8 selects the alpha channel, 0x4 the red channel, 0x2 the green channel, and 0x1 the blue channel. Add these values together to select multiple channels.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

This function works in place (but the planar buffer must be allocated separately).

`vImageOverwriteChannelsWithScalar_ARGB8888`

This function can be used to overwrite the pixels of one or more planes of an image buffer with a specified scalar value.

```
vImage_Error vImageOverwriteChannelsWithScalar_ARGB8888 (
    Pixel_8 scalar,
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    uint8_t copyMask,
    vImage_Flags flags
);
```

Parameters*scalar*

An 8-bit pixel value.

vImage Functions*src*

A source buffer. This is an object of type `vImage_Buffer` in ARGB8888 format.

dest

A destination buffer. This is an object of type `vImage_Buffer`, in ARGB8888 format.

copyMask

An output value that selects the plane(s) from the ARGB8888 source buffer to be replaced with the scalar value in all pixels. 0x8 selects the alpha channel, 0x4 the red channel, 0x2 the green channel, and 0x1 the blue channel. Add these values together to select multiple channels.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

This function works in place.

`vImageOverwriteChannelsWithScalar_ARGBFFFF`

This function can be used to overwrite the pixels of one or more planes of an image buffer with a specified scalar value.

```
vImage_Error vImageOverwriteChannelsWithScalar_ARGBFFFF (
    Pixel_F scalar,
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    uint8_t copyMask,
    vImage_Flags flags
);
```

Parameters*scalar*

A floating-point pixel value.

src

A source buffer. This is an object of type `vImage_Buffer` in ARGBFFFF format.

dest

A destination buffer. This is an object of type `vImage_Buffer`, in ARGBFFFF format.

copyMask

An output value that selects the plane(s) from the ARGBFFFF source buffer to be replaced with the scalar value in all pixels. 0x8 selects the alpha channel, 0x4 the red channel, 0x2 the green channel, and 0x1 the blue channel. Add these values together to select multiple channels.

vImage Functions*flags*

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

This function works in place.

`vImageOverwriteChannelsWithScalar_Planar8`

This function can be used to overwrite a Planar8 image buffer with a specified value.

```
vImage_Error vImageOverwriteChannelsWithScalar_Planar8 (
    Pixel_8 scalar,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters*scalar*

An 8-bit pixel value.

dest

A destination buffer. This is an object of type `vImage_Buffer`, in Planar8 format.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

This function works in place.

`vImageOverwriteChannelsWithScalar_PlanarF`

This function can be used to overwrite a PlanarF image buffer with a specified value.

```
vImage_Error vImageOverwriteChannelsWithScalar_PlanarF (
    Pixel_F scalar,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

vImage Functions**Parameters***scalar*

A floating-point pixel value.

dest

A destination buffer. This is an object of type `vImage_Buffer`, in PlanarF format.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

This function works in place.

`vImagePermuteChannels_ARGB8888`

This function reorders the channels in an ARGB8888 image.

```
vImage_Error vImagePermuteChannels_ARGB8888 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    const uint8_t permuteMap[4],
    vImage_Flags flags
);
```

Parameters*src*

A source buffer. This is an object of type `vImage_Buffer`, in ARGB8888 format.

dest

A destination buffer. This is an object of type `vImage_Buffer`, in ARGB8888 format.

permuteMap

An array of four 8-bit integers with the values 0, 1, 2, and 3, in some order. The *i*th value specifies the plane from the source image that will be copied to the *i*th plane of the destination image. 0 denotes the alpha channel, 1 the red channel, 2 the green channel, and 3 the blue channel.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

vImage Functions**Discussion**

This function works in place.

vImagePermuteChannels_ARGBFFFF

This function reorders the channels in an ARGBFFFF image.

```
vImage_Error vImagePermuteChannels_ARGBFFFF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    const uint8_t permuteMap[4],
    vImage_Flags flags
);
```

Parameters

src

A source buffer. This is an object of type `vImage_Buffer`, in ARGBFFFF format.

dest

A destination buffer. This is an object of type `vImage_Buffer`, in ARGBFFFF format.

permuteMap

An array of four 8-bit integers with the values 0, 1, 2, and 3, in some order. The *i*th value specifies the plane from the source image that will be copied to the *i*th plane of the destination image. 0 denotes the alpha channel, 1 the red channel, 2 the green channel, and 3 the blue channel.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

This function works in place.

vImageTableLookUp_ARGB8888

This function transforms an ARGB8888 image by substituting each pixel value with another pixel value, as specified by four lookup tables. Each pixel is separated into its four channels—alpha, red, green, and blue. The substitution is done for each channel separately, using the appropriate table. Then the pixel is recombined into a single ARGB8888 value. The transformed image is placed in a destination buffer.

```
vImage_Error vImageTableLookUp_ARGB8888 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    const Pixel_8 alphaTable[256],
    const Pixel_8 redTable[256],
    const Pixel_8 greenTable[256],
```

vImage Functions

```
const Pixel_8 blueTable[256],
vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the ARGB8888 source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the ARGB8888 destination image.

alphaTable

A lookup table to be used for the alpha channel of the source image. If you pass NULL for this table, the alpha channel will be copied unchanged to the destination buffer.

redTable

A lookup table to be used for the red channel of the source image. If you pass NULL for this table, the red channel will be copied unchanged to the destination buffer.

greenTable

A lookup table to be used for the green channel of the source image. If you pass NULL for this table, the green channel will be copied unchanged to the destination buffer.

blueTable

A lookup table to be used for the blue channel of the source image. If you pass NULL for this table, the blue channel will be copied unchanged to the destination buffer.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The source image and destination buffer must have the same height and the same width (in pixels). This function works in place.

Note that you can not use this function to do a completely arbitrary color mapping. If two different source colors have the same green component (just for example), they must be mapped to two destination colors whose green components are equal. They can not be mapped to two arbitrary colors.

`vImageTableLookUp_Planar8`

This function transforms an Planar8 image by substituting each pixel value with another pixel value, as specified by a lookup tables. The transformed image is placed in a destination buffer.

```
vImage_Error vImageTableLookUp_Planar8 (
const vImage_Buffer *src,
const vImage_Buffer *dest,
```

vImage Functions

```
const Pixel_8 table[256],
vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the Planar8 source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the Planar8 destination image.

table

The lookup table.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The source image and destination buffer must have the same height and the same width (in pixels). This function works in place.

Convolution Functions

For conceptual introduction to these functions, please see “[Convolution Operations](#)” (page 29).

vImageConvolve_ARGBFFF

This function performs a convolution of a region of interest within a source image by an M x N kernel, placing the result in a destination buffer. The image must be in ARGBFFFF format. The elements of the kernel must have (non-interleaved) floating-point values. M and N must both be odd. The same kernel is used for all channels.

```
vImage_Error vImageConvolve_ARGBFFF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    vImagePixelCount srcOffsetToROI_X,
    vImagePixelCount srcOffsetToROI_Y,
    const float *kernel,
    uint32_t kernel_height,
    uint32_t kernel_width,
    Pixel_FFFF backgroundColor,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

kernel

A pointer to the kernel data. The kernel data is a packed array of floating-point values.

kernel_height

The height of the kernel in pixels. This value must be odd.

kernel_width

The width of the kernel in pixels. This value must be odd.

backgroundColor

A pixel value to be used as a background color. Only used if the `kvImageBackgroundColorFill` flag is set.

vImage Functions*flags*

The following flags are used:

`kvImageCopyInPlace`

If set, the function will use the Copy In Place technique to access pixels outside the source image.

`kvImageTruncateKernel`

If set, only the values of the kernel that overlap the image will be used.

`kvImageBackgroundColorFill`

If set, the function will use the Background Color Fill technique to access pixels outside the source image.

`kvImageEdgeExtend`

If set, the function will use the Edge Extend technique to access pixels outside the source image.

`kvImageLeaveAlphaUnchanged`

If set, the function will copy the alpha channel to the destination image unchanged.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

The `kvImageCopyInPlace`, `kvImageTruncateKernel`, `kvImageBackgroundColorFill`, and `kvImageEdgeExtend` flags are mutually exclusive. You must set exactly one of these flags. See “[Pixels Outside the Image Buffer](#)” (page 21).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

In Mac OS X 10.4, some of the parameter types have been changed. For example, `unsigned int` has been changed to `uint32_t` or to `size_t` (for byte counts) or `vImagePixelCount` (for pixel counts). Although the changes are fully backward-compatible with existing binaries, it is advisable to update and recompile existing code.

In addition to supplying space for the destination image, the `dest` parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by `dest`.

For most purposes the kernel should be normalized—that is, the sum of its elements should be 1.0f. A non-normalized kernel will either brighten or darken the image.

`vImageConvolve_ARGB8888`

This function performs a convolution of a region of interest within a source image by an M x N kernel, then divides the pixel values by a divisor, placing the result in a destination buffer. The image must be in ARGB8888 format. The elements of the kernel must have (non-interleaved) integer values. M and N must both be odd. The same kernel is used for all channels.

```
vImage_Error vImageConvolve_ARGB8888 (
    const vImage_Buffer *src,
```

vImage Functions

```
const vImage_Buffer *dest,
void *tempBuffer,
vImagePixelCount srcOffsetToROI_X,
vImagePixelCount srcOffsetToROI_Y,
const int16_t *kernel,
uint32_t kernel_height,
uint32_t kernel_width,
int32_t divisor,
Pixel_8888 backgroundColor,
vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

kernel

A pointer to the kernel data. The kernel data is a packed array of integer values.

kernel_height

The height of the kernel in pixels. This value must be odd.

kernel_width

The width of the kernel in pixels. This value must be odd.

divisor

A value to divide the results of the convolution by. This is commonly used for normalization.

backgroundColor

A pixel value to be used as a background color. Only used if the `kvImageBackgroundColorFill` flag is set.

vImage Functions

flags

The following flags are used:

`kvImageCopyInPlace`

If set, the function will use the Copy In Place technique to access pixels outside the source image.

`kvImageTruncateKernel`

If set, only the values of the kernel that overlap the image will be used.

`kvImageBackgroundColorFill`

If set, the function will use the Background Color Fill technique to access pixels outside the source image.

`kvImageEdgeExtend`

If set, the function will use the Edge Extend technique to access pixels outside the source image.

`kvImageLeaveAlphaUnchanged`

If set, the function will copy the alpha channel to the destination image unchanged.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

The `kvImageCopyInPlace`, `kvImageTruncateKernel`, `kvImageBackgroundColorFill`, and `kvImageEdgeExtend` flags are mutually exclusive. You must set exactly one of these flags. See “[Pixels Outside the Image Buffer](#)” (page 21).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

In Mac OS X 10.4, some of the parameter types have been changed. For example, `unsigned int` has been changed to `uint32_t` or to `size_t` (for byte counts) or `vImagePixelCount` (for pixel counts). Although the changes are fully backward-compatible with existing binaries, it is advisable to update and recompile existing code.

In addition to supplying space for the destination image, the `dest` parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by `dest`.

Although the pixel values of the source image are in ARGB8888 (integer interleaved) format, the values of the kernel must be (non-interleaved) integer numbers.

The sum of any subset of elements of the kernel should be in the range $-(2^{24})$ to $+(2^{24})-1$. Otherwise integer overflow may occur. If your kernel does not meet this restriction, either use a floating-point format or scale the kernel down to use smaller values.

For most purposes the kernel, together with the divisor, should be normalized. That is, the sum of the kernel’s elements, divided by the divisor, should be 1. A non-normalized kernel will either brighten or darken the image.

vImageConvolve_PlanarF

This function performs a convolution of a region of interest within a source image by an M x N kernel, placing the result in a destination buffer. The image must be in PlanarF format. The elements of the kernel must have floating-point values. M and N must both be odd.

```
vImage_Error vImageConvolve_PlanarF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    vImagePixelCount srcOffsetToROI_X,
    vImagePixelCount srcOffsetToROI_Y,
    const float *kernel,
    uint32_t kernel_height,
    uint32_t kernel_width,
    Pixel_F backgroundColor,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

kernel

A pointer to the kernel data. The kernel data is a packed array of floating-point values.

kernel_height

The height of the kernel in pixels. This value must be odd.

kernel_width

The width of the kernel in pixels. This value must be odd.

vImage Functions*backgroundColor*

A pixel value to be used as a background color. Only used if the `kvImageBackgroundColorFill` flag is set.

flags

The following flags are used:

`kvImageCopyInPlace`

If set, the function will use the Copy In Place technique to access pixels outside the source image.

`kvImageTruncateKernel`

If set, only the values of the kernel that overlap the image will be used.

`kvImageBackgroundColorFill`

If set, the function will use the Background Color Fill technique to access pixels outside the source image.

`kvImageEdgeExtend`

If set, the function will use the Edge Extend technique to access pixels outside the source image.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

The `kvImageCopyInPlace`, `kvImageTruncateKernel`, `kvImageBackgroundColorFill`, and `kvImageEdgeExtend` flags are mutually exclusive. You must set exactly one of these flags. See “[Pixels Outside the Image Buffer](#)” (page 21).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

In Mac OS X 10.4, some of the parameter types have been changed. For example, `unsigned int` has been changed to `uint32_t` or to `size_t` (for byte counts) or `vImagePixelCount` (for pixel counts). Although the changes are fully backward-compatible with existing binaries, it is advisable to update and recompile existing code.

In addition to supplying space for the destination image, the `dest` parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by `dest`.

For most purposes the kernel should be normalized—that is, the sum of its elements should be 1.0f. A non-normalized kernel will either brighten or darken the image.

`vImageConvolve_Planar8`

This function performs a convolution of a region of interest within a source image by an M x N kernel, then divides the pixel values by a divisor, placing the result in a destination buffer. The image must be in Planar8 format. The elements of the kernel must have integer values. M and N must both be odd.

```
vImage_Error vImageConvolve_Planar8 (
```

vImage Functions

```
const vImage_Buffer *src,
const vImage_Buffer *dest,
void *tempBuffer,
vImagePixelCount srcOffsetToROI_X,
vImagePixelCount srcOffsetToROI_Y,
const int16_t *kernel,
uint32_t kernel_height,
uint32_t kernel_width,
int32_t divisor,
Pixel_8 backgroundColor,
vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

kernel

A pointer to the kernel data. The kernel data is a packed array of integer values.

kernel_height

The height of the kernel in pixels. This value must be odd.

kernel_width

The width of the kernel in pixels. This value must be odd.

divisor

A value to divide the results of the convolution with. This is commonly used for normalization.

vImage Functions

backgroundColor

A pixel value to be used as a background color. Only used if the `kvImageBackgroundColorFill` flag is set.

flags

The following flags are used:

`kvImageCopyInPlace`

If set, the function will use the Copy In Place technique to access pixels outside the source image.

`kvImageTruncateKernel`

If set, only the values of the kernel that overlap the image will be used.

`kvImageBackgroundColorFill`

If set, the function will use the Background Color Fill technique to access pixels outside the source image.

`kvImageEdgeExtend`

If set, the function will use the Edge Extend technique to access pixels outside the source image.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

The `kvImageCopyInPlace`, `kvImageTruncateKernel`, `kvImageBackgroundColorFill`, and `kvImageEdgeExtend` flags are mutually exclusive. You must set exactly one of these flags. See “[Pixels Outside the Image Buffer](#)” (page 21).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

In Mac OS X 10.4, some of the parameter types have been changed. For example, `unsigned int` has been changed to `uint32_t` or to `size_t` (for byte counts) or `vImagePixelCount` (for pixel counts). Although the changes are fully backward-compatible with existing binaries, it is advisable to update and recompile existing code.

In addition to supplying space for the destination image, the `dest` parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by `dest`.

The sum of any subset of elements of the kernel should be in the range $-(2^{24})$ to $+(2^{24})-1$. Otherwise integer overflow may occur. If your kernel does not meet this restriction, either use a floating-point format or scale the kernel down to use smaller values.

For most purposes the kernel, together with the divisor, should be normalized. That is, the sum of the kernel’s elements, divided by the divisor, should be 1. A non-normalized kernel will either brighten or darken the image.

vImageConvolveWithBias_ARGBFFFF

This function is a variant of vImageConvolve_ARGBFFFF. It performs a convolution of a region of interest within a source image by an M x N kernel, placing the result in a destination buffer. After the convolution and before any clipping is done, a specified bias value is applied to the result. The image must be in ARGBFFFF format. The elements of the kernel must have (non-interleaved) floating-point values. M and N must both be odd. The same kernel is used for all channels.

```
vImage_Error vImageConvolveWithBias_ARGBFFFF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    vImagePixelCount srcOffsetToROI_X,
    vImagePixelCount srcOffsetToROI_Y,
    const float *kernel,
    uint32_t kernel_height,
    uint32_t kernel_width,
    float bias,
    Pixel_FFFF backgroundColor,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type vImage_Buffer containing the source image.

dest

A pointer to a structure of type vImage_Buffer. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass NULL, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the kvImageGetTempBufferSize flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the kvImageGetTempBufferSize flag, passing the buffer pointer.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

kernel

A pointer to the kernel data. The kernel data is a packed array of floating-point values.

kernel_height

The height of the kernel in pixels. This value must be odd.

vImage Functions

kernel_width

The width of the kernel in pixels. This value must be odd.

bias

The value to be added to each element of the convolution result, before clipping.

backgroundColor

A pixel value to be used as a background color. Only used if the `kvImageBackgroundColorFill` flag is set.

flags

The following flags are used:

`kvImageCopyInPlace`

If set, the function will use the Copy In Place technique to access pixels outside the source image.

`kvImageTruncateKernel`

If set, only the values of the kernel that overlap the image will be used.

`kvImageBackgroundColorFill`

If set, the function will use the Background Color Fill technique to access pixels outside the source image.

`kvImageEdgeExtend`

If set, the function will use the Edge Extend technique to access pixels outside the source image.

`kvImageLeaveAlphaUnchanged`

If set, the function will copy the alpha channel to the destination image unchanged.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

The `kvImageCopyInPlace`, `kvImageTruncateKernel`, `kvImageBackgroundColorFill`, and `kvImageEdgeExtend` flags are mutually exclusive. You must set exactly one of these flags. See “[Pixels Outside the Image Buffer](#)” (page 21).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

In addition to supplying space for the destination image, the `dest` parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by `dest`.

For most purposes the kernel should be normalized—that is, the sum of its elements should be 1.0f. A non-normalized kernel will either brighten or darken the image.

vImageConvolveWithBias_ARGB8888

This function is a variant of vImageConvolve_ARGB8888. It performs a convolution of a region of interest within a source image by an M x N kernel, then divides the pixel values by a divisor, placing the result in a destination buffer. After the convolution and before the divisor is applied or any clipping is done, a specified bias value is applied to the result. The image must be in ARGB8888 format. The elements of the kernel must have (non-interleaved) integer values. M and N must both be odd. The same kernel is used for all channels.

```
vImage_Error vImageConvolveWithBias_ARGB8888 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    vImagePixelCount srcOffsetToROI_X,
    vImagePixelCount srcOffsetToROI_Y,
    const int16_t *kernel,
    uint32_t kernel_height,
    uint32_t kernel_width,
    int32_t divisor,
    int32_t bias,
    Pixel_8888 backgroundColor,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

kernel

A pointer to the kernel data. The kernel data is a packed array of integer values.

vImage Functions

kernel_height

The height of the kernel in pixels. This value must be odd.

kernel_width

The width of the kernel in pixels. This value must be odd.

divisor

A value to divide the results of the convolution by. This is commonly used for normalization.

bias

The value to be added to each element in the convolution result, before the divisor is applied or any clipping is done.

backgroundColor

A pixel value to be used as a background color. Only used if the `kvImageBackgroundColorFill` flag is set.

flags

The following flags are used:

`kvImageCopyInPlace`

If set, the function will use the Copy In Place technique to access pixels outside the source image.

`kvImageTruncateKernel`

If set, only the values of the kernel that overlap the image will be used.

`kvImageBackgroundColorFill`

If set, the function will use the Background Color Fill technique to access pixels outside the source image.

`kvImageEdgeExtend`

If set, the function will use the Edge Extend technique to access pixels outside the source image.

`kvImageLeaveAlphaUnchanged`

If set, the function will copy the alpha channel to the destination image unchanged.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

The `kvImageCopyInPlace`, `kvImageTruncateKernel`, `kvImageBackgroundColorFill`, and `kvImageEdgeExtend` flags are mutually exclusive. You must set exactly one of these flags. See “[Pixels Outside the Image Buffer](#)” (page 21).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

In addition to supplying space for the destination image, the `dest` parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by `dest`.

Although the pixel values of the source image are in ARGB8888 (integer interleaved) format, the values of the kernel must be (non-interleaved) integer numbers.

The sum of any subset of elements of the kernel should be in the range $-(2^{24})$ to $+(2^{24})-1$. Otherwise integer overflow may occur. If your kernel does not meet this restriction, either use a floating-point format or scale the kernel down to use smaller values.

For most purposes the kernel, together with the divisor, should be normalized. That is, the sum of the kernel's elements, divided by the divisor, should be 1. A non-normalized kernel will either brighten or darken the image.

vImageConvolveWithBias_PlanarF

This function is a variant of vImageConvolve_PlanarF. It performs a convolution of a region of interest within a source image by an M x N kernel, placing the result in a destination buffer. After the convolution and before any clipping is done, a specified bias value is applied to the result. The image must be in PlanarF format. The elements of the kernel must have floating-point values. M and N must both be odd.

```
vImage_Error vImageConvolveWithBias_PlanarF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    vImagePixelCount srcOffsetToROI_X,
    vImagePixelCount srcOffsetToROI_Y,
    float *kernel,
    uint32_t kernel_height,
    uint32_t kernel_width,
    float bias,
    Pixel_F backgroundColor,
    vImage_Flags flags
);
```

Parameters

src

A pointer to a structure of type vImage_Buffer containing the source image.

dest

A pointer to a structure of type vImage_Buffer. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass NULL, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the kvImageGetTempBufferSize flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the kvImageGetTempBufferSize flag, passing the buffer pointer.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

vImage Functions

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

kernel

A pointer to the kernel data. The kernel data is a packed array of floating-point values.

kernel_height

The height of the kernel in pixels. This value must be odd.

kernel_width

The width of the kernel in pixels. This value must be odd.

bias

The value to be added to each element of the convolution result, before clipping.

backgroundColor

A pixel value to be used as a background color. Only used if the `kvImageBackgroundColorFill` flag is set.

flags

The following flags are used:

`kvImageCopyInPlace`

If set, the function will use the Copy In Place technique to access pixels outside the source image.

`kvImageTruncateKernel`

If set, only the values of the kernel that overlap the image will be used.

`kvImageBackgroundColorFill`

If set, the function will use the Background Color Fill technique to access pixels outside the source image.

`kvImageEdgeExtend`

If set, the function will use the Edge Extend technique to access pixels outside the source image.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

The `kvImageCopyInPlace`, `kvImageTruncateKernel`, `kvImageBackgroundColorFill`, and `kvImageEdgeExtend` flags are mutually exclusive. You must set exactly one of these flags. See “[Pixels Outside the Image Buffer](#)” (page 21).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

In addition to supplying space for the destination image, the `dest` parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by `dest`.

For most purposes the kernel should be normalized—that is, the sum of its elements should be 1.0f. A non-normalized kernel will either brighten or darken the image.

vImageConvolveWithBias_Planar8

This function is a variant of vImageConvolve_Planar8. It performs a convolution of a region of interest within a source image by an M x N kernel, then divides the pixel values by a divisor, placing the result in a destination buffer. After the convolution and before the divisor is applied or any clipping is done, a specified bias value is applied to the result. The image must be in Planar8 format. The elements of the kernel must have integer values. M and N must both be odd.

```
vImage_Error vImageConvolveWithBias_Planar8 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    vImagePixelCount srcOffsetToROI_X,
    vImagePixelCount srcOffsetToROI_Y,
    int32_t *kernel,
    unsigned int kernel_height,
    unsigned int kernel_width,
    int32_t divisor,
    int32_t bias,
    Pixel_8 backgroundColor,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type vImage_Buffer containing the source image.

dest

A pointer to a structure of type vImage_Buffer. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass NULL, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the kvImageGetTempBufferSize flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the kvImageGetTempBufferSize flag, passing the buffer pointer.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

kernel

A pointer to the kernel data. The kernel data is a packed array of integer values.

vImage Functions

kernel_height

The height of the kernel in pixels. This value must be odd.

kernel_width

The width of the kernel in pixels. This value must be odd.

divisor

A value to divide the results of the convolution with. This is commonly used for normalization.

bias

The value to be added to each element in the convolution result, before the divisor is applied or any clipping is done.

backgroundColor

A pixel value to be used as a background color. Only used if the `kvImageBackgroundColorFill` flag is set.

flags

The following flags are used:

`kvImageCopyInPlace`

If set, the function will use the Copy In Place technique to access pixels outside the source image.

`kvImageTruncateKernel`

If set, only the values of the kernel that overlap the image will be used.

`kvImageBackgroundColorFill`

If set, the function will use the Background Color Fill technique to access pixels outside the source image.

`kvImageEdgeExtend`

If set, the function will use the Edge Extend technique to access pixels outside the source image.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

The `kvImageCopyInPlace`, `kvImageTruncateKernel`, `kvImageBackgroundColorFill`, and `kvImageEdgeExtend` flags are mutually exclusive. You must set exactly one of these flags. See “[Pixels Outside the Image Buffer](#)” (page 21).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

In addition to supplying space for the destination image, the `dest` parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by `dest`.

The sum of any subset of elements of the kernel should be in the range $-(2^{24})$ to $+(2^{24})-1$. Otherwise integer overflow may occur. If your kernel does not meet this restriction, either use a floating-point format or scale the kernel down to use smaller values.

vImage Functions

For most purposes the kernel, together with the divisor, should be normalized. That is, the sum of the kernel's elements, divided by the divisor, should be 1. A non-normalized kernel will either brighten or darken the image.

vImageConvolveMultiKernel_ARGBFFFF

This function is a variant of vImageConvolveWithBias_ARGBFFFF, using a separate kernel and a separate bias value for each channel in the interleaved image. For each channel, it performs a convolution of a region of interest within a source image by one of the four M x N kernels, placing the result in the destination buffer. After the convolution and before any clipping is done, one of the four specified bias value is applied to the result. The image must be in ARGBFFFF format. The elements of each kernel must have (non-interleaved) floating-point values. M and N must both be odd.

```
vImage_Error vImageConvolveMultiKernel_ARGBFFFF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    vImagePixelCount srcOffsetToROI_X,
    vImagePixelCount srcOffsetToROI_Y,
    const float *kernels[4],
    uint32_t kernel_height,
    uint32_t kernel_width,
    const float biases[4],
    Pixel_FFFF backgroundColor,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type vImage_Buffer containing the source image.

dest

A pointer to a structure of type vImage_Buffer. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass NULL, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the kvImageGetTempBufferSize flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the kvImageGetTempBufferSize flag, passing the buffer pointer.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

vImage Functions

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

kernels

An array of four pointers to the data for four kernels. The first kernel is for the alpha channel, the second for red, the third for green, and the fourth for blue. The data for each kernel is a packed array of floating-point values.

kernel_height

The height of the kernel in pixels. This value must be odd.

kernel_width

The width of the kernel in pixels. This value must be odd.

biases

An array of four values to be added to each element of the convolution result for one channel, before clipping. The first value is for the alpha channel, the second for red, the third for green, and the fourth for blue

backgroundColor

A pixel value to be used as a background color. Only used if the `kvImageBackgroundColorFill` flag is set.

flags

The following flags are used:

`kvImageCopyInPlace`

If set, the function will use the Copy In Place technique to access pixels outside the source image.

`kvImageTruncateKernel`

If set, only the values of the kernel that overlap the image will be used.

`kvImageBackgroundColorFill`

If set, the function will use the Background Color Fill technique to access pixels outside the source image.

`kvImageEdgeExtend`

If set, the function will use the Edge Extend technique to access pixels outside the source image.

`kvImageLeaveAlphaUnchanged`

If set, the function will copy the alpha channel to the destination image unchanged.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

The `kvImageCopyInPlace`, `kvImageTruncateKernel`, `kvImageBackgroundColorFill`, and `kvImageEdgeExtend` flags are mutually exclusive. You must set exactly one of these flags. See “[Pixels Outside the Image Buffer](#)” (page 21).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

vImage Functions**Discussion**

In addition to supplying space for the destination image, the dest parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by dest.

For most purposes the kernel should be normalized—that is, the sum of its elements should be 1.0f. A non-normalized kernel will either brighten or darken the image.

vImageConvolveMultiKernel_ARGB8888

This function is a variant of vImageConvolveWithBias_ARGB8888, using a separate kernel, a separate divisor, and a separate bias value for each channel in the interleaved image. For each channel, it performs a convolution of a region of interest within a source image by one of the four M x N kernels, then divides the pixel values by one of the four divisors, placing the result in the destination buffer. After the convolution and before the divisor is applied or any clipping is done, one of the four specified bias value is applied to the result. The image must be in ARGB8888 format. The elements of each kernel must have (non-interleaved) integer values. M and N must both be odd.

```
vImage_Error vImageConvolveMultiKernel_ARGB8888 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    vImagePixelCount srcOffsetToROI_X,
    vImagePixelCount srcOffsetToROI_Y,
    const int16_t *kernels[4],
    uint32_t kernel_height,
    uint32_t kernel_width,
    int32_t divisors[4],
    int32_t biases[4],
    Pixel_8888 backgroundColor,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type vImage_Buffer containing the source image.

dest

A pointer to a structure of type vImage_Buffer. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass NULL, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the kvImageGetTempBufferSize flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the kvImageGetTempBufferSize flag, passing the buffer pointer.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

kernels

An array of four pointers to the data for four kernels. The first kernel is for the alpha channel, the second for red, the third for green, and the fourth for blue. The data for each kernel is a packed array of integer values.

kernel_height

The height of the kernel in pixels. This value must be odd.

kernel_width

The width of the kernel in pixels. This value must be odd.

divisors

An array of four values to divide the results of the convolution for one channel by. This is commonly used for normalization.

biases

An array of four values to be added to each element of the convolution result for one channel, before clipping. The first value is for the alpha channel, the second for red, the third for green, and the fourth for blue

backgroundColor

A pixel value to be used as a background color. Only used if the `kvImageBackgroundColorFill` flag is set.

vImage Functions*flags*

The following flags are used:

kvImageCopyInPlace

If set, the function will use the Copy In Place technique to access pixels outside the source image.

kvImageTruncateKernel

If set, only the values of the kernel that overlap the image will be used.

kvImageBackgroundColorFill

If set, the function will use the Background Color Fill technique to access pixels outside the source image.

kvImageEdgeExtend

If set, the function will use the Edge Extend technique to access pixels outside the source image.

kvImageLeaveAlphaUnchanged

If set, the function will copy the alpha channel to the destination image unchanged.

kvImageDoNotTile

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

The `kvImageCopyInPlace`, `kvImageTruncateKernel`, `kvImageBackgroundColorFill`, and `kvImageEdgeExtend` flags are mutually exclusive. You must set exactly one of these flags. See “[Pixels Outside the Image Buffer](#)” (page 21).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

In addition to supplying space for the destination image, the `dest` parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by `dest`.

Although the pixel values of the source image are in ARGB8888 (integer interleaved) format, the values of the kernel must be (non-interleaved) integer numbers.

The sum of any subset of elements of the kernel should be in the range $-(2^{24})$ to $+(2^{24})-1$. Otherwise integer overflow may occur. If your kernel does not meet this restriction, either use a floating-point format or scale the kernel down to use smaller values.

For most purposes the kernel, together with the divisor, should be normalized. That is, the sum of the kernel’s elements, divided by the divisor, should be 1. A non-normalized kernel will either brighten or darken the image.

vImageBoxConvolve_Planar8

This function is a variant of `vImageConvolve_Planar8`, using an implicit kernel of specified size instead of one specified by the caller (the divisor is also implicit). It performs a convolution of a region of interest within a source image by an M x N kernel that has the effect of a box filter, placing the result in a destination buffer. The image must be in Planar8 format. M and N must both be odd.

vImage Functions

```
vImage_Error vImageBoxConvolve_Planar8 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    vImagePixelCount srcOffsetToROI_X,
    vImagePixelCount srcOffsetToROI_Y,
    uint32_t kernel_height,
    uint32_t kernel_width,
    Pixel_8 backgroundColor,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

kernel_height

The height of the kernel in pixels. This value must be odd.

kernel_width

The width of the kernel in pixels. This value must be odd.

backgroundColor

A pixel value to be used as a background color. Only used if the `kvImageBackgroundColorFill` flag is set.

vImage Functions*flags*

The following flags are used:

`kvImageCopyInPlace`

If set, the function will use the Copy In Place technique to access pixels outside the source image.

`kvImageTruncateKernel`

If set, only the values of the kernel that overlap the image will be used.

`kvImageBackgroundColorFill`

If set, the function will use the Background Color Fill technique to access pixels outside the source image.

`kvImageEdgeExtend`

If set, the function will use the Edge Extend technique to access pixels outside the source image.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

The `kvImageCopyInPlace`, `kvImageTruncateKernel`, `kvImageBackgroundColorFill`, and `kvImageEdgeExtend` flags are mutually exclusive. You must set exactly one of these flags. See “[Pixels Outside the Image Buffer](#)” (page 21).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

In addition to supplying space for the destination image, the `dest` parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by `dest`.

For a discussion of the values in the implicit kernel, please see “[High-Speed Box and Tent Filters](#)” (page 37).

`vImageBoxConvolve_ARGB8888`

This function is a variant of `vImageConvolve_ARGB8888`, using an implicit kernel of specified size instead of one specified by the caller (the divisor is also implicit). It performs a convolution of a region of interest within a source image by an $M \times N$ kernel that has the effect of a box filter, placing the result in a destination buffer. The image must be in ARGB8888 format. M and N must both be odd. The same kernel is used for all channels.

```
vImage_Error vImageBoxConvolve_ARGB8888 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    vImagePixelCount srcOffsetToROI_X,
    vImagePixelCount srcOffsetToROI_Y,
    const int16_t *kernel,
    uint32_t kernel_width,
    Pixel_8888 backgroundColor,
    vImage_Flags flags
```

);

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

kernel_height

The height of the kernel in pixels. This value must be odd.

kernel_width

The width of the kernel in pixels. This value must be odd.

backgroundColor

A pixel value to be used as a background color. Only used if the `kvImageBackgroundColorFill` flag is set.

vImage Functions*flags*

The following flags are used:

kvImageCopyInPlace

If set, the function will use the Copy In Place technique to access pixels outside the source image.

kvImageTruncateKernel

If set, only the values of the kernel that overlap the image will be used.

kvImageBackgroundColorFill

If set, the function will use the Background Color Fill technique to access pixels outside the source image.

kvImageEdgeExtend

If set, the function will use the Edge Extend technique to access pixels outside the source image.

kvImageLeaveAlphaUnchanged

If set, the function will copy the alpha channel to the destination image unchanged.

kvImageDoNotTile

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

The kvImageCopyInPlace, kvImageTruncateKernel, kvImageBackgroundColorFill, and kvImageEdgeExtend flags are mutually exclusive. You must set exactly one of these flags. See “[Pixels Outside the Image Buffer](#)” (page 21).

Return value

If the function succeeds, it returns kvImageNoError. Otherwise it returns an error code.

Discussion

In addition to supplying space for the destination image, the dest parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by dest.

For a discussion of the values in the implicit kernel, please see “[High-Speed Box and Tent Filters](#)” (page 37).

vImageTentConvolve_Planar8

This function is exactly like vImageBoxConvolve_Planar8 except that it provides a tent filter instead of a box filter. It is a variant of vImageConvolve_Planar8, using an implicit kernel of specified size instead of one specified by the caller (the divisor is also implicit). It performs a convolution of a region of interest within a source image by an M x N kernel that has the effect of a tent filter, placing the result in a destination buffer. The image must be in Planar8 format. M and N must both be odd.

```
vImage_Error vImageTentConvolve_Planar8 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    vImagePixelCount srcOffsetToROI_X,
    vImagePixelCount srcOffsetToROI_Y,
```

vImage Functions

```
uint32_t kernel_height,
uint32_t kernel_width,
Pixel_8 backgroundColor,
vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

kernel_height

The height of the kernel in pixels. This value must be odd.

kernel_width

The width of the kernel in pixels. This value must be odd.

backgroundColor

A pixel value to be used as a background color. Only used if the `kvImageBackgroundColorFill` flag is set.

vImage Functions*flags*

The following flags are used:

kvImageCopyInPlace

If set, the function will use the Copy In Place technique to access pixels outside the source image.

kvImageTruncateKernel

If set, only the values of the kernel that overlap the image will be used.

kvImageBackgroundColorFill

If set, the function will use the Background Color Fill technique to access pixels outside the source image.

kvImageEdgeExtend

If set, the function will use the Edge Extend technique to access pixels outside the source image.

kvImageDoNotTile

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

The `kvImageCopyInPlace`, `kvImageTruncateKernel`, `kvImageBackgroundColorFill`, and `kvImageEdgeExtend` flags are mutually exclusive. You must set exactly one of these flags. See “[Pixels Outside the Image Buffer](#)” (page 21).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

In addition to supplying space for the destination image, the `dest` parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by `dest`.

For a discussion of the values in the implicit kernel, please see “[High-Speed Box and Tent Filters](#)” (page 37).

vImageTentConvolve_ARGB8888

This function is exactly like `vImageBoxConvolve_ARGB8888` except that it provides a tent filter instead of a box filter. It is a variant of `vImageConvolve_ARGB8888`, using an implicit kernel of specified size instead of one specified by the caller (the divisor is also implicit). It performs a convolution of a region of interest within a source image by an $M \times N$ kernel that has the effect of a tent filter, placing the result in a destination buffer. The image must be in ARGB8888 format. M and N must both be odd. The same kernel is used for all channels.

```
vImage_Error vImageTentConvolve_ARGB8888 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    vImagePixelCount srcOffsetToROI_X,
    vImagePixelCount srcOffsetToROI_Y,
    const int16_t *kernel,
    uint32_t kernel_width,
    Pixel_8888 backgroundColor,
```

vImage Functions

```
vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

kernel_height

The height of the kernel in pixels. This value must be odd.

kernel_width

The width of the kernel in pixels. This value must be odd.

backgroundColor

A pixel value to be used as a background color. Only used if the `kvImageBackgroundColorFill` flag is set.

vImage Functions*flags*

The following flags are used:

kvImageCopyInPlace

If set, the function will use the Copy In Place technique to access pixels outside the source image.

kvImageTruncateKernel

If set, only the values of the kernel that overlap the image will be used.

kvImageBackgroundColorFill

If set, the function will use the Background Color Fill technique to access pixels outside the source image.

kvImageEdgeExtend

If set, the function will use the Edge Extend technique to access pixels outside the source image.

kvImageLeaveAlphaUnchanged

If set, the function will copy the alpha channel to the destination image unchanged.

kvImageDoNotTile

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

The `kvImageCopyInPlace`, `kvImageTruncateKernel`, `kvImageBackgroundColorFill`, and `kvImageEdgeExtend` flags are mutually exclusive. You must set exactly one of these flags. See “[Pixels Outside the Image Buffer](#)” (page 21).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

In addition to supplying space for the destination image, the `dest` parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by `dest`.

For a discussion of the values in the implicit kernel, please see “[High-Speed Box and Tent Filters](#)” (page 37).

[vImageGetMinimumTempBufferSizeForConvolution](#)

This function is deprecated in Mac OS X v10.4. Please see “[Temporary Buffers for vImage Functions](#)” (page 26). This function returns the minimum size, in bytes, for the temporary buffer that the caller supplies to any of the Convolve functions. (If the caller supplies `NULL`, these functions will allocate their own buffers.)

```
size_t vImageGetMinimumTempBufferSizeForConvolution (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    unsigned int kernel_height,
    unsigned int kernel_width,
    vImage_Flags flags,
    size_t bytesPerPixel
);
```

vImage Functions**Parameters***src*

A pointer to a structure of type `vImage_Buffer` that will be used in the Convolve function.

dest

A pointer to a structure of type `vImage_Buffer` that will be used in the Convolve function.

kernel_height

The height, in pixels, of the kernel that will be used in the Convolve function.

kernel_width

The width, in pixels, of the kernel that will be used in the Convolve function.

flags

The flags that will be passed to the Convolve function.

bytesPerPixel

The number of bytes in one pixel. This depends on the format of the image to be processed:

ARGBFFFF : 16 bytes

ARGB8888: 4 bytes

PlanarF: 4 bytes

Planar8: 1 byte

Return value

The minimum size, in bytes, of the temporary buffer.

Discussion

This function does not depend on the *data* or *rowBytes* fields of the *src* or *dest* parameters; it only uses the *height* and *width* fields from those parameters. If the size of the images you are processing stay the same, then the required size of the buffer will also stay the same. More specifically, if, between two calls to `vImageGetMinimumTempBufferSizeForConvolution`, the *height* and *width* of the *src* and *dest* parameters do not increase, and the other parameters remain the same, then the result of the `vImageGetMinimumTempBufferSizeForConvolution` will not increase. This makes it easy to reuse the same temporary buffer when you are processing a number of images of the same size, as in tiling. See “[Temporary Buffers for vImage Functions](#)” (page 26).

vImageRichardsonLucyDeConvolve_ARGBFFFF

This function can be used to sharpen an image by “undoing” a presumed convolution that has blurred it — such as diffraction effects in a camera lens. See “[Richardson-Lucy Deconvolution Functions](#)” (page 40). It performs a Richardson-Lucy deconvolution of a region of interest within a source image by an M × N kernel, performing a specified number of iterations and placing the result in a destination buffer. The specified kernel expresses the blurring convolution or “point spread function.” A second kernel may also be provided for cases where the first kernel is asymmetrical. The image must be in ARGBFFFF format. The elements of the kernel(s) must have (non-interleaved) floating-point values. M and N must both be odd. The same kernel is used for all channels.

```
vImage_Error vImageRichardsonLucyDeConvolve_ARGBFFFF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    vImagePixelCount srcOffsetToROI_X,
```

vImage Functions

```
vImagePixelCount srcOffsetToROI_Y,
const float *kernel,
const float *kernel2,
uint32_t kernel_height,
uint32_t kernel_width,
uint32_t kernel_height2,
uint32_t kernel_width2,
Pixel_FFFF backgroundColor,
uint32_t iterationCount,
vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

kernel

A pointer to the data of a kernel for the blurring convolution that is to be “undone.” The kernel data is a packed array of floating-point values.

kernel2

A pointer to the data of a second kernel. The kernel data is a packed array of floating-point values. Specify this kernel if the first kernel is asymmetrical; otherwise pass `NULL`.

kernel_height

The height of the first kernel in pixels. This value must be odd.

kernel_width

The width of the first kernel in pixels. This value must be odd.

vImage Functions

kernel_height2

The height of the second kernel in pixels (ignored if *kernel2* is NULL). This value must be odd.

kernel_width2

The width of the second kernel in pixels (ignored if *kernel2* is NULL). This value must be odd.

backgroundColor

A pixel value to be used as a background color. Only used if the `kvImageBackgroundColorFill` flag is set.

iterationCount

The number of times to iterate the deconvolution algorithm.

flags

The following flags are used:

`kvImageCopyInPlace`

If set, the function will use the Copy In Place technique to access pixels outside the source image.

`kvImageTruncateKernel`

If set, only the values of the kernel that overlap the image will be used.

`kvImageBackgroundColorFill`

If set, the function will use the Background Color Fill technique to access pixels outside the source image.

`kvImageEdgeExtend`

If set, the function will use the Edge Extend technique to access pixels outside the source image.

`kvImageLeaveAlphaUnchanged`

If set, the function will copy the alpha channel to the destination image unchanged.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

The `kvImageCopyInPlace`, `kvImageTruncateKernel`, `kvImageBackgroundColorFill`, and `kvImageEdgeExtend` flags are mutually exclusive. You must set exactly one of these flags. See “[Pixels Outside the Image Buffer](#)” (page 21).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

In addition to supplying space for the destination image, the *dest* parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by *dest*.

For further discussion see “[Richardson-Lucy Deconvolution Functions](#)” (page 40).

vImageRichardsonLucyDeConvolve_ARGB8888

This function can be used to sharpen an image by “undoing” a presumed convolution that has blurred it — such as diffraction effects in a camera lens. See “[Richardson-Lucy Deconvolution Functions](#)” (page 40). It performs a Richardson-Lucy deconvolution of a region of interest within a source image by an M x N kernel, performing a specified number of iterations and placing the result in a destination buffer. The specified kernel expresses the blurring convolution or “point spread function.” A second kernel may also be provided for cases where the first kernel is asymmetrical. The image must be in ARGB8888 format. The elements of the kernel(s) must have (non-interleaved) integer values. M and N must both be odd. The same kernel is used for all channels.

```
vImage_Error vImageRichardsonLucyDeConvolve_ARGB8888 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    vImagePixelCount srcOffsetToROI_X,
    vImagePixelCount srcOffsetToROI_Y,
    const int16_t *kernel,
    const int16_t *kernel2,
    uint32_t kernel_height,
    uint32_t kernel_width,
    uint32_t kernel_height2,
    uint32_t kernel_width2,
    int32_t divisor,
    int32_t divisor2,
    Pixel_8888 backgroundColor,
    uint32_t iterationCount,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

kernel1

A pointer to the data of a kernel for the blurring convolution that is to be “undone.” The kernel data is a packed array of integer values.

kernel12

A pointer to the data of a second kernel. The kernel data is a packed array of integer values. Specify this kernel if the first kernel is asymmetrical; otherwise pass NULL.

kernel1_height

The height of the first kernel in pixels. This value must be odd.

kernel1_width

The width of the first kernel in pixels. This value must be odd.

kernel1_height2

The height of the second kernel in pixels (ignored if *kernel12* is NULL). This value must be odd.

kernel1_width2

The width of the second kernel in pixels (ignored if *kernel12* is NULL). This value must be odd.

divisor

The divisor to be used in convolutions with the first kernel.

divisor2

The divisor to be used in convolutions with the second kernel.

backgroundColor

A pixel value to be used as a background color. Only used if the `kvImageBackgroundColorFill` flag is set.

iterationCount

The number of times to iterate the deconvolution algorithm.

vImage Functions*flags*

The following flags are used:

`kvImageCopyInPlace`

If set, the function will use the Copy In Place technique to access pixels outside the source image.

`kvImageTruncateKernel`

If set, only the values of the kernel that overlap the image will be used.

`kvImageBackgroundColorFill`

If set, the function will use the Background Color Fill technique to access pixels outside the source image.

`kvImageEdgeExtend`

If set, the function will use the Edge Extend technique to access pixels outside the source image.

`kvImageLeaveAlphaUnchanged`

If set, the function will copy the alpha channel to the destination image unchanged.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

The `kvImageCopyInPlace`, `kvImageTruncateKernel`, `kvImageBackgroundColorFill`, and `kvImageEdgeExtend` flags are mutually exclusive. You must set exactly one of these flags. See “[Pixels Outside the Image Buffer](#)” (page 21).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

In addition to supplying space for the destination image, the `dest` parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by `dest`.

For further discussion see “[Richardson-Lucy Deconvolution Functions](#)” (page 40).

vImageRichardsonLucyDeConvolve_PlanarF

This function is exactly like `vImageRichardsonLucyDeConvolveARGBFFFF` except that it work on a planar image. It can be used to sharpen an image by “undoing” a presumed convolution that has blurred it — such as diffraction effects in a camera lens. See “[Richardson-Lucy Deconvolution Functions](#)” (page 40). It performs a Richardson-Lucy deconvolution of a region of interest within a source image by an $M \times N$ kernel, performing a specified number of iterations and placing the result in a destination buffer. The specified kernel expresses the blurring convolution or “point spread function.” A second kernel may also be provided for cases where the first kernel is asymmetrical. The image must be in PlanarF format. The elements of the kernel(s) must have floating-point values. M and N must both be odd.

```
vImage_Error vImageRichardsonLucyDeConvolve_PlanarF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
```

vImage Functions

```
void *tempBuffer,
vImagePixelCount srcOffsetToROI_X,
vImagePixelCount srcOffsetToROI_Y,
const float *kernel,
const float *kernel2,
uint32_t kernel_height,
uint32_t kernel_width,
uint32_t kernel_height2,
uint32_t kernel_width2,
Pixel_F backgroundColor,
uint32_t iterationCount,
vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

kernel

A pointer to the data of a kernel for the blurring convolution that is to be “undone.” The kernel data is a packed array of floating-point values.

kernel2

A pointer to the data of a second kernel. The kernel data is a packed array of floating-point values. Specify this kernel if the first kernel is asymmetrical; otherwise pass `NULL`.

kernel_height

The height of the first kernel in pixels. This value must be odd.

kernel_width

The width of the first kernel in pixels. This value must be odd.

vImage Functions*kernel_height2*

The height of the second kernel in pixels (ignored if *kernel2* is NULL). This value must be odd.

kernel_width2

The width of the second kernel in pixels (ignored if *kernel2* is NULL). This value must be odd.

backgroundColor

A pixel value to be used as a background color. Only used if the `kvImageBackgroundColorFill` flag is set.

iterationCount

The number of times to iterate the deconvolution algorithm.

flags

The following flags are used:

`kvImageCopyInPlace`

If set, the function will use the Copy In Place technique to access pixels outside the source image.

`kvImageTruncateKernel`

If set, only the values of the kernel that overlap the image will be used.

`kvImageBackgroundColorFill`

If set, the function will use the Background Color Fill technique to access pixels outside the source image.

`kvImageEdgeExtend`

If set, the function will use the Edge Extend technique to access pixels outside the source image.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

The `kvImageCopyInPlace`, `kvImageTruncateKernel`, `kvImageBackgroundColorFill`, and `kvImageEdgeExtend` flags are mutually exclusive. You must set exactly one of these flags. See “[Pixels Outside the Image Buffer](#)” (page 21).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

In addition to supplying space for the destination image, the *dest* parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by *dest*.

For further discussion see “[Richardson-Lucy Deconvolution Functions](#)” (page 40).

vImageRichardsonLucyDeConvolve_Planar8

This function is exactly like `vImageRichardsonLucyDeConvolveARGB8888` except that it work on a planar image. It can be used to sharpen an image by “undoing” a presumed convolution that has blurred it — such as diffraction effects in a camera lens. See “[Richardson-Lucy Deconvolution Functions](#)” (page 40). It performs a Richardson-Lucy deconvolution of a region of interest within a

vImage Functions

source image by an M x N kernel, performing a specified number of iterations and placing the result in a destination buffer. The specified kernel expresses the blurring convolution or “point spread function.” A second kernel may also be provided for cases where the first kernel is asymmetrical. The image must be in Planar8 format. The elements of the kernel(s) must have floating-point values. M and N must both be odd.

```
vImage_Error vImageRichardsonLucyDeConvolve_Planar8 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    vImagePixelCount srcOffsetToROI_X,
    vImagePixelCount srcOffsetToROI_Y,
    const int16_t *kernel,
    const int16_t *kernel2,
    uint32_t kernel_height,
    uint32_t kernel_width,
    uint32_t kernel_height2,
    uint32_t kernel_width2,
    Pixel_8 backgroundColor,
    uint32_t iterationCount,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

kernel1

A pointer to the data of a kernel for the blurring convolution that is to be “undone.” The kernel data is a packed array of floating-point values.

vImage Functions

kernel12

A pointer to the data of a second kernel. The kernel data is a packed array of floating-point values. Specify this kernel if the first kernel is asymmetrical; otherwise pass NULL.

kernel1_height

The height of the first kernel in pixels. This value must be odd.

kernel1_width

The width of the first kernel in pixels. This value must be odd.

kernel1_height2

The height of the second kernel in pixels (ignored if *kernel12* is NULL). This value must be odd.

kernel1_width2

The width of the second kernel in pixels (ignored if *kernel12* is NULL). This value must be odd.

backgroundColor

A pixel value to be used as a background color. Only used if the `kvImageBackgroundColorFill` flag is set.

iterationCount

The number of times to iterate the deconvolution algorithm.

flags

The following flags are used:

`kvImageCopyInPlace`

If set, the function will use the Copy In Place technique to access pixels outside the source image.

`kvImageTruncateKernel`

If set, only the values of the kernel that overlap the image will be used.

`kvImageBackgroundColorFill`

If set, the function will use the Background Color Fill technique to access pixels outside the source image.

`kvImageEdgeExtend`

If set, the function will use the Edge Extend technique to access pixels outside the source image.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

The `kvImageCopyInPlace`, `kvImageBackgroundColorFill`, and `kvImageEdgeExtend` flags are mutually exclusive. You must set exactly one of these flags. See “[Pixels Outside the Image Buffer](#)” (page 21).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

In addition to supplying space for the destination image, the `dest` parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by `dest`.

For further discussion see “[Richardson-Lucy Deconvolution Functions](#)” (page 40).

Image Transformation Functions: Matrix Multiplication

For conceptual introduction to these functions, please see “[Matrix Multiplication Functions](#)” (page 87).

vImageMatrixMultiply_Planar8

This function operates upon a set of 8-bit source image planes, multiplying each pixel by a specified matrix to produce a set of 8-bit destination image planes.

```
vImage_Error vImageMatrixMultiply_Planar8 (
    const vImage_Buffer *srcs[],
    const vImage_Buffer *dests[],
    uint32_t src_planes,
    uint32_t dest_planes,
    const int16_t matrix[],
    int32_t divisor,
    const int16_t *pre_bias,
    const int32_t *post_bias,
    vImage_Flags flags
);
```

Parameters

srcs

A pointer to an array of pointers to structures of type `vImage_Buffer`, one buffer for each source plane.

dests

A pointer to an array of pointers to structures of type `vImage_Buffer`, one buffer for each destination plane. On return, the buffers contain the planes of the destination image. You must set the fields of these structures yourself, and allocate memory for the array. When you are done with it you must deallocate the memory.

src_planes

The number of source planes as a value of type `uint32_t`.

dest_planes

The number of destination planes as a value of type `uint32_t`.

matrix

A 1-dimensional array of values of type `int16_t`, the values of a matrix whose dimensions are *dest_planes* × *src_planes*. Each source pixel is multiplied by this matrix to obtain a destination pixel

divisor

A value of type `int32_t`, used to normalize the final values after matrix multiplication.

pre_bias

A packed array of values of type `int16_t`, one for each source plane. The source planes are multiplied by these values before matrix multiplication. Pass `NULL` if you do not want a pre-bias.

vImage Functions*post_bias*

A packed array of values of type `int16_t`, one for each destination plane. The source planes are multiplied by these values after matrix multiplication and before the final division. Pass `NULL` if you do not want a post-bias.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

Be aware that 32-bit signed accumulators are used. If the sum over any matrix column is greater than $\pm 2^{23}$, overflow may occur. Generally speaking this will not happen because the matrix elements are 16-bit integers, so it would take more than 256 source planes before trouble could arise.

See the discussion in “[Matrix Multiplication Functions](#)” (page 87) for an example of the application of matrix multiplication.

vImageMatrixMultiply_PlanarF

This function operates upon a set of floating-point source image planes, multiplying each pixel by a specified matrix to produce a set of floating-point destination image planes. (The operation is the same as `vImageMatrixMultiply_Planar8` except that floating-point values are used and there is no divisor.)

```
vImage_Error vImageMatrixMultiply_PlanarF (
    const vImage_Buffer *srcs[],
    const vImage_Buffer *dests[],
    uint32_t src_planes,
    uint32_t dest_planes,
    const float matrix[],
    const float *pre_bias,
    const float *post_bias,
    vImage_Flags flags
);
```

Parameters*srcs*

A pointer to an array of pointers to structures of type `vImage_Buffer`, one buffer for each source plane.

dests

A pointer to an array of pointers to structures of type `vImage_Buffer`, one buffer for each destination plane. On return, the buffers contain the planes of the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

src_planes

The number of source planes as a value of type `uint32_t`.

vImage Functions*dest_planes*

The number of destination planes as a value of type `uint32_t`.

matrix

A 1-dimensional array of values of type `float`, the values of a matrix whose dimensions are `dest_planes x src_planes`. Each source pixel is multiplied by this matrix to obtain a destination pixel

pre_bias

A packed array of values of type `float`, one for each source plane. The source planes are multiplied by these values before matrix multiplication. Pass `NULL` if you do not want a pre-bias.

post_bias

A packed array of values of type `float`, one for each destination plane. The source planes are multiplied by these values after matrix multiplication and before the final division. Pass `NULL` if you do not want a post-bias.

flags

No flags are honored; you should pass 0.

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

See the discussion in “[Matrix Multiplication Functions](#)” (page 87) for an example of the application of matrix multiplication.

vImageMatrixMultiply_ARGB8888

This function operates upon an interleaved 8-bit source image, multiplying each pixel by a specified matrix to produce an interleaved 8-bit destination image.

```
vImage_Error vImageMatrixMultiply_ARGB8888 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    const int16_t matrix[4*4],
    int32_t divisor,
    const int16_t *pre_bias,
    const int32_t *post_bias,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer`.

dest

A pointer to a structure of type `vImage_Buffer`. On return, the buffers contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

matrix

An array of 16 values of type `int16_t`, the values of a 4x4 matrix. Each source pixel is multiplied by this matrix to obtain a destination pixel

vImage Functions*divisor*

A value of type `int32_t`, used to normalize the final values after matrix multiplication.

pre_bias

A packed array of 4 values of type `int16_t`. The pixels are multiplied by these values before matrix multiplication. Pass `NULL` if you do not want a pre-bias.

post_bias

A packed array of 4 values of type `int16_t`. The pixels are multiplied by these values after matrix multiplication and before the final division. Pass `NULL` if you do not want a post-bias.

flags

No flags are honored; you should pass 0.

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

Be aware that 32-bit signed accumulators are used. If the sum over any matrix column is greater than $\pm 2^{23}$, overflow may occur. Generally speaking this will not happen because the matrix elements are 16-bit integers, so it would take more than 256 source planes before trouble could arise.

See the discussion in “[Matrix Multiplication Functions](#)” (page 87) for an example of the application of matrix multiplication.

vImageMatrixMultiply_ARGBFFFF

This function operates upon an interleaved floating-point source image, multiplying each pixel by a specified matrix to produce an interleaved floating-point destination image. (The operation is the same as `vImageMatrixMultiply_ARGB8888` except that floating-point values are used and there is no divisor.)

```
vImage_Error vImageMatrixMultiply_ARGBFFFF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    const float matrix[4*4],
    const float *pre_bias,
    const float *post_bias,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer`.

dest

A pointer to a structure of type `vImage_Buffer`. On return, the buffers contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

matrix

An array of 16 values of type `float`, the values of a 4×4 matrix. Each source pixel is multiplied by this matrix to obtain a destination pixel

vImage Functions*pre_bias*

A packed array of 4 values of type `float`. The pixels are multiplied by these values before matrix multiplication. Pass `NULL` if you do not want a pre-bias.

post_bias

A packed array of 4 values of type `float`. The pixels are multiplied by these values after matrix multiplication and before the final division. Pass `NULL` if you do not want a post-bias.

flags

No flags are honored; you should pass 0.

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

Be aware that 32-bit signed accumulators are used. If the sum over any matrix column is greater than $\pm 2^{23}$, overflow may occur. Generally speaking this will not happen because the matrix elements are 16-bit integers, so it would take more than 256 source planes before trouble could arise.

See the discussion in “[Matrix Multiplication Functions](#)” (page 87) for an example of the application of matrix multiplication.

Image Transformation Functions: Gamma Corrections

For conceptual introduction to these functions, please see “[Gamma Correction](#)” (page 89).

vImageCreateGammaFunction

This function encapsulates a gamma function of a specified type as an opaque object, and returns a pointer to that object. The pointer can be passed to any of the three gamma correction functions, `vImageGamma_Planar8ToPlanarF`, `vImageGamma_PlanarFToPlanar8`, `vImageGamma_PlanarF`. (Note that an 8-bit to 8-bit remapping can be done with `vImageTableLookUp_Planar8` (page 112).)

```
GammaFunction vImageCreateGammaFunction (
    float *gamma,
    int gamma_type,
    vImage_Flags flags
);
```

Parameters*gamma*

A value of type `float`, used as the exponent of a power function when a full-precision gamma correction is calculated. See discussion below.

gamma-type

An `int` value used to select the type of gamma correction to be used. See discussion below.

flags

No flags are honored; you should pass 0.

Return value

A value of type `GammaFunction`.

Discussion

The gamma-type parameter determines the type of calculation to be used. The simplest calculation is:

```
if (value == 0) result = 0;

else {
    if (value < 0)
        sign = -1.0f;
    else
        sign = 1.0f;
    result = pow( fabs( value ), gamma ) * sign;
}
```

This provides for symmetric gamma curves about 0, and makes sure that only well-behaved values are used in `pow()`.

An equivalent calculation can also be done in other, more efficient ways, depending on the desired precision.

In addition to the full-precision gamma correction, there is a faster half-precision option that provides 12-bit precision.

Half precision is intended for use when the data will ultimately be converted to 8-bit integer data. Therefore the half-precision variants only work correctly for floating-point input values in the range 0.0 ... 1.0, though out-of-range values produce results that clamp appropriately to 0 or 255 on conversion back to 8-bit. In addition, there are restrictions on the range of the exponent: it must be positive, in the range 0.1 to 10.0.

Finally, there is a set of still faster half-precision options that use predefined gamma values, ignoring the value set in `vImageCreateGammaFunction`. These options have the same restrictions on input values as stated above.

The constants used to select the type of calculation are:

- `kvImageGamma_UseGammaValue`: Full-precision calculation using the gamma value set in `vImageCreateGammaFunction`.
- `kvImageGamma_UseGammaValue_half_precision`: Half-precision calculation using the gamma value set in `vImageCreateGammaFunction`.
- `kvImageGamma_5_over_9_half_precision`: Half-precision calculation using a gamma value of 5/9 or 1/1.8.
- `kvImageGamma_9_over_5_half_precision`: Half-precision calculation using a gamma value of 9/5 or 1.8.
- `kvImageGamma_5_over_11_half_precision`: Half-precision calculation using a gamma value of 5/11 or 1/2.2.
- `kvImageGamma_11_over_5_half_precision`: Half-precision calculation using a gamma value of 11/5 or 2.2. On exit, gamma is 5/11.
- `kvImageGamma_sRGB_forward_half_precision`: Half-precision calculation using the sRGB standard gamma value of 2.2.
- `kvImageGamma_sRGB_reverse_half_precision`: Half-precision calculation using the sRGB standard gamma value of 1/2.2.

vImage Functions

- `kvImageGamma_11_over_9_half_precision`: Half-precision calculation using a gamma value of 11/9 or (11/5)/(9/5).
- `kvImageGamma_9_over_11_half_precision`: Half-precision calculation using a gamma value of 9/11 or (9/5)/(11/5).
- `kvImageGamma_BT709_forward_half_precision`: ITU-R BT.709 standard. This is like `kvImageGamma_sRGB_forward_half_precision` above but without the 1.125 viewing gamma for computer graphics: $x < 0.081? x / 4.5 : \text{pow}((x + 0.099) / 1.099, 1 / 0.45)$.
- `kvImageGamma_BT709_reverse_half_precision`: ITU-R BT.709 standard reverse. This is like `kvImageGamma_sRGB_reverse_half_precision` above but without the 1.125 viewing gamma for computer graphics: $x < 0.018? 4.5 * x : 1.099 * \text{pow}(x, 0.45) - 0.099$.

vImageDestroyGammaFunction

This function destroys a `GammaFunction` object created with `vImageCreateGammaFunction`.

```
void vImageDestroyGammaFunction ( GammaFunction f )
```

Parameters

f

A gamma function object created with `vImageCreateGammaFunction`.

Return value

`void`.

vImageGamma_Planar8ToPlanarF

This function applies a gamma function to an image in Planar8 format, producing an image in PlanarF format.

```
vImage_Error vImageGamma_Planar8ToPlanarF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    const GammaFunction gamma,
    vImage_Flags flags)
```

Parameters

src

A pointer to a structure of type of type `vImage_Buffer` containing the source image in Planar8 format.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image in PlanarF format. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

gamma

An object of type `GammaFunction`, created with `vImageCreateGammaFunction`.

flags

No flags are honored; you should pass 0.

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

vImageGamma_PlanarFToPlanar8

This function applies a gamma function to an image in PlanarF format, producing an image in Planar8 format.

```
vImage_Error vImageGamma_PlanarFToPlanar8 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    const GammaFunction gamma,
    vImage_Flags flags)
```

Parameters*src*

A pointer to a structure of type of type `vImage_Buffer` containing the source image in PlanarF format.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image in Planar8 format. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

gamma

An object of type `GammaFunction`, created with `vImageCreateGammaFunction`.

flags

No flags are honored; you should pass 0.

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

vImageGamma_PlanarF

This function applies a gamma function to an image in PlanarF format, producing an image in PlanarF format.

```
vImage_Error vImageGamma_PlanarFToPlanar8 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    const GammaFunction gamma,
    vImage_Flags flags)
```

Parameters*src*

A pointer to a structure of type of type `vImage_Buffer` containing the source image in PlanarF format.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image in PlanarF format. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

vImage Functions*gamma*

An object of type `GammaFunction`, created with `vImageCreateGammaFunction`.

flags

No flags are honored; you should pass 0.

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

vImagePiecewisePolynomial_PlanarF

This function applies a set of piecewise polynomials to an image in PlanarF format, producing an image in PlanarF format. By careful choice of polynomials and the ranges of input values they operate on, it is possible to approximate many different correction functions.

```
vImage_Error vImagePiecewisePolynomial_PlanarF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    const float **coefficients,
    const float *boundaries,
    uint32_t order,
    const uint32_t log2segments
    vImage_Flags flags)
```

Parameters*src*

A pointer to a structure of type of type `vImage_Buffer` containing the source image in PlanarF format.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image in PlanarF format. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

coefficients

A pointer to an array of pointers to arrays of floating-point values. Each array of values defines the coefficients of one polynomial. See the discussion below for details.

boundaries

A pointer to an array of floating-point values. Each of these defines a boundary that separates two adjacent ranges of pixel values. See the discussion below for details.

flags

No flags are honored; you should pass 0.

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

Suppose that we want to use N polynomials of order R to process N contiguous ranges of pixel values. N must be a non-negative integer power of 2.

vImage Functions

For each pixel in the image, the range of usable values is divided into segments by the values passed in the *boundaries* array. Each segment will be processed by the corresponding polynomial. Since there are N polynomials, then there must be N segments, so N+1 boundaries are required. Note that in general, the segments will not cover all possible floating-point values; pixel values below the lowest boundary or above the highest are clipped so as to fall within the boundaries.

The boundaries must be ordered by increasing value. The ith segment is defined as the set of pixel values that fall in the range

```
boundary[i] <= value < boundary{i+1}
```

where i ranges from 0 to N. Values in this segment are processed by the i-th polynomial.

The polynomials are defined by the array of arrays of floating-point coefficients. The polynomials must all be of the same order; note that a polynomial of order R has R+1 coefficients. All the coefficient arrays must be the same size, R+1, and in each array the coefficients must be ordered from the 0th-order term to the highest-order term.

Performance note: It costs much more to resolve additional polynomials than to work with higher-order polynomials. You will typically achieve better performance with one 9th-order polynomial that covers the whole range of values you are interested in than with many lower-order polynomials covering the range piecewise.

Accuracy note: Single-precision floating-point arithmetic is used. Polynomials with large high-order coefficients may cause significant rounding error.

vImagePieceWisePolynomial_Planar8ToPlanarF

This function applies a set of piecewise polynomials to an image in Planar8 format, producing an image in PlanarF format. By careful choice of polynomials and the ranges they operate on, it is possible to approximate many different correction functions. Except that the source image is in Planar8 format, its description is identical to [vImagePieceWisePolynomial_PlanarF](#) (page 226).

```
vImage_Error vImagePiecewisePolynomial_PlanarF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    const float **coefficients,
    const float *boundaries,
    uint32_t order,
    const uint32_t log2segments
    vImage_Flags flags)
```

Parameters

src

A pointer to a structure of type of type `vImage_Buffer` containing the source image in Planar8 format.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image in PlanarF format. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

coefficients

A pointer to an array of pointers to arrays of floating-point values. Each array of values defines the coefficients of one polynomial. See the discussion below for details.

vImage Functions*boundaries*

A pointer to an array of floating-point values. Each of these defines a boundary that separates two adjacent ranges of pixel values. See the discussion below for details.

flags

No flags are honored; you should pass 0.

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

Please see the discussion for [vImagePiecewisePolynomial_PlanarF](#) (page 226).

vImagePiecewisePolynomial_PlanarFToPlanar8

This function applies a set of piecewise polynomials to an image in PlanarF format, producing an image in Planar8 format. By careful choice of polynomials and the ranges they operate on, it is possible to approximate many different correction functions. Except that the destination image is in Planar8 format, its description is identical to [vImagePiecewisePolynomial_PlanarF](#) (page 226).

```
vImage_Error vImagePiecewisePolynomial_PlanarFToPlanar8 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    const float **coefficients,
    const float *boundaries,
    uint32_t order,
    const uint32_t log2segments
    vImage_Flags flags)
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image in PlanarF format.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image in Planar8 format. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

coefficients

A pointer to an array of pointers to arrays of floating-point values. Each array of values defines the coefficients of one polynomial. See the discussion below for details.

boundaries

A pointer to an array of floating-point values. Each of these defines a boundary that separates two adjacent ranges of pixel values. See the discussion below for details.

flags

No flags are honored; you should pass 0.

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

Please see the discussion for [vImagePiecewisePolynomial_PlanarF](#) (page 226).

vImageLookupTable_Planar8ToPlanarF

This function applies a lookup table to an image in Planar8 format, producing an image in PlanarF format. The table contains 256 values; it is entered with an 8-bit pixel value from the source image, to look up a floating-point value for the destination image.

```
vImage_Error vImageLookupTable_Planar8ToPlanarF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    const float table[256],
    vImage_Flags flags)
```

Parameters*src*

A pointer to a structure of type of type `vImage_Buffer` containing the source image in Planar8 format.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image in PlanarF format. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

table

An array of 256 values of type `float`.

flags

No flags are honored; you should pass 0.

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

For each pixel, the 8-bit value from the source Planar8 image is used as an index to get a floating-point value from the table. This value is used as the corresponding pixel in the PlanarF result image.

vImageLookupTable_PlanarFToPlanar8

This function applies a lookup table to an image in PlanarF format, producing an image in Planar8 format. The table contains 4096 values; it is entered with an integer index derived from a pixel value in the source image, to look up an 8-bit value for the destination image.

```
vImage_Error vImageLookupTable_PlanarFToPlanar8 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    const float table[4096],
    vImage_Flags flags)
```

Parameters*src*

A pointer to a structure of type of type `vImage_Buffer` containing the source image in PlanarF format.

vImage Functions*dest*

A pointer to a structure of type vImage_Buffer. On return, it contains the destination image in Planar8 format. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

table

An array of 4096 values of type uint8_t.

flags

No flags are honored; you should pass 0.

Return value

If the function succeeds, it returns kvImageNoError. Otherwise it returns an error code.

Discussion

For each pixel, the floating-point value from the source PlanarF image is first clipped to the range 0.0 ... 1.0, and then converted to an integer in the range 0 ... 4095. The conversion calculation is equivalent to

```
if (realValue < 0.0f) realValue = 0.0f;
if (realValue > 1.0f) realValue = 1.0f;
intValue = (int)(realValue * 4095.0f + 0.5f);
```

This integer is used as an index to get an 8-bit value from the table. This value is used as the corresponding pixel in the Planar8 result image.

vImageInterpolatedLookupTable_PlanarF

This function applies a lookup table to an image in PlanarF format, producing an image in PlanarF format. It will work in place. The table contains an arbitrary number of values; it is entered with an index interpolated from a value from the source image, to look up a floating-point value for the destination image.

```
vImage_Error vImageInterpolatedLookupTable_PlanarF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    const float *table,
    vImagePixelCount tableEntries,
    float maxFloat,
    float minFloat
    vImage_Flags flags)
```

Parameters*src*

A pointer to a structure of type vImage_Buffer containing the source image in PlanarF format.

dest

A pointer to a structure of type vImage_Buffer. On return, it contains the destination image in Planar8 format.

table

An array of values of type float.

vImage Functions*tableEntries*

A value of type `vImagePixelCount`, giving the number of values in the array.

maxFloat

A value of type `float`.

minFloat

A value of type `float`.

flags

`kvImageDoNotTile` is honored.

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The input pixel is first clipped to the range `minFloat` ... `maxFloat`. The result is then calculated as

```

float clippedPixel = MAX(MIN(src_pixel, maxFloat), minFloat);
float fIndex = (float) (tableEntries - 1) * (clippedPixel - minFloat)
               / (maxFloat - minFloat);
float fract = fIndex - floor(fIndex);
unsigned long i = fIndex;
float result = table[i] * (1.0f - fract) + table[i + 1] * fract;

```

Histogram Functions

For conceptual introduction to these functions, please see “[Histogram Operations](#)” (page 67).

vImageContrastStretch_ARGBFFFF

This function performs a contrast stretch operation on a source image, placing the result in a destination buffer. The contrast stretch operation alters the image histogram so that pixel values can be found at both the lowest and highest end of the histogram, with values in between “stretched” in a linear fashion. The image must be in ARGBFFFF format.

```

vImage_Error vImageContrastStretch_ARGBFFFF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    unsigned int histogram_entries,
    Pixel_F minValue,
    Pixel_F maxValue,
    vImage_Flags flags
);

```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

vImage Functions

tempBuffer

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

histogram_entries

The number of histogram entries, or “bins,” to be used in histograms for this operation.

minVal

A minimum pixel value, the low end of the histogram. Any pixel value less than this will be clipped to this value (for the purposes of histogram calculation), and assigned to the first histogram entry. This minimum value is applied to each of the four channels separately.

maxVal

A maximum pixel value, the high end of the histogram. Any pixel value greater than this will be clipped to this value (for the purposes of histogram calculation), and assigned to the last histogram entry. This maximum value is applied to each of the four channels separately.

flags

The following flags are used:

`kvImageLeaveAlphaUnchanged`

If set, the function will copy the alpha channel to the destination image unchanged.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The contrast stretch operation is done separately for each of the four channels—alpha, red, green, and blue. However, *histogram_entries*, *minVal*, and *maxVal* are the same for each of the four histograms.

To specify the size and range values of the histograms used in this operation, this function uses the histogram definition parameters *histogram_entries*, *minVal*, and *maxVal*. See [Histogram Definition Parameters For Floating-Point Formats](#).

The source and destination buffers must have the same height and the same width (in pixels).

vImageContrastStretch_ARGB8888

This function performs a contrast stretch operation on a source image, placing the result in a destination buffer. The contrast stretch operation alters the image histogram so that pixel values can be found at both the lowest and highest end of the histogram, with values in between “stretched” in a linear fashion. The image must be in ARGB8888 format.

```
vImage_Error vImageContrastStretch_ARGB8888 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

flags

The following flags are used:

`kvImageLeaveAlphaUnchanged`

If set, the function will copy the alpha channel to the destination image unchanged.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The contrast stretch operation is done separately for each of the four channels—alpha, red, green, and blue.

The source and destination buffers must have the same height and the same width (in pixels).

vImageContrastStretch_PlanarF

This function performs a contrast stretch operation on a source image, placing the result in a destination buffer. The contrast stretch operation alters the image histogram so that pixel values can be found at both the lowest and highest end of the histogram, with values in between “stretched” in a linear fashion. The image must be in PlanarF format.

```
vImage_Error vImageContrastStretch_PlanarF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    unsigned int histogram_entries,
    Pixel_F minValue,
    Pixel_F maxValue,
```

vImage Functions

```
vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

histogram_entries

The number of histogram entries, or “bins,” to be used in histograms for this operation.

minVal

A minimum pixel value, the low end of the histogram. Any pixel value less than this will be clipped to this value (for the purposes of histogram calculation), and assigned to the first histogram entry.

maxVal

A maximum pixel value, the high end of the histogram. Any pixel value greater than this will be clipped to this value (for the purposes of histogram calculation), and assigned to the last histogram entry.

flags

The following flags are used:

`kvImageLeaveAlphaUnchanged`

If set, the function will copy the alpha channel to the destination image unchanged.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

To specify the size and range values of the histograms used in this operation, this function uses the histogram definition parameters `histogram_entries`, `minVal`, and `maxVal`. See [Histogram Definition Parameters For Floating-Point Formats](#).

vImage Functions

The source and destination buffers must have the same height and the same width (in pixels).

vImageContrastStretch_Planar8

This function performs a contrast stretch operation on a source image, placing the result in a destination buffer. The contrast stretch operation alters the image histogram so that pixel values can be found at both the lowest and highest end of the histogram, with values in between “stretched” in a linear fashion. The image must be in Planar8 format.

```
vImage_Error vImageContrastStretch_Planar8 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

flags

The following flags are used:

`kvImageLeaveAlphaUnchanged`

If set, the function will copy the alpha channel to the destination image unchanged.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The source and destination buffers must have the same height and the same width (in pixels).

vImageEndsInContrastStretch_ARGBFFFF

This function performs an ends-in contrast stretch operation on a source image, placing the result in a destination buffer. The ends-in contrast stretch operation alters the image histogram so that a certain percentage of pixels are mapped to the lowest end of the histogram, a certain percentage are mapped to the highest end, and the values in between “stretched” between the lowest and the highest. The image must be in ARGBFFFF format.

```
vImage_Error vImageEndsInContrastStretch_ARGBFFFF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    const unsigned int percent_low[4],
```

vImage Functions

```
const unsigned int percent_high[4],
unsigned int histogram_entries,
Pixel_F minVal,
Pixel_F maxVal,
vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

percent_low

A percentage value. The number of pixels that map to the lowest end of the histogram of the transformed image should represent this percentage of the total pixels.

percent_high

A percentage value. The number of pixels that map to the highest end of the histogram of the transformed image should represent this percentage of the total pixels.

histogram_entries

The number of histogram entries, or “bins,” to be used in histograms for this operation.

minVal

A minimum pixel value, the low end of the histogram. Any pixel value less than this will be clipped to this value (for the purposes of histogram calculation), and assigned to the first histogram entry. This minimum value is applied to each of the four channels separately.

maxVal

A maximum pixel value, the high end of the histogram. Any pixel value greater than this will be clipped to this value (for the purposes of histogram calculation), and assigned to the last histogram entry. This maximum value is applied to each of the four channels separately.

vImage Functions*flags*

The following flags are used:

`kvImageLeaveAlphaUnchanged`

If set, the function will copy the alpha channel to the destination image unchanged.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The ends-in contrast stretch operation is done separately for each of the four channels—alpha, red, green, and blue. However, `histogram_entries`, `minVal`, and `maxVal` are the same for each of the four histograms.

To specify the size and range values of the histograms used in this operation, this function uses the histogram definition parameters `histogram_entries`, `minVal`, and `maxVal`. See [Histogram Definition Parameters For Floating-Point Formats](#).

In general, it is not possible to get exactly the desired percentage of pixels at the low end and the high end of the histogram of the transformed image. This operation only approximates the given values.

The source and destination buffers must have the same height and the same width (in pixels).

vImageEndsInContrastStretch_ARGB8888

This function performs an ends-in contrast stretch operation on a source image, placing the result in a destination buffer. The ends-in contrast stretch operation alters the image histogram so that a certain percentage of pixels are mapped to the lowest end of the histogram, a certain percentage are mapped to the highest end, and the values in between “stretched” between the lowest and the highest. The image must be in ARGB8888 format.

```
vImage_Error vImageEndsInContrastStretch_ARGB8888 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    const unsigned int percent_low[4],
    const unsigned int percent_high[4],
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

vImage Functions*percent_low*

A percentage value. The number of pixels that map to the lowest end of the histogram of the transformed image should represent this percentage of the total pixels.

percent_high

A percentage value. The number of pixels that map to the highest end of the histogram of the transformed image should represent this percentage of the total pixels.

flags

The following flags are used:

kvImageLeaveAlphaUnchanged

If set, the function will copy the alpha channel to the destination image unchanged.

kvImageDoNotTile

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The ends-in contrast stretch operation is done separately for each of the four channels—alpha, red, green, and blue.

In general, it is not possible to get exactly the desired percentage of pixels at the low end and the high end of the histogram of the transformed image. This operation only approximates the given values.

The source and destination buffers must have the same height and the same width (in pixels).

vImageEndsInContrastStretch_PlanarF

This function performs an ends-in contrast stretch operation on a source image, placing the result in a destination buffer. The ends-in contrast stretch operation alters the image histogram so that a certain percentage of pixels are mapped to the lowest end of the histogram, a certain percentage are mapped to the highest end, and the values in between “stretched” between the lowest and the highest. The image must be in PlanarF format.

```
vImage_Error vImageEndsInContrastStretch_PlanarF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    unsigned int percent_low,
    unsigned int percent_high,
    unsigned int histogram_entries,
    Pixel_F minVal,
    Pixel_F maxVal,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

vImage Functions

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

percent_low

A percentage value. The number of pixels that map to the lowest end of the histogram of the transformed image should represent this percentage of the total pixels.

percent_high

A percentage value. The number of pixels that map to the highest end of the histogram of the transformed image should represent this percentage of the total pixels.

histogram_entries

The number of histogram entries, or “bins,” to be used in histograms for this operation.

minVal

A minimum pixel value, the low end of the histogram. Any pixel value less than this will be clipped to this value (for the purposes of histogram calculation), and assigned to the first histogram entry.

maxVal

A maximum pixel value, the high end of the histogram. Any pixel value greater than this will be clipped to this value (for the purposes of histogram calculation), and assigned to the last histogram entry.

flags

The following flags are used:

`kvImageLeaveAlphaUnchanged`

If set, the function will copy the alpha channel to the destination image unchanged.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

To specify the size and range values of the histograms used in this operation, this function uses the histogram definition parameters `histogram_entries`, `minVal`, and `maxVal`. See [Histogram Definition Parameters For Floating-Point Formats](#).

vImage Functions

In general, it is not possible to get exactly the desired percentage of pixels at the low end and the high end of the histogram of the transformed image. This operation only approximates the given values.

The source and destination buffers must have the same height and the same width (in pixels).

vImageEndsInContrastStretch_Planar8

This function performs an ends-in contrast stretch operation on a source image, placing the result in a destination buffer. The ends-in contrast stretch operation alters the image histogram so that a certain percentage of pixels are mapped to the lowest end of the histogram, a certain percentage are mapped to the highest end, and the values in between “stretched” between the lowest and the highest. The image must be in Planar8 format.

```
vImage_Error vImageEndsInContrastStretch_Planar8 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    unsigned int percent_low,
    unsigned int percent_high,
    vImage_Flags flags
);
```

Parameters

src

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

percent_low

A percentage value. The number of pixels that map to the lowest end of the histogram of the transformed image should represent this percentage of the total pixels.

percent_high

A percentage value. The number of pixels that map to the highest end of the histogram of the transformed image should represent this percentage of the total pixels.

flags

The following flags are used:

`kvImageLeaveAlphaUnchanged`

If set, the function will copy the alpha channel to the destination image unchanged.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

In general, it is not possible to get exactly the desired percentage of pixels at the low end and the high end of the histogram of the transformed image. This operation only approximates the given values.

The source and destination buffers must have the same height and the same width (in pixels).

vImageEqualization_ARGBFFFF

This function performs a histogram equalization operation on a source image, placing the result in a destination buffer. The equalization operation alters the image histogram so that it is closer to a uniform intensity distribution. The image must be in ARGBFFFF format.

```
vImage_Error vImageEqualization_ARGBFFFF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    unsigned int histogram_entries,
    Pixel_F minVal,
    Pixel_F maxVal,
    vImage_Flags flags
);
```

Parameters

src

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

histogram_entries

The number of histogram entries, or “bins,” to be used in histograms for this operation.

minVal

A minimum pixel value. Any pixel value less than this will be clipped to this value (for the purposes of histogram calculation), and assigned to the first histogram entry. This minimum value is applied to each of the four channels separately.

maxVal

A maximum pixel value. Any pixel value greater than this will be clipped to this value (for the purposes of histogram calculation), and assigned to the last histogram entry. This maximum value is applied to each of the four channels separately.

vImage Functions*flags*

The following flags are used:

`kvImageLeaveAlphaUnchanged`

If set, the function will copy the alpha channel to the destination image unchanged.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The histogram equalization operation is done separately for each of the four channels—alpha, red, green, and blue. However, `histogram_entries`, `minVal`, and `maxVal` are the same for each of the four histograms.

To specify the size and range values of the histograms used in this operation, this function uses the histogram definition parameters `histogram_entries`, `minVal`, and `maxVal`. See [Histogram Definition Parameters For Floating-Point Formats](#).

The source and destination buffers must have the same height and the same width (in pixels).

vImageEqualization_ARGB8888

This function performs a histogram equalization operation on a source image, placing the result in a destination buffer. The equalization operation alters the image histogram so that it is closer to a uniform intensity distribution. The image must be in ARGB8888 format.

```
vImage_Error vImageEqualization_ARGB8888 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters

src

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

flags

The following flags are used:

`kvImageLeaveAlphaUnchanged`

If set, the function will copy the alpha channel to the destination image unchanged.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

vImage Functions**Return value**

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The histogram equalization operation is done separately for each of the four channels—alpha, red, green, and blue.

The source and destination buffers must have the same height and the same width (in pixels).

`vImageEqualization_PlanarF`

This function performs a histogram equalization operation on a source image, placing the result in a destination buffer. The equalization operation alters the image histogram so that it is closer to a uniform intensity distribution. The image must be in PlanarF format.

```
vImage_Error vImageEqualization_PlanarF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    unsigned int histogram_entries,
    Pixel_F minVal,
    Pixel_F maxVal,
    vImage_Flags flags
);
```

Parameters

src

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

histogram_entries

The number of histogram entries, or “bins,” to be used in histograms for this operation.

minVal

A minimum pixel value. Any pixel value less than this will be clipped to this value (for the purposes of histogram calculation), and assigned to the first histogram entry. This minimum value is applied to each of the four channels separately.

vImage Functions*maxVal*

A maximum pixel value. Any pixel value greater than this will be clipped to this value (for the purposes of histogram calculation), and assigned to the last histogram entry. This maximum value is applied to each of the four channels separately.

flags

The following flags are used:

kvImageLeaveAlphaUnchanged

If set, the function will copy the alpha channel to the destination image unchanged.

kvImageDoNotTile

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

To specify the size and range values of the histograms used in this operation, this function uses the histogram definition parameters `histogram_entries`, `minVal`, and `maxVal`. See [Histogram Definition Parameters For Floating-Point Formats](#).

The source and destination buffers must have the same height and the same width (in pixels).

vImageEqualization_Planar8

This function performs a histogram equalization operation on a source image, placing the result in a destination buffer. The equalization operation alters the image histogram so that it is closer to a uniform intensity distribution. The image must be in ARGB8888 format.

```
vImage_Error vImageEqualization_Planar8 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

vImage Functions*flags*

The following flags are used:

kvImageLeaveAlphaUnchanged

If set, the function will copy the alpha channel to the destination image unchanged.

kvImageDoNotTile

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The source and destination buffers must have the same height and the same width (in pixels).

vImageGetMinimumTempBufferSizeForHistogram

This function is deprecated in Mac OS X v10.4. Please see “[Temporary Buffers for vImage Functions](#)” (page 26). This function returns the minimum size, in bytes, for the temporary buffer that the caller supplies to any of the histogram functions that have a `tempBuffer` parameter. The functions are all the histogram functions that handle images in floating-point formats, excluding the `HistogramCalculation` calls. (If the caller supplies `NULL`, these functions will allocate their own buffers.)

DEPRECATED IN MAC OS X 10.4

```
size_t vImageGetMinimumTempBufferSizeForHistogram (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    unsigned int histogram_entries,
    vImage_Flags flags,
    size_t pixelBytes
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` that will be used in the histogram function.

dest

A pointer to a structure of type `vImage_Buffer` that will be used in the histogram function.

histogram_entries

The number of histogram entries to be used for histograms for the histogram function to be called. This is the same as the value of the `histogram_entries` parameter that will be used in the call.

flags

The flags that will be passed to the histogram function.

vImage Functions*pixelBytes*

The number of bytes in one pixel. This depends on the format of the image to be processed:

ARGBFFFF: 16 bytes

ARGB8888: 4 bytes

PlanarF: 4 bytes

Planar8: 1 byte

Return value

The minimum size, in bytes, of the temporary buffer.

Discussion

This function does not depend on the *data* or *rowBytes* fields of the *src* or *dest* parameters; it only uses the *height* and *width* fields from those parameters. If the size of the images you are processing stay the same, then the required size of the buffer will also stay the same. More specifically, if, between two calls to `vImageGetMinimumTempBufferSizeForHistogram`, the *height* and *width* of the *src* and *dest* parameters do not increase, and the other parameters remain the same, then the result of the `vImageGetMinimumTempBufferSizeForHistogram` will not increase. This makes it easy to reuse the same temporary buffer when you are processing a number of images of the same size, as in tiling. See “[Temporary Buffers for vImage Functions](#)” (page 26).

vImageHistogramCalculation_ARGBFFFF

This function calculates four histograms for an image, one each for the alpha, red, green, and blue channels. The image must be in ARGBFFFF format. The caller specifies a minimum and maximum for pixel values (values outside of these are clipped), and a number of histogram entries (or “bins”).

```
vImage_Error vImageHistogramCalculation_ARGBFFFF (
    const vImage_Buffer *src,
    unsigned int *histogram[4],
    unsigned int histogram_entries,
    Pixel_F minVal,
    Pixel_F maxVal,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

histogram

A pointer to an array of four histograms, one each for alpha, red, green, and blue (in that order). On return, this array will contain the four histograms for the corresponding channels. Each of the four histograms will be an array with *histogram_entries* elements.

histogram_entries

The number of histogram entries, or “bins.” Each of the four calculated histograms will be an array with *histogram_entries* elements.

minVal

A minimum pixel value. Any pixel value less than this will be clipped to this value (for the purposes of histogram calculation), and assigned to the first histogram entry. This minimum value is applied to each of the four channels separately.

vImage Functions*maxVal*

A maximum pixel value. Any pixel value greater than this will be clipped to this value (for the purposes of histogram calculation), and assigned to the last histogram entry. This maximum value is applied to each of the four channels separately.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The function calculates the histogram for each channel completely separately from the others. However, `histogram_entries`, `minVal`, and `maxVal` are the same for each of the four histograms.

To specify the size and range values of the histogram, this function uses the histogram definition parameters `histogram_entries`, `minVal`, and `maxVal`. See [Histogram Definition Parameters For Floating-Point Formats](#).

vImageHistogramCalculation_ARGB8888

This function calculates four histograms for an image, one each for the alpha, red, green, and blue channels. The image must be in ARGB8888 format.

```
vImage_Error vImageHistogramCalculation_ARGB8888 (
    const vImage_Buffer *src,
    unsigned int *histogram[4],
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

histogram

A pointer to a histograms, one each for alpha, red, green, and blue (in that order). On return, this array will contain the four histograms for the corresponding channels. Each of the four histograms will be an array with 256 elements.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The function calculates the histogram for each channel completely separately from the others.

vImageHistogramCalculation_PlanarF

This function calculates a histogram for an image. The image must be in PlanarF format. The caller specifies a minimum and maximum for pixel values (values outside of these are clipped), and a number of histogram entries (or “bins”).

```
vImage_Error vImageHistogramCalculation_PlanarF (
    const vImage_Buffer *src,
    unsigned int *histogram,
    unsigned int histogram_entries,
    Pixel_F minVal,
    Pixel_F maxVal,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

histogram

A pointer to a histogram. On return, this array will contain the histogram for the source image. The histogram will have `histogram_entries` elements.

histogram_entries

The number of histogram entries, or “bins.” The histogram will be an array with `histogram_entries` elements.

minVal

A minimum pixel value. Any pixel value less than this will be clipped to this value (for the purposes of histogram calculation), and assigned to the first histogram entry.

maxVal

A maximum pixel value. Any pixel value greater than this will be clipped to this value (for the purposes of histogram calculation), and assigned to the last histogram entry.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

To specify the size and range values of the histogram, this function uses the histogram definition parameters `histogram_entries`, `minVal`, and `maxVal`. See [Histogram Definition Parameters For Floating-Point Formats](#).

vImageHistogramCalculation_Planar8

This function calculates a histogram for an image. The image must be in Planar8 format.

```
vImage_Error vImageHistogramCalculation_Planar8 (
    const vImage_Buffer *src,
```

vImage Functions

```
unsigned int *histogram,
vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

histogram

A pointer to a histogram. On return, this array will contain the histogram for the source image. The histogram will be an array with 256 elements.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

`vImageHistogramSpecification_ARGBFFFF`

This function performs a histogram specification operation on a source image, placing the result in a destination buffer. The specification operation alters the image histogram so that it more closely resembles a given histogram. The image must be in ARGBFFFF format.

```
vImage_Error vImageHistogramSpecification_ARGBFFFF (
const vImage_Buffer *src,
const vImage_Buffer *dest,
void *tempBuffer,
const unsigned int *desired_histogram[4],
unsigned int histogram_entries,
Pixel_F minValue,
Pixel_F maxValue,
vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

vImage Functions

tempBuffer

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

desiredHistogram

A pointer to an array of four histograms, one each for alpha, red, green, and blue (in that order). These are the desired histograms for the transformed image. Each histogram should be an array with `histogram_entries` elements.

histogram_entries

The number of histogram entries, or “bins,” to be used in histograms for this operation.

minVal

A minimum pixel value. Any pixel value less than this will be clipped to this value (for the purposes of histogram calculation), and assigned to the first histogram entry. This minimum value is applied to each of the four channels separately.

maxVal

A maximum pixel value. Any pixel value greater than this will be clipped to this value (for the purposes of histogram calculation), and assigned to the last histogram entry. This maximum value is applied to each of the four channels separately.

flags

The following flags are used:

`kvImageLeaveAlphaUnchanged`

If set, the function will copy the alpha channel to the destination image unchanged.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The histogram specification operation is done separately for each of the four channels—alpha, red, green, and blue. However, `histogram_entries`, `minVal`, and `maxVal` are the same for each of the four histograms, and for each of the four desired histograms.

To specify the size and range values of the histograms used in this operation, this function uses the histogram definition parameters `histogram_entries`, `minVal`, and `maxVal`. See [Histogram Definition Parameters For Floating-Point Formats](#).

The source and destination buffers must have the same height and the same width (in pixels).

vImageHistogramSpecification_ARGB8888

This function performs a histogram specification operation on a source image, placing the result in a destination buffer. The specification operation alters the image histogram so that it more closely resembles a given histogram. The image must be in ARGB8888 format.

```
vImage_Error vImageHistogramSpecification_ARGB8888 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    const unsigned int *desired_histogram[4],
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

desiredHistogram

A pointer to an array of four histograms, one each for alpha, red, green, and blue (in that order). These are the desired histograms for the transformed image. Each histogram should be an array with 256 elements.

flags

The following flags are used:

`kvImageLeaveAlphaUnchanged`

If set, the function will copy the alpha channel to the destination image unchanged.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The histogram specification operation is done separately for each of the four channels—alpha, red, green, and blue.

The source and destination buffers must have the same height and the same width (in pixels).

vImageHistogramSpecification_PlanarF

This function performs a histogram specification operation on a source image, placing the result in a destination buffer. The specification operation alters the image histogram so that it more closely resembles a given histogram. The image must be in PlanarF format.

```
vImage_Error vImageHistogramSpecification_PlanarF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
```

vImage Functions

```
void *tempBuffer,
const unsigned int *desired_histogram,
unsigned int histogram_entries,
Pixel_F minVal,
Pixel_F maxVal,
vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

desiredHistogram

A pointer to the desired histogram for the transformed image. The histogram should be an array with `histogram_entries` elements.

histogram_entries

The number of histogram entries, or “bins,” to be used in histograms for this operation.

minVal

A minimum pixel value. Any pixel value less than this will be clipped to this value (for the purposes of histogram calculation), and assigned to the first histogram entry. This minimum value is applied to each of the four channels separately.

maxVal

A maximum pixel value. Any pixel value greater than this will be clipped to this value (for the purposes of histogram calculation), and assigned to the last histogram entry. This maximum value is applied to each of the four channels separately.

flags

The following flags are used:

`kvImageLeaveAlphaUnchanged`

If set, the function will copy the alpha channel to the destination image unchanged.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

vImage Functions**Return value**

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

To specify the size and range values of the histograms used in this operation, this function uses the histogram definition parameters `histogram_entries`, `minVal`, and `maxVal`. See [Histogram Definition Parameters For Floating-Point Formats](#). The same histogram definition parameters are used for both the desired histogram and the image histograms.

The source and destination buffers must have the same height and the same width (in pixels).

vImageHistogramSpecification_Planar8

This function performs a histogram specification operation on a source image, placing the result in a destination buffer. The specification operation alters the image histogram so that it more closely resembles a given histogram. The image must be in Planar8 format.

```
vImage_Error vImageHistogramSpecification_Planar8 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    const unsigned int *desired_histogram,
    vImage_Flags flags
);
```

Parameters

src

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

desiredHistogram

A pointer to the desired histogram for the transformed image. The histogram should be an array with 256 elements.

flags

The following flags are used:

`kvImageLeaveAlphaUnchanged`

If set, the function will copy the alpha channel to the destination image unchanged.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see [“Tiling for Cache Utilization”](#) (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The source and destination buffers must have the same height and the same width (in pixels).

Morphological Functions

For conceptual introduction to these functions, please see “[Morphological Operations](#)” (page 43).

vImageDilate_ARGBFFFF

This function performs a dilation of a region of interest within a source image by an M x N kernel, placing the result in a destination buffer. The image must be in ARGBFFFF format. The elements of the kernel must have (non-interleaved) floating-point values. M and N must both be odd. The same kernel is used for all channels.

```
vImage_Error vImageDilate_ARGBFFFF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    unsigned int srcOffsetToROI_X,
    unsigned int srcOffsetToROI_Y,
    const float *kernel,
    unsigned int kernel_height,
    unsigned int kernel_width,
    vImage_Flags flags
);
```

Parameters

src

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

kernel

A pointer to the kernel data. The kernel data is a packed array of `float` values.

kernel_height

The height of the kernel in pixels. This value must be odd.

kernel_width

The width of the kernel in pixels. This value must be odd.

vImage Functions*flags*

The following flags are used:

kvImageLeaveAlphaUnchanged

If set, the function will copy the alpha channel to the destination image unchanged.

kvImageDoNotTile

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

In addition to supplying space for the destination image, the `dest` parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by `dest`.

vImageDilate_ARGB8888

This function performs a dilation of a region of interest within a source image by an $M \times N$ kernel, placing the result in a destination buffer. The image must be in ARGB8888 format. The elements of the kernel must have (non-interleaved) integer values. M and N must both be odd. The same kernel is used for all channels.

```
vImage_Error vImageDilate_ARGB8888 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    unsigned int srcOffsetToROI_X,
    unsigned int srcOffsetToROI_Y,
    const unsigned char *kernel,
    unsigned int kernel_height,
    unsigned int kernel_width,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

vImage Functions*kernel*

A pointer to the kernel data. The kernel data is a packed array of `uint8_t` values.

kernel_height

The height of the kernel in pixels. This value must be odd.

kernel_width

The width of the kernel in pixels. This value must be odd.

flags

The following flags are used:

`kvImageLeaveAlphaUnchanged`

If set, the function will copy the alpha channel to the destination image unchanged.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

In addition to supplying space for the destination image, the `dest` parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by `dest`.

`vImageDilate_PlanarF`

This function performs a dilation of a region of interest within a source image by an $M \times N$ kernel, placing the result in a destination buffer. The image must be in PlanarF format. The elements of the kernel must have floating-point values. M and N must both be odd.

```
vImage_Error vImageDilate_PlanarF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    unsigned int srcOffsetToROI_X,
    unsigned int srcOffsetToROI_Y,
    const float *kernel,
    unsigned int kernel_height,
    unsigned int kernel_width,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

vImage Functions*srcOffsetToROI_X*

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

kernel

A pointer to the kernel data. The kernel data is a packed array of float values.

kernel_height

The height of the kernel in pixels. This value must be odd.

kernel_width

The width of the kernel in pixels. This value must be odd.

flags

The following flags are used:

kvImageLeaveAlphaUnchanged

If set, the function will copy the alpha channel to the destination image unchanged.

kvImageDoNotTile

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

In addition to supplying space for the destination image, the `dest` parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by `dest`.

vImageDilate_Planar8

This function performs a dilation of a region of interest within a source image by an M x N kernel, placing the result in a destination buffer. The image must be in Planar8 format. The elements of the kernel must have integer values. M and N must both be odd.

```
vImage_Error vImageDilate_Planar8 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    unsigned int srcOffsetToROI_X,
    unsigned int srcOffsetToROI_Y,
    const unsigned char *kernel,
    unsigned int kernel_height,
    unsigned int kernel_width,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

vImage Functions*dest*

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

kernel

A pointer to the kernel data. The kernel data is a packed array of `uint8_t` values.

kernel_height

The height of the kernel in pixels. This value must be odd.

kernel_width

The width of the kernel in pixels. This value must be odd.

flags

The following flags are used:

`kvImageLeaveAlphaUnchanged`

If set, the function will copy the alpha channel to the destination image unchanged.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

In addition to supplying space for the destination image, the *dest* parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by *dest*.

`vImageErode_ARGBFFF`

This function performs an erosion of a region of interest within a source image by an M x N kernel, placing the result in a destination buffer. The image must be in ARGBFFFF format. The elements of the kernel must have (non-interleaved) floating-point values. M and N must both be odd. The same kernel is used for all channels.

```
vImage_Error vImageErode_ARGBFFF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    unsigned int srcOffsetToROI_X,
    unsigned int srcOffsetToROI_Y,
    const float *kernel,
    unsigned int kernel_height,
    unsigned int kernel_width,
```

vImage Functions

```
vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

kernel

A pointer to the kernel data. The kernel data is a packed array of `float` values.

kernel_height

The height of the kernel in pixels. This value must be odd.

kernel_width

The width of the kernel in pixels. This value must be odd.

flags

The following flags are used:

`kvImageLeaveAlphaUnchanged`

If set, the function will copy the alpha channel to the destination image unchanged.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

In addition to supplying space for the destination image, the *dest* parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by *dest*.

`vImageErode_ARGB8888`

This function performs an erosion of a region of interest within a source image by an M x N kernel, placing the result in a destination buffer. The image must be in ARGB8888 format. The elements of the kernel must have (non-interleaved) integer values. M and N must both be odd. The same kernel is used for all channels.

```
vImage_Error vImageErode_ARGB8888 (
```

vImage Functions

```
const vImage_Buffer *src,
const vImage_Buffer *dest,
unsigned int srcOffsetToROI_X,
unsigned int srcOffsetToROI_Y,
const unsigned char *kernel,
unsigned int kernel_height,
unsigned int kernel_width,
vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

kernel

A pointer to the kernel data. The kernel data is a packed array of `uint8_t` values.

kernel_height

The height of the kernel in pixels. This value must be odd.

kernel_width

The width of the kernel in pixels. This value must be odd.

flags

The following flags are used:

`kvImageLeaveAlphaUnchanged`

If set, the function will copy the alpha channel to the destination image unchanged.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

In addition to supplying space for the destination image, the *dest* parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by *dest*.

vImageErode_PlanarF

This function performs an erosion of a region of interest within a source image by an M x N kernel, placing the result in a destination buffer. The image must be in PlanarF format. The elements of the kernel must have floating-point values. M and N must both be odd.

```
vImage_Error vImageErode_PlanarF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    unsigned int srcOffsetToROI_X,
    unsigned int srcOffsetToROI_Y,
    const float *kernel,
    unsigned int kernel_height,
    unsigned int kernel_width,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

kernel

A pointer to the kernel data. The kernel data is a packed array of `float` values.

kernel_height

The height of the kernel in pixels. This value must be odd.

kernel_width

The width of the kernel in pixels. This value must be odd.

flags

The following flags are used:

kvImageLeaveAlphaUnchanged

If set, the function will copy the alpha channel to the destination image unchanged.

kvImageDoNotTile

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

In addition to supplying space for the destination image, the *dest* parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by *dest*.

vImageErode_Planar8

This function performs an erosion of a region of interest within a source image by an M x N kernel, placing the result in a destination buffer. The image must be in Planar8 format. The elements of the kernel must have integer values. M and N must both be odd.

```
vImage_Error vImageErode_Planar8 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    unsigned int srcOffsetToROI_X,
    unsigned int srcOffsetToROI_Y,
    const unsigned char *kernel,
    unsigned int kernel_height,
    unsigned int kernel_width,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

kernel

A pointer to the kernel data. The kernel data is a packed array of `uint8_t` values.

kernel_height

The height of the kernel in pixels. This value must be odd.

kernel_width

The width of the kernel in pixels. This value must be odd.

vImage Functions*flags*

The following flags are used:

kvImageLeaveAlphaUnchanged

If set, the function will copy the alpha channel to the destination image unchanged.

kvImageDoNotTile

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

In addition to supplying space for the destination image, the `dest` parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by `dest`.

vImageGetMinimumTempBufferSizeForMinMax

This function is deprecated in Mac OS X v10.4. Please see “[Temporary Buffers for vImage Functions](#)” (page 26). This function returns the minimum size, in bytes, for the temporary buffer that the caller supplies to any of the Min or Max morphological functions. (If the caller supplies `NULL`, these functions will allocate their own buffers.)

DEPRECATED IN MAC OS X 10.4

```
size_t vImageGetMinimumTempBufferSizeForMinMax (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    unsigned int kernel_height,
    unsigned int kernel_width,
    vImage_Flags flags,
    size_t bytesPerPixel
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` that will be used in the Min or Max function.

dest

A pointer to a structure of type `vImage_Buffer` that will be used in the Min or Max function.

kernel_height

The height, in pixels, of the kernel that will be used in the Min or Max function.

kernel_width

The width, in pixels, of the kernel that will be used in the Min or Max function.

flags

The flags that will be passed to the Min or Max function.

vImage Functions*pixelBytes*

The number of bytes in one pixel. This depends on the format of the image to be processed:

ARGBFFFF: 16 bytes

ARGB8888: 4 bytes

PlanarF: 4 bytes

Planar8: 1 byte

Return value

The minimum size, in bytes, of the temporary buffer.

Discussion

This function does not depend on the *data* or *rowBytes* fields of the *src* or *dest* parameters; it only uses the *height* and *width* fields from those parameters. If the size of the images you are processing stay the same, then the required size of the buffer will also stay the same. More specifically, if, between two calls to `vImageGetMinimumTempBufferSizeForMinMax`, the *height* and *width* of the *src* and *dest* parameters do not increase, and the *height* and *width* of the *src* and *dest* parameters remain the same, then the result of the `vImageGetMinimumTempBufferSizeForMinMax` will not increase. This makes it easy to reuse the same temporary buffer when you are processing a number of images of the same size, as in tiling. See “[Temporary Buffers for vImage Functions](#)” (page 26).

vImageMax_ARGBFFFF

This function performs the morphological operation Max with a region of interest within a source image by an M x N kernel, placing the result in a destination buffer. The image must be in ARGBFFFF format. M and N must both be odd.

```
vImage_Error vImageMax_ARGBFFFF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    unsigned int srcOffsetToROI_X,
    unsigned int srcOffsetToROI_Y,
    unsigned int kernel_height,
    unsigned int kernel_width,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

vImage Functions*tempBuffer*

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

kernel_height

The height of the kernel in pixels. This value must be odd.

kernel_width

The width of the kernel in pixels. This value must be odd.

flags

The following flags are used:

`kvImageLeaveAlphaUnchanged`

If set, the function will copy the alpha channel to the destination image unchanged.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The morphological operation Max is a special case of the dilation operation. In the Max operation, all the elements of the kernel have the same value. The actual value does not matter; only the size of the kernel is significant. vImage optimizes this special case, so the Max function is considerably faster than the Dilate function called with a uniform kernel.

In addition to supplying space for the destination image, the `dest` parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by `dest`.

vImageMax_ARGB8888

This function performs the morphological operation Max with a region of interest within a source image by an M x N kernel, placing the result in a destination buffer. The image must be in ARGB8888 format. M and N must both be odd.

vImage Functions

```
vImage_Error vImageMax_ARGB8888 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    unsigned int srcOffsetToROI_X,
    unsigned int srcOffsetToROI_Y,
    unsigned int kernel_height,
    unsigned int kernel_width,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

kernel_height

The height of the kernel in pixels. This value must be odd.

kernel_width

The width of the kernel in pixels. This value must be odd.

flags

The following flags are used:

`kvImageLeaveAlphaUnchanged`

If set, the function will copy the alpha channel to the destination image unchanged.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

vImage Functions**Return value**

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The morphological operation Max is a special case of the dilation operation. In the Max operation, all the elements of the kernel have the same value. The actual value does not matter; only the size of the kernel is significant. vImage optimizes this special case, so the Max function is considerably faster than the Dilate function called with a uniform kernel.

In addition to supplying space for the destination image, the `dest` parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by `dest`.

vImageMax_PlanarF

This function performs the morphological operation Min with a region of interest within a source image by an M x N kernel, placing the result in a destination buffer. The image must be in PlanarF format. M and N must both be odd.

```
vImage_Error vImageMax_PlanarF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    unsigned int srcOffsetToROI_X,
    unsigned int srcOffsetToROI_Y,
    unsigned int kernel_height,
    unsigned int kernel_width,
    vImage_Flags flags
);
```

Parameters

src

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

vImage Functions*srcOffsetToROI_Y*

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

kernel_height

The height of the kernel in pixels. This value must be odd.

kernel_width

The width of the kernel in pixels. This value must be odd.

flags

The following flags are used:

kvImageLeaveAlphaUnchanged

If set, the function will copy the alpha channel to the destination image unchanged.

kvImageDoNotTile

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The morphological operation Max is a special case of the dilation operation. In the Max operation, all the elements of the kernel have the same value. The actual value does not matter; only the size of the kernel is significant. vImage optimizes this special case, so the Max function is considerably faster than the Dilate function called with a uniform kernel.

In addition to supplying space for the destination image, the `dest` parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by `dest`.

vImageMax_Planar8

This function performs the morphological operation Max with a region of interest within a source image by an M x N kernel, placing the result in a destination buffer. The image must be in Planar8 format. M and N must both be odd.

```
vImage_Error vImageMax_Planar8 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    unsigned int srcOffsetToROI_X,
    unsigned int srcOffsetToROI_Y,
    unsigned int kernel_height,
    unsigned int kernel_width,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

vImage Functions

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

kernel_height

The height of the kernel in pixels. This value must be odd.

kernel_width

The width of the kernel in pixels. This value must be odd.

flags

The following flags are used:

`kvImageLeaveAlphaUnchanged`

If set, the function will copy the alpha channel to the destination image unchanged.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The morphological operation Max is a special case of the dilation operation. In the Max operation, all the elements of the kernel have the same value. The actual value does not matter; only the size of the kernel is significant. vImage optimizes this special case, so the Max function is considerably faster than the Dilate function called with a uniform kernel.

In addition to supplying space for the destination image, the *dest* parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by *dest*.

vImageMin_ARGBFFFF

This function performs the morphological operation Min with a region of interest within a source image by an M x N kernel, placing the result in a destination buffer. The image must be in ARGBFFFF format. M and N must both be odd.

```
vImage_Error vImageMin_ARGBFFFF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    unsigned int srcOffsetToROI_X,
    unsigned int srcOffsetToROI_Y,
    unsigned int kernel_height,
    unsigned int kernel_width,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

kernel_height

The height of the kernel in pixels. This value must be odd.

kernel_width

The width of the kernel in pixels. This value must be odd.

vImage Functions*flags*

The following flags are used:

`kvImageLeaveAlphaUnchanged`

If set, the function will copy the alpha channel to the destination image unchanged.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The morphological operation Min is a special case of the erosion operation. In the Min operation, all the elements of the kernel have the same value. The actual value does not matter; only the size of the kernel is significant. vImage optimizes this special case, so the Max function is considerably faster than the Erode function called with a uniform kernel.

In addition to supplying space for the destination image, the `dest` parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by `dest`.

vImageMin_ARGB8888

This function performs the morphological operation Min with a region of interest within a source image by an M x N kernel, placing the result in a destination buffer. The image must be in ARGB8888 format. M and N must both be odd.

```
vImage_Error vImageMin_ARGB8888 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    unsigned int srcOffsetToROI_X,
    unsigned int srcOffsetToROI_Y,
    unsigned int kernel_height,
    unsigned int kernel_width,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

vImage Functions*tempBuffer*

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

kernel_height

The height of the kernel in pixels. This value must be odd.

kernel_width

The width of the kernel in pixels. This value must be odd.

flags

The following flags are used:

`kvImageLeaveAlphaUnchanged`

If set, the function will copy the alpha channel to the destination image unchanged.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The morphological operation Min is a special case of the erosion operation. In the Min operation, all the elements of the kernel have the same value. The actual value does not matter; only the size of the kernel is significant. vImage optimizes this special case, so the Min function is considerably faster than the Erode function called with a uniform kernel.

In addition to supplying space for the destination image, the `dest` parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by `dest`.

vImageMin_PlanarF

This function performs the morphological operation Min with a region of interest within a source image by an $M \times N$ kernel, placing the result in a destination buffer. The image must be in PlanarF format. M and N must both be odd.

vImage Functions

```
vImage_Error vImageMin_PlanarF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    unsigned int srcOffsetToROI_X,
    unsigned int srcOffsetToROI_Y,
    unsigned int kernel_height,
    unsigned int kernel_width,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

kernel_height

The height of the kernel in pixels. This value must be odd.

kernel_width

The width of the kernel in pixels. This value must be odd.

flags

The following flags are used:

`kvImageLeaveAlphaUnchanged`

If set, the function will copy the alpha channel to the destination image unchanged.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

vImage Functions**Return value**

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The morphological operation Min is a special case of the erosion operation. In the Min operation, all the elements of the kernel have the same value. The actual value does not matter; only the size of the kernel is significant. vImage optimizes this special case, so the Min function is considerably faster than the Erode function called with a uniform kernel.

In addition to supplying space for the destination image, the `dest` parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by `dest`.

vImageMin_Planar8

This function performs the morphological operation Min with a region of interest within a source image by an M x N kernel, placing the result in a destination buffer. The image must be in Planar8 format. M and N must both be odd.

```
vImage_Error vImageMin_Planar8 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    unsigned int srcOffsetToROI_X,
    unsigned int srcOffsetToROI_Y,
    unsigned int kernel_height,
    unsigned int kernel_width,
    vImage_Flags flags
);
```

Parameters

src

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

vImage Functions*srcOffsetToROI_Y*

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

kernel_height

The height of the kernel in pixels. This value must be odd.

kernel_width

The width of the kernel in pixels. This value must be odd.

flags

The following flags are used:

kvImageLeaveAlphaUnchanged

If set, the function will copy the alpha channel to the destination image unchanged.

kvImageDoNotTile

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The morphological operation Min is a special case of the erosion operation. In the Min operation, all the elements of the kernel have the same value. The actual value does not matter; only the size of the kernel is significant. vImage optimizes this special case, so the Min function is considerably faster than the Erode function called with a uniform kernel.

In addition to supplying space for the destination image, the `dest` parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by `dest`.

Geometric Functions

For conceptual introduction to these functions, please see “[Geometric Operations](#)” (page 49).

vImageAffineWarp_ARGBFFF

This function performs an affine transform on a source image, placing the result in the destination buffer. The image must be in ARGBFFFF format.

```
vImage_Error vImageAffineWarp_ARGBFFF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    const vImage_AffineTransform *transform,
    Pixel_FFFF backgroundColor,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

transform

The affine transformation matrix.

backgroundColor

A pixel value to be used as a background color. Only used if the `kvImageBackgroundColorFill` flag is set.

flags

The following flags are used:

`kvImageBackgroundColorFill`

If set, the function will use the Background Color Fill technique to access pixels outside the source image.

`kvImageEdgeExtend`

If set, the function will use the Edge Extend technique to access pixels outside the source image.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

`kvImageHighQualityResampling`

If set, the function will use a Lanczos5 kernel for resampling. Otherwise it will use a Lanczos3 kernel.

The `kvImageBackgroundColorFill` and `kvImageEdgeExtend` flags are mutually exclusive. You must set exactly one of these flags. See “[Pixels Outside the Image Buffer](#)” (page 21).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

In an affine transformation, each pixel in the source image, given by its coordinates $[x, y]$, is mapped to a new position $[x', y']$ in the destination image by the formula $[x', y'] = [x, y] \times \text{transform}$, where `transform` is the 3x3 affine transformation matrix. See “[Affine Transformations](#)” (page 51) for more information.

Resampling is done using a Lanczos filter. Either Lanczos3 or Lanczos5 will be used, depending on a flag setting. Using the Lanczos5 kernel will give you slower but more accurate results.

The function uses Background Color Fill to assign values to pixels that are outside the source buffer.

vImageAffineWarp_ARGB8888

This function performs an affine transform on a source image, placing the result in the destination buffer. The image must be in ARGB8888 format.

```
vImage_Error vImageAffineWarp_ARGB8888 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    const vImage_AffineTransform *transform,
    Pixel_8888 backColor,
    vImage_Flags flags
);
```

Parameters

src

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

transform

The affine transformation matrix.

backgroundColor

A pixel value to be used as a background color. Only used if the `kvImageBackgroundColorFill` flag is set.

vImage Functions*flags*

The following flags are used:

kvImageBackgroundColorFill

If set, the function will use the Background Color Fill technique to access pixels outside the source image.

kvImageEdgeExtend

If set, the function will use the Edge Extend technique to access pixels outside the source image.

kvImageDoNotTile

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

kvImageHighQualityResampling

If set, the function will use a Lanczos5 kernel for resampling. Otherwise it will use a Lanczos3 kernel.

The kvImageBackgroundColorFill and kvImageEdgeExtend flags are mutually exclusive. You must set exactly one of these flags. See “[Pixels Outside the Image Buffer](#)” (page 21).

Return value

If the function succeeds, it returns kvImageNoError. Otherwise it returns an error code.

Discussion

In an affine transformation, each pixel in the source image, given by its coordinates $[x, y]$, is mapped to a new position $[x', y']$ in the destination image by the formula $[x', y'] = [x, y] \times \text{transform}$, where transform is the 3x3 affine transformation matrix. See “[Affine Transformations](#)” (page 51) for more information.

Resampling is done using a Lanczos filter. Either Lanczos3 or Lanczos5 will be used, depending on a flag setting. Using the Lanczos5 kernel will give you slower but more accurate results.

The function uses Background Color Fill to assign values to pixels that are outside the source buffer.

vImageAffineWarp_PlanarF

This function performs an affine transform on a source image, placing the result in the destination buffer. The image must be in PlanarF format.

```
vImage_Error vImageAffineWarp_PlanarF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    const vImage_AffineTransform *transform,
    Pixel_F backColor,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type vImage_Buffer containing the source image.

vImage Functions

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

transform

The affine transformation matrix.

backgroundColor

A pixel value to be used as a background color. Only used if the `kvImageBackgroundColorFill` flag is set.

flags

The following flags are used:

`kvImageBackgroundColorFill`

If set, the function will use the Background Color Fill technique to access pixels outside the source image.

`kvImageEdgeExtend`

If set, the function will use the Edge Extend technique to access pixels outside the source image.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

`kvImageHighQualityResampling`

If set, the function will use a Lanczos5 kernel for resampling. Otherwise it will use a Lanczos3 kernel.

The `kvImageBackgroundColorFill` and `kvImageEdgeExtend` flags are mutually exclusive. You must set exactly one of these flags. See “[Pixels Outside the Image Buffer](#)” (page 21).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

In an affine transformation, each pixel in the source image, given by its coordinates $[x, y]$, is mapped to a new position $[x', y']$ in the destination image by the formula $[x', y'] = [x, y] \times \text{transform}$, where `transform` is the 3x3 affine transformation matrix. See “[Affine Transformations](#)” (page 51) for more information.

Resampling is done using a Lanczos filter. Either Lanczos3 or Lanczos5 will be used, depending on a flag setting. Using the Lanczos5 kernel will give you slower but more accurate results.

The function uses Background Color Fill to assign values to pixels that are outside the source buffer.

vImageAffineWarp_Planar8

This function performs an affine transform on a source image, placing the result in the destination buffer. The image must be in Planar8 format.

```
vImage_Error vImageAffineWarp_Planar8 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    const vImage_AffineTransform *transform,
    Pixel_8 backColor,
    vImage_Flags flags
);
```

Parameters

src

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

transform

The affine transformation matrix.

backgroundColor

A pixel value to be used as a background color. Only used if the `kvImageBackgroundColorFill` flag is set.

vImage Functions*flags*

The following flags are used:

`kvImageBackgroundColorFill`

If set, the function will use the Background Color Fill technique to access pixels outside the source image.

`kvImageEdgeExtend`

If set, the function will use the Edge Extend technique to access pixels outside the source image.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

`kvImageHighQualityResampling`

If set, the function will use a Lanczos5 kernel for resampling. Otherwise it will use a Lanczos3 kernel.

The `kvImageBackgroundColorFill` and `kvImageEdgeExtend` flags are mutually exclusive.

You must set exactly one of these flags. See “[Pixels Outside the Image Buffer](#)” (page 21).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

In an affine transformation, each pixel in the source image, given by its coordinates $[x, y]$, is mapped to a new position $[x', y']$ in the destination image by the formula $[x', y'] = [x, y] \times \text{transform}$, where `transform` is the 3x3 affine transformation matrix. See “[Affine Transformations](#)” (page 51) for more information.

Resampling is done using a Lanczos filter. Either Lanczos3 or Lanczos5 will be used, depending on a flag setting. Using the Lanczos5 kernel will give you slower but more accurate results.

The function uses Background Color Fill to assign values to pixels that are outside the source buffer.

vImageDestroyResamplingFilter

This function disposes of a ResamplingFilter object that was created by `vImageNewResamplingFilter`.

```
void vImageDestroyResamplingFilter (
    ResamplingFilter filter
);
```

Parameters*filter*

The ResamplingFilter object to be disposed of.

Return value

none.

Discussion

This function deallocates the memory associated with a ResamplingFilter object that was created by `vImageNewResamplingFilter`. Do not attempt to directly deallocate this memory yourself.

Do not pass this function a ResamplingFilter object created by vImageNewResamplingFilterForFunctionUsingBuffer. You are responsible for deallocating the memory associated with ResamplingFilter objects created by that call yourself.

vImageHorizontalReflect_ARGBFFF

This function reflects a source image left to right across the center vertical line of the image, placing the result in a destination buffer. No scaling or resampling is done. The image must be in ARGBFFFF format.

```
vImage_Error vImageHorizontalReflect_ARGBFFF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters

src

A pointer to a structure of type vImage_Buffer containing the source image.

dest

A pointer to a structure of type vImage_Buffer. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The source and destination buffers must have the same height and the same width (in pixels).

vImageHorizontalReflect_PlanarF

This function reflects a source image left to right across the center vertical line of the image, placing the result in a destination buffer. No scaling or resampling is done. The image must be in PlanarF format.

```
vImage_Error vImageHorizontalReflect_PlanarF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters

src

A pointer to a structure of type vImage_Buffer containing the source image.

vImage Functions*dest*

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The source and destination buffers must have the same height and the same width (in pixels).

`vImageHorizontalReflect_Planar8`

This function reflects a source image left to right across the center vertical line of the image, placing the result in a destination buffer. No scaling or resampling is done. The image must be in PlanarF format.

```
vImage_Error vImageHorizontalReflect_Planar8 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The source and destination buffers must have the same height and the same width (in pixels).

vImageHorizontalReflect_ARGB8888

This function reflects a source image left to right across the center vertical line of the image, placing the result in a destination buffer. No scaling or resampling is done. The image must be in ARGB8888 format.

```
vImage_Error vImageHorizontalReflect_ARGB8888 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The source and destination buffers must have the same height and the same width (in pixels).

vImageHorizontalShear_ARGBFFFF

This function performs a horizontal shear operation on a region of interest of a source image. It also translates and scales the image (both in the horizontal direction). The transformed image is placed in a destination buffer. More than just the region of interest may be transformed; the function transforms as much of the source image as it needs in order to attempt to fill the destination buffer. The image must be in ARGBFFFF format.

```
vImage_Error vImageHorizontalShear_ARGBFFFF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    uint32_t srcOffsetToROI_X,
    uint32_t srcOffsetToROI_Y,
    float xTranslate,
    float shearSlope,
    ResamplingFilter filter,
    Pixel_FFFF backColor,
    vImage_Flags flags
);
```

vImage Functions

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

xTranslate

A translation value for the horizontal direction.

shearSlope

The slope of the front edge of the sheared image, measured in a clockwise direction.

filter

The resampling filter to be used with this function. You create this object by calling `vImageNewResamplingFilter` (to use a default resampling filter supplied by `vImage`) or `vImageNewResamplingFilterForFunctionUsingBuffer` (to use a custom resampling filter that you supply). When the resampling filter is created, you can also set a scale factor that will be used in the horizontal shear operation.

backgroundColor

A pixel value to be used as a background color. Only used if the `kvImageBackgroundColorFill` is set.

flags

The following flags are used:

`kvImageBackgroundColorFill`

If set, the function will use the Background Color Fill technique to handle pixels outside the source image.

`kvImageEdgeExtend`

If set, the function will use the Edge Extend technique to handle pixels outside the source image.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

`kvImageHighQualityResampling`

Set this flag to the same value used when you create the `filter` parameter.

The `kvImageBackgroundColorFill` and `kvImageEdgeExtend` flags are mutually exclusive. You must set exactly one of these flags. See “[Pixels Outside the Image Buffer](#)” (page 21).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

In addition to supplying space for the destination image, the `dest` parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by `dest`.

vImageHorizontalShear_ARGB8888

This function performs a horizontal shear operation on a region of interest of a source image. It also translates and scales the image (both in the horizontal direction). The transformed image is placed in a destination buffer. More than just the region of interest may be transformed; the function transforms as much of the source image as it needs in order to attempt to fill the destination buffer. The image must be in ARGB8888 format.

```
vImage_Error vImageHorizontalShear_ARGB8888 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    uint32_t srcOffsetToROI_X,
    uint32_t srcOffsetToROI_Y,
    float xTranslate, float shearSlope,
    ResamplingFilter filter,
    Pixel_8888 backColor,
    vImage_Flags flags
);
```

Parameters

src

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

xTranslate

A translation value for the horizontal direction.

shearSlope

The slope of the front edge of the sheared image, measured in a clockwise direction.

vImage Functions*filter*

The resampling filter to be used with this function. You create this object by calling `vImageNewResamplingFilter` (to use a default resampling filter supplied by vImage) or `vImageNewResamplingFilterForFunctionUsingBuffer` (to use a custom resampling filter that you supply). When the resampling filter is created, you can also set a scale factor that will be used in the horizontal shear operation.

backgroundColor

A pixel value to be used as a background color. Only used if the `kvImageBackgroundColorFill` is set.

flags

The following flags are used: Set these flags to the same value used when you create the filter parameter.

`kvImageBackgroundColorFill`

If set, the function will use the Background Color Fill technique to handle pixels outside the source image.

`kvImageEdgeExtend`

If set, the function will use the Edge Extend technique to handle pixels outside the source image.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

`kvImageHighQualityResampling`

Set this flag to the same value used when you create the `filter` parameter.

The `kvImageBackgroundColorFill` and `kvImageEdgeExtend` flags are mutually exclusive. You must set exactly one of these flags. See “[Pixels Outside the Image Buffer](#)” (page 21).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

In addition to supplying space for the destination image, the `dest` parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by `dest`.

`vImageHorizontalShear_PlanarF`

This function performs a horizontal shear operation on a region of interest of a source image. It also translates and scales the image (both in the horizontal direction). The transformed image is placed in a destination buffer. More than just the region of interest may be transformed; the function transforms as much of the source image as it needs in order to attempt to fill the destination buffer. The image must be in PlanarF format.

```
vImage_Error vImageHorizontalShear_PlanarF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    uint32_t srcOffsetToROI_X,
    uint32_t srcOffsetToROI_Y,
    float xTranslate,
```

vImage Functions

```
float shearSlope,
ResamplingFilter filter,
Pixel_F backColor,
vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

xTranslate

A translation value for the horizontal direction.

shearSlope

The slope of the front edge of the sheared image, measured in a clockwise direction.

filter

The resampling filter to be used with this function. You create this object by calling `vImageNewResamplingFilter` (to use a default resampling filter supplied by `vImage`) or `vImageNewResamplingFilterForFunctionUsingBuffer` (to use a custom resampling filter that you supply). When the resampling filter is created, you can also set a scale factor that will be used in the horizontal shear operation.

backgroundColor

A pixel value to be used as a background color. Only used if the `kvImageBackgroundColorFill` is set.

vImage Functions*flags*

The following flags are used:

kvImageBackgroundColorFill

If set, the function will use the Background Color Fill technique to handle pixels outside the source image. Set this flag to the same value used when you create the filter parameter.

kvImageEdgeExtend

If set, the function will use the Edge Extend technique to handle pixels outside the source image. Set this flag to the same value used when you create the filter parameter.

kvImageDoNotTile

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

kvImageHighQualityResampling

Set this flag to the same value used when you create the *filter* parameter.

The kvImageBackgroundColorFill and kvImageEdgeExtend flags are mutually exclusive. You must set exactly one of these flags. See “[Pixels Outside the Image Buffer](#)” (page 21).

Return value

If the function succeeds, it returns kvImageNoError. Otherwise it returns an error code.

Discussion

In addition to supplying space for the destination image, the *dest* parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by *dest*.

vImageHorizontalShear_Planar8

This function performs a horizontal shear operation on a region of interest of a source image. It also translates and scales the image (both in the horizontal direction). The transformed image is placed in a destination buffer. More than just the region of interest may be transformed; the function transforms as much of the source image as it needs in order to attempt to fill the destination buffer. The image must be in Planar8 format.

```
vImage_Error vImageHorizontalShear_Planar8 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    uint32_t srcOffsetToROI_X,
    uint32_t srcOffsetToROI_Y,
    float xTranslate,
    float shearSlope,
    ResamplingFilter filter,
    Pixel_8 backColor,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type vImage_Buffer containing the source image.

vImage Functions

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

xTranslate

A translation value for the horizontal direction.

shearSlope

The slope of the front edge of the sheared image, measured in a clockwise direction.

filter

The resampling filter to be used with this function. You create this object by calling `vImageNewResamplingFilter` (to use a default resampling filter supplied by vImage) or `vImageNewResamplingFilterForFunctionUsingBuffer` (to use a custom resampling filter that you supply). When the resampling filter is created, you can also set a scale factor that will be used in the horizontal shear operation.

backgroundColor

A pixel value to be used as a background color. Only used if the `kvImageBackgroundColorFill` is set.

flags

The following flags are used:

`kvImageBackgroundColorFill`

If set, the function will use the Background Color Fill technique to handle pixels outside the source image.

`kvImageEdgeExtend`

If set, the function will use the Edge Extend technique to handle pixels outside the source image.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

`kvImageHighQualityResampling`

Set this flag to the same value used when you create the filter parameter.

The `kvImageBackgroundColorFill` and `kvImageEdgeExtend` flags are mutually exclusive. Do not set more than one; if you do, you will get a `kvImageInvalidEdgeStyle` error. You must set exactly one of these flags. See “[Pixels Outside the Image Buffer](#)” (page 21).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

vImage Functions**Discussion**

In addition to supplying space for the destination image, the *dest* parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by *dest*.

vImageGetMinimumGeometryTempBufferSize

This function is deprecated in Mac OS X v10.4. Please see “[Temporary Buffers for vImage Functions](#)” (page 26). This function returns the minimum size, in bytes, for the temporary buffer that the caller supplies to any of the high-level geometry functions Rotate, Scale, or AffineWarp. (If the caller supplies `NULL`, these functions will allocate their own buffers.)

DEPRECATED IN MAC OS X 10.4

```
size_t vImageGetMinimumGeometryTempBufferSize (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    vImage_Flags flags,
    size_t bytesPerPixel
);
```

Parameters

src

A pointer to a structure of type `vImage_Buffer` that will be used in the geometry function.

dest

A pointer to a structure of type `vImage_Buffer` that will be used in the histogram function. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

flags

The flags that will be passed to the geometry function.

bytesPerPixel

The number of bytes in one pixel. This depends on the format of the image to be processed:

ARGBFFFF: 16 bytes

ARGB8888: 4 bytes

PlanarF: 4 bytes

Planar8: 1 byte

Return value

The minimum size, in bytes, of the temporary buffer.

Discussion

This function does not depend on the *data* or *rowBytes* fields of the *src* or *dest* parameters; it only uses the *height* and *width* fields from those parameters. If the size of the images you are processing stay the same, then the required size of the buffer will also stay the same. More specifically, if, between two calls to `vImageGetMinimumGeometryTempBufferSize`, the *height* and *width* of the *src* and *dest* parameters do not increase, and the other parameters remain the same, then the result of the `vImageGetMinimumGeometryTempBufferSize` will not increase. This makes it easy to reuse the same temporary buffer when you are processing a number of images of the same size, as in tiling. See “[Temporary Buffers for vImage Functions](#)” (page 26).

vImageGetResamplingFilterSize

This function returns the minimum size, in bytes, for the buffer that you must pass to vImageNewResamplingFilterForFunctionUsingBuffer.

```
size_t vImageGetResamplingFilterSize (
    float scale,
    void (*kernelFunc)(
        const float *xArray,
        float *yArray,
        unsigned int count,
        void *userData
    ),
    float kernelWidth,
    vImage_Flags flags
);
```

Parameters*scale*

The scale factor that will be passed to vImageNewResamplingFilterForFunctionUsingBuffer.

kernelFunc

The function pointer that will be passed to vImageNewResamplingFilterForFunctionUsingBuffer.

userData

The user data pointer that will be passed to vImageNewResamplingFilterForFunctionUsingBuffer.

kernelWidth

The kernel width that will be passed to vImageNewResamplingFilterForFunctionUsingBuffer.

flags

The kernel width that will be passed to vImageNewResamplingFilterForFunctionUsingBuffer.

Return value

The minimum size, in bytes, of the buffer.

vImageNewResamplingFilter

This function creates a new ResamplingFilter object corresponding to a default kernel supplied by vImage—currently a Lanczos filter. This object can be passed to a Shear function. This ResampleFilter object also contains a scale factor, which will be used in the Shear operation.

```
ResamplingFilter vImageNewResamplingFilter (
    float scale,
    vImage_Flags flags
);
```

Parameters*scale*

A scale factor to be used with a Shear operation.

vImage Functions*flags*

The following flags are used:

kvImageBackgroundColorFill

If set, the function will use the Background Color Fill technique to handle pixels outside the source image. Set this flag to the same value used when you call the Shear function.

kvImageEdgeExtend

If set, the function will use the Edge Extend technique to handle pixels outside the source image. Set this flag to the same value used when you call the Shear function.

kvImageHighQualityResampling

If set, the ResamplingFilter will use a Lanczos5 kernel. Otherwise it will use a Lanczos3 kernel. Set this flag to the same value used when you call the Shear function.

The `kvImageBackgroundColorFill` and `kvImageEdgeExtend` flags are mutually exclusive. You must set exactly one of these flags. See “[Pixels Outside the Image Buffer](#)” (page 21).

Return value

A pointer to a new ResamplingFilter object. If the function fails, NULL will be returned.

Discussion

This function will allocate the memory needed for the ResamplingFilter object. You do not need to pass in a buffer. To deallocate this memory, call `vImageDestroyResamplingFilter`. Do not attempt to deallocate the memory yourself.

`vImageNewResamplingFilterForFunctionUsingBuffer`

This function creates a ResamplingFilter object that encapsulates a resampling kernel function that you provide. This object is placed in a buffer that you provide. The object can be passed to a Shear function. The object also contains a scale factor, which will be used in the Shear operation.

```
vImage_Error vImageNewResamplingFilterForFunctionUsingBuffer (
    ResamplingFilter filter,
    float scale,
    void (*kernelFunc)
    (
        const float *xArray,
        float *yArray,
        unsigned int count,
        void *userData
    ),
    float kernelWidth,
    void *userData,
    vImage_Flags flags
);
```

Parameters*filter*

A pointer to a buffer. On return, it will contain the ResamplingFilter object. You must allocate the memory for this buffer yourself. Call `vImageGetResamplingFilterSize` to get the size of this buffer.

scale

A scale factor to be used with a Shear operation.

vImage Functions*kernelFunc*

A pointer to your custom resampling kernel function. If this pointer is `NULL`, vImage will create a `ResamplingFilter` object corresponding to a Lanczos filter.

kernelWidth

A bounding value for the domain of your resampling kernel function. When your function is called, the x-values it will be passed will lie between `-kernelWidth` and `+kernelWidth`, inclusive.

userData

A pointer to custom data that you want to use when calculating your resampling kernel function. When your resampling kernel function is called, this pointer will be passed to it. The data can be anything you wish. If you do not use user data, pass `NULL`.

flags

The following flags are used:

`kvImageBackgroundColorFill`

If set, the function will use the Background Color Fill technique to handle pixels outside the source image. Set this flag to the same value used when you call the `Shear` function.

`kvImageEdgeExtend`

If set, the function will use the Edge Extend technique to handle pixels outside the source image. Set this flag to the same value used when you call the `Shear` function.

`kvImageHighQualityResampling`

If set, and the `kernelFunc` parameter is `NULL`, the `ResamplingFilter` will use a Lanczos5 kernel. If not set, and the `kernelFunc` is `NULL`, it will use a Lanczos3 kernel. Set this flag to the same value used when you call the `Shear` function.

The `kvImageBackgroundColorFill` and `kvImageEdgeExtend` flags are mutually exclusive. You must set exactly one of these flags. See “[Pixels Outside the Image Buffer](#)” (page 21).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

You must allocate a buffer to contain the `ResamplingFilter` object returned by this function. When you are done with this object, you are responsible for deallocating its memory. Do not call `vImageDestroyResamplingFilter` on it.

The `userData` can point to any data you want to use in calculating your resampling kernel function—a table, a list of pointers to related functions, or so on. The data must remain valid for the life of the `ResamplingKernel` object.

The function pointed to by `kernelFunc` must also remain valid for the life of the `ResamplingFilter` object.

If you pass `NULL` for `kernelFunc`, `vImageNewResamplingFilterForFunctionUsingBuffer` will return a `ResamplingFilter` object corresponding to a Lanczos filter—either Lanczos3 or Lanczos5, depending on a flag setting. Usually, if you wanted to use a Lanczos filter, you would call `vImageNewResamplingFilter` instead. However, by calling `vImageNewResamplingFilterForFunctionUsingBuffer`, you gain control over the memory used for the object. You can determine where in memory it is located, you can reuse the buffer to cut down

vImage Functions

on memory allocation, and so on. Note that a `ResamplingFilter` object created in this way is not necessarily the same as one created by `vImageNewResamplingFilter`. You still need to deallocate the object yourself; you can not pass it to `vImageDestroyResamplingFilter`.

`vImageRotate90_ARGBFFFF`

This function rotates a source image by either 0, 90, 180, or 270 degrees (depending on a value supplied by the caller), placing the result in a destination buffer. The center point of the source image is mapped to the center point of the destination image. No scaling or resampling is done. The image must be in ARGBFFFF format.

```
vImage_Error vImageRotate90_ARGBFFFF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    uint8_t rotationConstant,
    Pixel_FFFF backgroundColor,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

rotationConstant

A value specifying the angle of rotation. See “[Rotation Constants for Use With Rotate90 Function](#)” (page 329).

backgroundColor

A pixel value to be used as a background color.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

Because no resampling is done—instead, individual pixels are copied unchanged to new locations—this function places certain restrictions on the pixel height and widths of the source and destination buffers, so that it can map the center of the source to the center of the destination precisely. The restrictions are these:

If you are rotating 90 or 270 degrees, the height of the source image and the width of the destination image must both be even or both be odd; and the width of the source image and the height of the destination image must both be even or both be odd.

vImage Functions

If your images do not meet these restrictions, you can use the general (high-level) Rotate function instead, with an angle of 90 or 270 degrees.

Depending on the relative sizes of the source image and the destination buffer, parts of the source image may be clipped, and areas outside the source image (colored with a caller-supplied background color) may appear in the destination image.

vImageRotate90_ARGB8888

This function rotates a source image by either 0, 90, 180, or 270 degrees (depending on a value supplied by the caller), placing the result in a destination buffer. The center point of the source image is mapped to the center point of the destination image. No scaling or resampling is done. The image must be in ARGB8888 format.

```
vImage_Error vImageRotate90_ARGB8888 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    uint8_t rotationConstant,
    Pixel_8888 backgroundColor,
    vImage_Flags flags
);
```

Parameters

src

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

rotationConstant

A value specifying the angle of rotation. See “[Rotation Constants for Use With Rotate90 Function](#)” (page 329).

backgroundColor

A pixel value to be used as a background color.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

Because no resampling is done—instead, individual pixels are copied unchanged to new locations—this function places certain restrictions on the pixel height and widths of the source and destination buffers, so that it can map the center of the source to the center of the destination precisely. The restrictions are these:

vImage Functions

If you are rotating 90 or 270 degrees, the height of the source image and the width of the destination image must both be even or both be odd; and the width of the source image and the height of the destination image must both be even or both be odd.

If your images do not meet these restrictions, you can use the general (high-level) Rotate function instead, with an angle of 90 or 270 degrees.

Depending on the relative sizes of the source image and the destination buffer, parts of the source image may be clipped, and areas outside the source image (colored with a caller-supplied background color) may appear in the destination image.

vImageRotate90_PlanarF

This function rotates a source image by either 0, 90, 180, or 270 degrees (depending on a value supplied by the caller), placing the result in a destination buffer. The center point of the source image is mapped to the center point of the destination image. No scaling or resampling is done. The image must be in PlanarF format.

```
vImage_Error vImageRotate90_PlanarF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    uint8_t rotationConstant,
    Pixel_F backgroundColor,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

rotationConstant

A value specifying the angle of rotation. See “[Rotation Constants for Use With Rotate90 Function](#)” (page 329).

backgroundColor

A pixel value to be used as a background color.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

Because no resampling is done—instead, individual pixels are copied unchanged to new locations—this function places certain restrictions on the pixel height and widths of the source and destination buffers, so that it can map the center of the source to the center of the destination precisely. The restrictions are these:

If you are rotating 90 or 270 degrees, the height of the source image and the width of the destination image must both be even or both be odd; and the width of the source image and the height of the destination image must both be even or both be odd.

If your images do not meet these restrictions, you can use the general (high-level) Rotate function instead, with an angle of 90 or 270 degrees.

Depending on the relative sizes of the source image and the destination buffer, parts of the source image may be clipped, and areas outside the source image (colored with a caller-supplied background color) may appear in the destination image.

vImageRotate90_Planar8

This function rotates a source image by either 0, 90, 180, or 270 degrees (depending on a value supplied by the caller), placing the result in a destination buffer. The center point of the source image is mapped to the center point of the destination image. No scaling or resampling is done. The image must be in Planar8 format.

```
vImage_Error vImageRotate90_Planar8 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    uint8_t rotationConstant,
    Pixel_8 backColor,
    vImage_Flags flags
);
```

Parameters

src

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

rotationConstant

A value specifying the angle of rotation. See “[Rotation Constants for Use With Rotate90 Function](#)” (page 329).

backgroundColor

A pixel value to be used as a background color.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

vImage Functions**Return value**

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

Because no resampling is done—instead, individual pixels are copied unchanged to new locations—this function places certain restrictions on the pixel height and widths of the source and destination buffers, so that it can map the center of the source to the center of the destination precisely. The restrictions are these:

If you are rotating 90 or 270 degrees, the height of the source image and the width of the destination image must both be even or both be odd; and the width of the source image and the height of the destination image must both be even or both be odd.

If your images do not meet these restrictions, you can use the general (high-level) Rotate function instead, with an angle of 90 or 270 degrees.

Depending on the relative sizes of the source image and the destination buffer, parts of the source image may be clipped, and areas outside the source image (colored with a caller-supplied background color) may appear in the destination image.

`vImageRotate_ARGBFFF`

This function rotates a source image by a given angle, placing the result in a destination buffer. The center point of the source image is mapped to the center point of the destination image. No scaling is done. The image must be in ARGBFFFF format.

```
vImage_Error vImageRotate_ARGBFFF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    float angleInRadians,
    Pixel_FFFF backgroundColor,
    vImage_Flags flags
);
```

Parameters

src

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

vImage Functions

tempBuffer

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

angleInRadians

The angle of rotation, in radians.

backgroundColor

A pixel value to be used as a background color. Only used if the `kvImageBackgroundColorFill` flag is set.

flags

The following flags are used:

`kvImageBackgroundColorFill`

If set, the function will use the Background Color Fill technique to access pixels outside the source image.

`kvImageEdgeExtend`

If set, the function will use the Edge Extend technique to access pixels outside the source image.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

`kvImageHighQualityResampling`

If set, the function will use a Lanczos5 kernel for resampling. Otherwise it will use a Lanczos3 kernel.

The `kvImageBackgroundColorFill` and `kvImageEdgeExtend` flags are mutually exclusive. You must set exactly one of these flags. See “[Pixels Outside the Image Buffer](#)” (page 21).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

Depending on the relative sizes of the source image and destination image, parts of the source image may be clipped, and areas outside the source image may appear in the destination image.

Resampling is done using a Lanczos filter. Either Lanczos3 or Lanczos5 will be used, depending on a flag setting. Using the Lanczos5 kernel will give you slower but more accurate results.

vImageRotate_ARGB8888

This function rotates a source image by a given angle, placing the result in a destination buffer. The center point of the source image is mapped to the center point of the destination image. No scaling is done. The image must be in ARGB8888 format.

```
vImage_Error vImageRotate_ARGB8888 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    float angleInRadians,
    Pixel_8888 backgroundColor,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

angleInRadians

The angle of rotation, in radians.

backgroundColor

A pixel value to be used as a background color. Only used if the `kvImageBackgroundColorFill` flag is set.

vImage Functions*flags*

The following flags are used:

kvImageBackgroundColorFill

If set, the function will use the Background Color Fill technique to access pixels outside the source image.

kvImageEdgeExtend

If set, the function will use the Edge Extend technique to access pixels outside the source image.

kvImageDoNotTile

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

kvImageHighQualityResampling

If set, the function will use a Lanczos5 kernel for resampling. Otherwise it will use a Lanczos3 kernel.

The kvImageBackgroundColorFill and kvImageEdgeExtend flags are mutually exclusive. You must set exactly one of these flags. See “[Pixels Outside the Image Buffer](#)” (page 21).

Return value

If the function succeeds, it returns kvImageNoError. Otherwise it returns an error code.

Discussion

Depending on the relative sizes of the source image and destination image, parts of the source image may be clipped, and areas outside the source image may appear in the destination image.

Resampling is done using a Lanczos filter. Either Lanczos3 or Lanczos5 will be used, depending on a flag setting. Using the Lanczos5 kernel will give you slower but more accurate results.

vImageRotate_PlanarF

This function rotates a source image by a given angle, placing the result in a destination buffer. The center point of the source image is mapped to the center point of the destination image. No scaling is done. The image must be in PlanarF format.

```
vImage_Error vImageRotate_PlanarF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    float angleInRadians,
    Pixel_F backgroundColor,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type vImage_Buffer containing the source image.

dest

A pointer to a structure of type vImage_Buffer. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

vImage Functions

tempBuffer

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

angleInRadians

The angle of rotation, in radians.

backgroundColor

A pixel value to be used as a background color. Only used if the `kvImageBackgroundColorFill` flag is set.

flags

The following flags are used:

`kvImageBackgroundColorFill`

If set, the function will use the Background Color Fill technique to access pixels outside the source image.

`kvImageEdgeExtend`

If set, the function will use the Edge Extend technique to access pixels outside the source image.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

`kvImageHighQualityResampling`

If set, the function will use a Lanczos5 kernel for resampling. Otherwise it will use a Lanczos3 kernel.

The `kvImageBackgroundColorFill` and `kvImageEdgeExtend` flags are mutually exclusive. You must set exactly one of these flags. See “[Pixels Outside the Image Buffer](#)” (page 21).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

Depending on the relative sizes of the source image and destination image, parts of the source image may be clipped, and areas outside the source image may appear in the destination image.

Resampling is done using a Lanczos filter. Either Lanczos3 or Lanczos5 will be used, depending on a flag setting. Using the Lanczos5 kernel will give you slower but more accurate results.

vImageRotate_Planar8

This function rotates a source image by a given angle, placing the result in a destination buffer. The center point of the source image is mapped to the center point of the destination image. No scaling is done. The image must be in Planar8 format.

```
vImage_Error vImageRotate_Planar8 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    float angleInRadians,
    Pixel_8 backgroundColor,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

angleInRadians

The angle of rotation, in radians.

backgroundColor

A pixel value to be used as a background color. Only used if the `kvImageBackgroundColorFill` flag is set.

vImage Functions*flags*

The following flags are used:

`kvImageBackgroundColorFill`

If set, the function will use the Background Color Fill technique to access pixels outside the source image.

`kvImageEdgeExtend`

If set, the function will use the Edge Extend technique to access pixels outside the source image.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

`kvImageHighQualityResampling`

If set, the function will use a Lanczos5 kernel for resampling. Otherwise it will use a Lanczos3 kernel.

The `kvImageBackgroundColorFill` and `kvImageEdgeExtend` flags are mutually exclusive. You must set exactly one of these flags. See “[Pixels Outside the Image Buffer](#)” (page 21).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

Depending on the relative sizes of the source image and destination image, parts of the source image may be clipped, and areas outside the source image may appear in the destination image.

Resampling is done using a Lanczos filter. Either Lanczos3 or Lanczos5 will be used, depending on a flag setting. Using the Lanczos5 kernel will give you slower but more accurate results.

vImageScale_ARGBFFFF

This function scales a source image to fit a destination buffer, placing the result in the destination buffer. The image must be in ARGBFFFF format.

```
vImage_Error vImageScale_ARGBFFFF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    vImage_Flags flags
);
```

Parameters

src

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

vImage Functions***tempBuffer***

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

flags

The following flags are used: You must set exactly one of the following flag fields: `kvImageBackgroundColorFill` or `kvImageEdgeExtend`. See “[Pixels Outside the Image Buffer](#)” (page 21).

`kvImageBackgroundColorFill`

If set, the function will use the Background Color Fill technique to access pixels outside the source image.

`kvImageEdgeExtend`

If set, the function will use the Edge Extend technique to access pixels outside the source image.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

`kvImageHighQualityResampling`

If set, the function will use a Lanczos5 kernel for resampling. Otherwise it will use a Lanczos3 kernel.

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The relative size of the source image and the destination buffer determine the scaling factors. They may be different in the X and Y directions.

Resampling is done using a Lanczos filter. Either Lanczos3 or Lanczos5 will be used, depending on a flag setting. Using the Lanczos5 kernel will give you slower but more accurate results.

****vImageScale_ARGB8888****

This function scales a source image to fit a destination buffer, placing the result in the destination buffer. The image must be in ARGB8888 format.

```
vImage_Error vImageScale_ARGB8888 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

flags

The following flags are used:

`kvImageBackgroundColorFill`

If set, the function will use the Background Color Fill technique to access pixels outside the source image.

`kvImageEdgeExtend`

If set, the function will use the Edge Extend technique to access pixels outside the source image.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

`kvImageHighQualityResampling`

If set, the function will use a Lanczos5 kernel for resampling. Otherwise it will use a Lanczos3 kernel.

The `kvImageBackgroundColorFill` and `kvImageEdgeExtend` flags are mutually exclusive. You must set exactly one of these flags. See “[Pixels Outside the Image Buffer](#)” (page 21).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The relative size of the source image and the destination buffer determine the scaling factors. They may be different in the X and Y directions.

Resampling is done using a Lanczos filter. Either Lanczos3 or Lanczos5 will be used, depending on a flag setting. Using the Lanczos5 kernel will give you slower but more accurate results.

The function uses Edge Extend to assign values to pixels that are outside the source buffer. This prevents a background color from “bleeding” into the edges of the scaled image.

vImageScale_PlanarF

This function scales a source image to fit a destination buffer, placing the result in the destination buffer. The image must be in PlanarF format.

```
vImage_Error vImageScale_PlanarF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

flags

The following flags are used:

`kvImageBackgroundColorFill`

If set, the function will use the Background Color Fill technique to access pixels outside the source image.

`kvImageEdgeExtend`

If set, the function will use the Edge Extend technique to access pixels outside the source image.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

`kvImageHighQualityResampling`

If set, the function will use a Lanczos5 kernel for resampling. Otherwise it will use a Lanczos3 kernel.

The `kvImageBackgroundColorFill` and `kvImageEdgeExtend` flags are mutually exclusive. You must set exactly one of these flags. See “[Pixels Outside the Image Buffer](#)” (page 21).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The relative size of the source image and the destination buffer determine the scaling factors. They may be different in the X and Y directions.

Resampling is done using a Lanczos filter. Either Lanczos3 or Lanczos5 will be used, depending on a flag setting. Using the Lanczos5 kernel will give you slower but more accurate results.

The function uses Edge Extend to assign values to pixels that are outside the source buffer. This prevents a background color from “bleeding” into the edges of the scaled image.

vImageScale_Planar8

This function scales a source image to fit a destination buffer, placing the result in the destination buffer. The image must be in Planar8 format.

```
vImage_Error vImageScale_Planar8 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    void *tempBuffer,
    vImage_Flags flags
);
```

Parameters

src

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

tempBuffer

A pointer to a temporary buffer. If you pass `NULL`, the function will allocate the buffer, then deallocate it before returning. Alternatively, you may allocate the buffer yourself, in which case you are responsible for deallocating it when it is no longer needed.

You can determine the minimum size for this buffer by calling this function first with the `kvImageGetTempBufferSize` flag. Pass the same values for all other parameters that you intend to use in your “real” call. The return value (if positive) will be required minimum size; no other processing will be done. If a negative value is returned, it indicates an error as usual.

Then allocate your temporary buffer and make a second call without the `kvImageGetTempBufferSize` flag, passing the buffer pointer.

vImage Functions*flags*

The following flags are used:

kvImageBackgroundColorFill

If set, the function will use the Background Color Fill technique to access pixels outside the source image.

kvImageEdgeExtend

If set, the function will use the Edge Extend technique to access pixels outside the source image.

kvImageDoNotTile

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

kvImageHighQualityResampling

If set, the function will use a Lanczos5 kernel for resampling. Otherwise it will use a Lanczos3 kernel.

The `kvImageBackgroundColorFill` and `kvImageEdgeExtend` flags are mutually exclusive. You must set exactly one of these flags. See “[Pixels Outside the Image Buffer](#)” (page 21).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The relative size of the source image and the destination buffer determine the scaling factors. They may be different in the X and Y directions.

Resampling is done using a Lanczos filter. Either Lanczos3 or Lanczos5 will be used, depending on a flag setting. Using the Lanczos5 kernel will give you slower but more accurate results.

The function uses Edge Extend to assign values to pixels that are outside the source buffer. This prevents a background color from “bleeding” into the edges of the scaled image.

vImageVerticalReflect_ARGBFFFF

This function reflects a source image top to bottom across the center vertical line of the image, placing the result in a destination buffer. This causes the image to appear upside down as if seen from behind. No scaling or resampling is done. The image must be in ARGBFFFF format.

```
vImage_Error vImageHorizontalReflect_ARGBFFFF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

vImage Functions*flags*

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The source and destination buffers must have the same height and the same width (in pixels).

vImageVerticalReflect_ARGB8888

This function reflects a source image top to bottom across the center vertical line of the image, placing the result in a destination buffer. This causes the image to appear upside down as if seen from behind. No scaling or resampling is done. The image must be in ARGBFFFF format.

```
vImage_Error vImageVerticalReflect_ARGB8888 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The source and destination buffers must have the same height and the same width (in pixels).

vImageVerticalReflect_PlanarF

This function reflects a source image top to bottom across the center vertical line of the image, placing the result in a destination buffer. This causes the image to appear upside down as if seen from behind. No scaling or resampling is done. The image must be in PlanarF format.

vImage Functions

```
vImage_Error vImageVerticalReflect_PlanarF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

flags

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The source and destination buffers must have the same height and the same width (in pixels).

`vImageVerticalReflect_Planar8`

This function reflects a source image top to bottom across the center vertical line of the image, placing the result in a destination buffer. This causes the image to appear upside down as if seen from behind. No scaling or resampling is done. The image must be in Planar8 format.

```
vImage_Error vImageVerticalReflect_PlanarF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

vImage Functions*flags*

The following flag is used:

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

The source and destination buffers must have the same height and the same width (in pixels).

`vImageVerticalShear_ARGBFFFF`

This function performs a vertical shear operation on a region of interest of a source image. It also translates and scales the image (both in the vertical direction). The transformed image is placed in a destination buffer. More than just the region of interest may be transformed; the function transforms as much of the source image as it needs in order to attempt to fill the destination buffer. The image must be in ARGBFFFF format.

```
vImage_Error vImageVerticalShear_ARGBFFFF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    uint32_t srcOffsetToROI_X,
    uint32_t srcOffsetToROI_Y,
    float yTranslate,
    float shearSlope,
    ResamplingFilter filter,
    Pixel_FFFF backColor,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

yTranslate

A translation value for the vertical direction.

shearSlope

The slope of the top edge of the sheared image, measured in a clockwise direction.

vImage Functions*filter*

The resampling filter to be used with this function. You create this object by calling `vImageNewResamplingFilter` (to use a default resampling filter supplied by `vImage`) or `vImageNewResamplingFilterForFunctionUsingBuffer` (to use a custom resampling filter that you supply). When the resampling filter is created, you can also set a scale factor that will be used in the horizontal shear operation.

backgroundColor

A pixel value to be used as a background color. Only used if the `kvImageBackgroundColorFill` is set.

flags

The following flags are used: You must set exactly one of the following flags: `kvImageBackgroundColorFill`, `kvImageEdgeExtend`. Set these flags to the same value used when you create the filter parameter.

`kvImageBackgroundColorFill`

If set, the function will use the Background Color Fill technique to handle pixels outside the source image.

`kvImageEdgeExtend`

If set, the function will use the Edge Extend technique to handle pixels outside the source image.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

`kvImageHighQualityResampling`

Set this flag to the same value used when you create the filter parameter.

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

In addition to supplying space for the destination image, the `dest` parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by `dest`.

`vImageVerticalShear_ARGB8888`

This function performs a vertical shear operation on a region of interest of a source image. It also translates and scales the image (both in the vertical direction). The transformed image is placed in a destination buffer. More than just the region of interest may be transformed; the function transforms as much of the source image as it needs in order to attempt to fill the destination buffer. The image must be in ARGB8888 format.

```
vImage_Error vImageVerticalShear_ARGBFFFF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    uint32_t srcOffsetToROI_X,
    uint32_t srcOffsetToROI_Y,
    float yTranslate,
    float shearSlope,
    ResamplingFilter filter,
```

vImage Functions

```
Pixel_FFFF backColor,
vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

yTranslate

A translation value for the vertical direction.

shearSlope

The slope of the top edge of the sheared image, measured in a clockwise direction.

filter

The resampling filter to be used with this function. You create this object by calling `vImageNewResamplingFilter` (to use a default resampling filter supplied by vImage) or `vImageNewResamplingFilterForFunctionUsingBuffer` (to use a custom resampling filter that you supply). When the resampling filter is created, you can also set a scale factor that will be used in the horizontal shear operation.

backgroundColor

A pixel value to be used as a background color. Only used if the `kvImageBackgroundColorFill` is set.

vImage Functions*flags*

The following flags are used: You must set exactly one of the following flags: kvImageBackgroundColorFill, kvImageEdgeExtend. Set these flags to the same value used when you create the filter parameter.

kvImageBackgroundColorFill

If set, the function will use the Background Color Fill technique to handle pixels outside the source image.

kvImageEdgeExtend

If set, the function will use the Edge Extend technique to handle pixels outside the source image.

kvImageDoNotTile

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

kvImageHighQualityResampling

Set this flag to the same value used when you create the filter parameter.

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

In addition to supplying space for the destination image, the `dest` parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by `dest`.

vImageVerticalShear_PlanarF

This function performs a vertical shear operation on a region of interest of a source image. It also translates and scales the image (both in the vertical direction). The transformed image is placed in a destination buffer. More than just the region of interest may be transformed; the function transforms as much of the source image as it needs in order to attempt to fill the destination buffer. The image must be in PlanarF format.

```
vImage_Error vImageVerticalShear_PlanarF (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    uint32_t srcOffsetToROI_X,
    uint32_t srcOffsetToROI_Y,
    float yTranslate,
    float shearSlope,
    ResamplingFilter filter,
    Pixel_F backColor,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

vImage Functions

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

yTranslate

A translation value for the vertical direction.

shearSlope

The slope of the top edge of the sheared image, measured in a clockwise direction.

filter

The resampling filter to be used with this function. You create this object by calling `vImageNewResamplingFilter` (to use a default resampling filter supplied by vImage) or `vImageNewResamplingFilterForFunctionUsingBuffer` (to use a custom resampling filter that you supply). When the resampling filter is created, you can also set a scale factor that will be used in the horizontal shear operation.

backgroundColor

A pixel value to be used as a background color. Only used if the `kvImageBackgroundColorFill` is set.

flags

The following flags are used: You must set exactly one of the following flags:
`kvImageBackgroundColorFill`, `kvImageEdgeExtend`. Set these flags to the same value used when you create the filter parameter.

`kvImageBackgroundColorFill`

If set, the function will use the Background Color Fill technique to handle pixels outside the source image.

`kvImageEdgeExtend`

If set, the function will use the Edge Extend technique to handle pixels outside the source image.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

`kvImageHighQualityResampling`

Set this flag to the same value used when you create the filter parameter.

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

In addition to supplying space for the destination image, the *dest* parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by *dest*.

vImageVerticalShear_Planar8

This function performs a vertical shear operation on a region of interest of a source image. It also translates and scales the image (both in the vertical direction). The transformed image is placed in a destination buffer. More than just the region of interest may be transformed; the function transforms as much of the source image as it needs in order to attempt to fill the destination buffer. The image must be in ARGBFFFF format.

```
vImage_Error vImageVerticalShear_Planar8 (
    const vImage_Buffer *src,
    const vImage_Buffer *dest,
    uint32_t srcOffsetToROI_X,
    uint32_t srcOffsetToROI_Y,
    float yTranslate,
    float shearSlope,
    ResamplingFilter filter,
    Pixel_8 backColor,
    vImage_Flags flags
);
```

Parameters*src*

A pointer to a structure of type `vImage_Buffer` containing the source image.

dest

A pointer to a structure of type `vImage_Buffer`. On return, it contains the destination image. You must set the fields of this structure yourself, and allocate memory for its data. When you are done with it you must deallocate the memory.

srcOffsetToROI_X

The horizontal offset, in pixels, to the upper-left pixel of the region of interest within the source image.

srcOffsetToROI_Y

The vertical offset, in pixels, to the upper-left pixel of the region of interest within the source image.

yTranslate

A translation value for the vertical direction.

shearSlope

The slope of the top edge of the sheared image, measured in a clockwise direction.

filter

The resampling filter to be used with this function. You create this object by calling `vImageNewResamplingFilter` (to use a default resampling filter supplied by `vImage`) or `vImageNewResamplingFilterForFunctionUsingBuffer` (to use a custom resampling filter that you supply). When the resampling filter is created, you can also set a scale factor that will be used in the horizontal shear operation.

backgroundColor

A pixel value to be used as a background color. Only used if the `kvImageBackgroundColorFill` is set.

flags

The following flags are used: You must set exactly one of the following flags: `kvImageBackgroundColorFill`, `kvImageEdgeExtend`. Set these flags to the same value used when you create the filter parameter.

`kvImageBackgroundColorFill`

If set, the function will use the Background Color Fill technique to handle pixels outside the source image.

`kvImageEdgeExtend`

If set, the function will use the Edge Extend technique to handle pixels outside the source image.

`kvImageDoNotTile`

Setting this flag prevents internal tiling and/or multithreading. Please see “[Tiling for Cache Utilization](#)” (page 23).

`kvImageHighQualityResampling`

Set this flag to the same value used when you create the filter parameter.

Return value

If the function succeeds, it returns `kvImageNoError`. Otherwise it returns an error code.

Discussion

In addition to supplying space for the destination image, the `dest` parameter also specifies the size of the region of interest in within the source image. The region of interest has the same height and width (in pixels) as the destination buffer pointed to by `dest`.

vImage Callback Functions

MyResamplingFilter

```
void
MyResamplingFilterFunc (
    const float *xArray,
    float *yArray,
    int count,
    void *userData
)
```

Parameters

xArray

A variable-length array of values, supplied by vImage, for which the your resampling kernel function should be evaluated. The values will be between $-kernelWidth$ and $+kernelWidth$, inclusive, where $kernelWidth$ is the value supplied to `vImageNewResamplingFilterForFunctionUsingBuffer` along with this function.

yArray

A variable-length array, the same length as *xArray*. On return, each element *yArray[i]* should contain the value of your resampling kernel function evaluated on *xArray[i]*. That is, $yArray[i] = f(xArray[i])$. You are responsible for setting the values of the elements of *yArray*. In most cases, best results will be achieved if the values in the *yArray* add up to 1.0 (that is, if the resampling filter values are normalized).

count

The length of *xArray*.

userData

The pointer to user data you supplied to `vImageNewResamplingFilterForFunctionUsingBuffer` along with this function. If you passed `NULL`, it will be `NULL`.

Discussion

C H A P T E R 1 1
vImage Callback Functions

vImage Data Types

vImage_Buffer

This data structure contains an image, which can be in Planar8, PlanarF, ARGB8888, or ARGBFFFF format.

```
typedef struct vImage_Buffer
{
    void        *data;
    uint32_t    height;
    uint32_t    width;
    uint32_t    rowBytes;
} vImage_Buffer;
```

Fields

data

A pointer to the top left pixel of the image. If the `vImage_Buffer` is being used as a destination buffer, so that the pixel data will be filled in by some function call, the pixel data may not exist. But the data pointer must be valid and point to an area of memory that is an appropriate size for the `vImage_Buffer`. Specifically, it must be at least `height * rowBytes` bytes.

height

The number of rows in the image.

width

The number of pixels in one row of the image.

rowBytes

The number of bytes in a pixel row. This is the distance, in bytes, between the start of one row of the image and the start of the next. This must be at least `width * pixel size`, where pixel size depends on the image format. It can be more, in which case extra bytes extend beyond the end of each row of pixels. This is a convenient way to pad the row length for improved performance, or to describe an image within a larger image without copying the data. The extra bytes are not considered part of the image represented by the `vImage_Buffer`.

Discussion

This is the basic data structure used by vImage. Given a `vImage_Buffer` as an input, vImage functions will not attempt to read pixel data outside the area described by the `height` and `width` fields. They will also not write data outside that area.

vImage_AffineTransform

This data structure contains a matrix representing an affine transformation.

```
typedef struct vImage_AffineTransform
{
    float a, b, c, d;
    float tx, ty;
} vImage_AffineTransform;
```

Discussion

This structure represents the 3x2 matrix

$$\begin{pmatrix} a & b \\ c & d \\ tx & ty \end{pmatrix}$$

It is defined to be the same as the CGAffineTransform. Some functions for creating and manipulating matrixes of this form can be found in CoreGraphics/CGAffineTransform.h.

vImage_Error

Error codes generated by vImage

```
typedef int32_t vImage_Error;
```

vImage_Flags

Flags used by vImage

```
typedef uint32_t vImage_Flags;
```

Pixel_8

8 bit planar pixel value

```
typedef uint8_t Pixel_8;
```

Pixel_F

floating point planar pixel value

```
typedef float Pixel_F;
```

Pixel_8888

ARGB interleaved (8 bit/channel) pixel value. uint8_t[4] = { alpha, red, green, blue }

```
typedef uint8_t Pixel_8888[4];
```

Pixel_FFFF

ARGB interleaved (floating point) pixel value. float[4] = { alpha, red, green, blue }

```
typedef uint8_t Pixel_8888[4];
```

ResamplingFilter

Object encapsulating a resampling kernel function.

```
typedef void *ResamplingFilter;
```

Discussion

The internal structure of this object is deliberately undocumented. It may contain a pointer to a function, rows of precalculated values, flag settings, and so on. It always contains a scale factor, which is used by the Shear function the object is passed to.

vImage Constants

vImage Error Codes

```
enum
{
    kvImageNoError                    = 0,
    kvImageRoiLargerThanInputBuffer   = -21766,
    kvImageInvalidKernelSize          = -21767,
    kvImageNoEdgeStyleSpecified       = -21768,
    kvImageInvalidOffset_X            = -21769,
    kvImageInvalidOffset_Y            = -21770,
    kvImageMemoryAllocationError     = -21771,
    kvImageNullPointerArgument       = -21772,
    kvImageInvalidParameter          = -21773,
    kvImageBufferSizeMismatch        = -21774
};
```

Constants

kvImageNoError

The vImage function completed without error.

kvImageRoiLargerThanInputBuffer

The region of interest, as specified by srcOffsetToROI_X, srcOffsetToROI_Y, and the height and width of the destination buffer, extends beyond the bottom edge or right edge of the source buffer.

kvImageInvalidKernelSize

Either the kernel height, the kernel width, or both, are even.

kvImageNoEdgeStyleSpecified

The function requires at least one of the “edge-style flags” (kvImageCopyInPlace, kvImageBackgroundColorFill, or kvImageEdgeExtend) to be set, but none is.

kvImageInvalidOffset_X

The srcOffsetToROI_X parameter used to specify the left edge of the region of interest is greater than the width of the source image.

kvImageInvalidOffset_Y

The srcOffsetToROI_Y parameter used to specify the top edge of the region of interest is greater than the height of the source image.

kvImageMemoryAllocationError

An attempt to allocate memory failed.

vImage Constants

kvImageNullPointerArgument
A pointer parameter required to be non-NULL was NULL.

kvImageInvalidParameter
Invalid parameter.

kvImageBufferSizeMismatch
The function requires the source and destination buffers to have the same height and the same width (in pixels), and they do not.

vImage Flags

```
enum
{
    kvImageNoFlags          = 0,
    kvImageLeaveAlphaUnchanged = 1,
    kvImageCopyInPlace       = 2,
    kvImageBackgroundColorFill = 4,
    kvImageEdgeExtend        = 8,
    kvImageDoNotTile         = 16,
    kvImageHighQualityResampling = 32,
    kvImageTruncateKernel   = 64,
    kvImageGetTempBufferSize = 128
};
```

Constants

kvImageNoFlags
No flags set.

kvImageLeaveAlphaUnchanged
Operate on red, green, and blue channels only. Copy alpha channel unchanged from source to destination.

kvImageCopyInPlace
Use the Copy In Place technique for handling pixels out the image buffer.

kvImageBackgroundColorFill
Use the Background Color Fill technique for handling pixels out the image buffer.

kvImageEdgeExtend
Use the Edge Extend technique for handling pixels out the image buffer.

kvImageDoNotTile
Do not use vImage internal tiling routines.

kvImageHighQualityResampling
Use a higher quality, slower filter for resampling.

kvImageTruncateKernel
Use only the part of the kernel that overlaps the image being processed.

kvImageGetTempBufferSize
Return the minimum temporary buffer size for the operation, given the parameters provided.

Rotation Constants for Use With Rotate90 Function

```
enum
{
    kRotate0DegreesClockwise      = 0,
    kRotate90DegreesClockwise     = 3,
    kRotate180DegreesClockwise    = 2,
    kRotate270DegreesClockwise    = 1,
    kRotate0DegreesCounterClockwise = 0,
    kRotate90DegreesCounterClockwise= 1,
    kRotate180DegreesCounterClockwise= 2,
    kRotate270DegreesCounterClockwise= 3
};
```

Constants

kRotate0DegreesClockwise	Rotate 0 degrees (that is, copy without rotating)
kRotate90DegreesClockwise	Rotate 90 degrees clockwise
kRotate180DegreesClockwise	Rotate 180 degrees clockwise
kRotate270DegreesClockwise	Rotate 270 degrees clockwise
kRotate0DegreesCounterClockwise	Rotate 0 degrees (that is, copy without rotating)
kRotate90DegreesCounterClockwise	Rotate 90 degrees counter-clockwise
kRotate180DegreesCounterClockwise	Rotate 180 degrees counter-clockwise
kRotate270DegreesCounterClockwise	Rotate 270 degrees counter-clockwise

Document Revision History

This table describes the changes to *Optimizing Image Processing With vImage*.

Date	Notes
2005-04-29	Fixed an error in the name of a function, and a reference to the title in the "About This Document" section.
	Updated for Mac OS X v10.4 and the current versions of all tools. Added a chapter on image transformation operations and added information on many new functions to other chapters.
2003-08-26	New document describing the vImage framework.

R E V I S I O N H I S T O R Y

Document Revision History

Index

A

affine transformations 51
affine warp functions 55
alpha blend function
 constant 82
 non-premultiplied to premultiplied 81
 non-premultiplied 78
 premultiplied 80
alpha compositing 77, 95
alpha compositing functions
 clip to alpha 85
 alpha blend
 non-premultiplied 78
 non-premultiplied to premultiplied 81
 premultiplied 80, 82
 premultiply data 85
alpha compositing functions
 unpremultiply data 84
ARGB8888 17
ARGBFFFF 17

B

background color fill 21

C

close
 definition 44
contrast stretch functions 73
conversion functions
 convert chunky to planar 114
 convert planar to chunky 115
 pointer format 113
 vImageBufferFill 95
 vImageClip_PlanarF 97
 vImageConvert_16SToF 101

vImageConvert_16UToF 101
vImageConvert_16UToPlanar8 102
vImageConvert_ARGB1555ToARGB8888 105
vImageConvert_ARGB1555ToPlanar8 106
vImageConvert_ARGB8888ToARGB1555 106
vImageConvert_ARGB8888toPlanar8 99
vImageConvert_ARGB8888ToRGB565 108
vImageConvert_ARGB8888ToRGB888 110
vImageConvert_ARGBFFFFtoPlanarF 100
vImageConvert_FTo16S 102
vImageConvert_FTo16U 103
vImageConvert_Planar16FToPlanarF 104
vImageConvert_Planar8To16U 104
vImageConvert_Planar8ToARGB1555 107
vImageConvert_Planar8toARGB8888 99
vImageConvert_Planar8ToPlanarF 98
vImageConvert_Planar8ToRGB565 109
vImageConvert_Planar8ToRGB888 111
vImageConvert_PlanarFtoARGBFFFF 100
vImageConvert_PlanarFToPlanar16F 105
vImageConvert_PlanarFToRGBFFF 111
vImageConvert_RGB565ToARGB8888 107
vImageConvert_RGB565ToPlanar8 109
vImageConvert_RGB888ToARGB8888 110
vImageConvert_RGB888ToPlanar8 111
vImageConvert_RGBFFFToPlanarF 112
vImageFlatten 97
vImageOverwriteChannels 96
vImageOverwriteChannelsWithScalar 96
vImagePermuteChannels 97
vImageTableLookUp_ARGB8888 113
vImageTableLookUp_Planar8 112
convolution functions
 vImageGetMinimumTempBufferSizeForConvolution 39
convolution
 defined 29
 edge condition 30
 kernels for 30
 normalization 30
convolve functions 31
copy in place 21

D

dilate functions 45
dilation
definition 43

E

edge extend 21
ends-in contrast stretch functions 73
equalization functions 70
erode functions 46
erosion
definition 44

F

flags 20

G

geometric functions
affine warp 55
horizontal reflect 57
horizontal shear 59
rotate 52
rotate90 58
scale 54
vertical reflect 57
vertical shear 61
vImageDestroyResamplingFilter 63
vImageGetMinimumGeometryTempBufferSize 56
vImageGetResamplingFilterSize 64
vImageNewResamplingFilter 63
vImageNewResamplingFilterForFunctionUsingBuffer 64
geometric operations 49
edge conditions 52

H

histogram calculation functions 69
histogram defintion parameters 68
histogram functions
contrast stretch 73
equalization 70
histogram calculation 69

histogram specification 72
vImageGetMinimumBufferSizeForHistogram 76
histogram operations 68
ends-in contrast stretch 73
histogram specification functions 72
histograms 67
horizontal reflect functions 57
horizontal shear functions 59

I

image transformations 87
interleaved 24

K

kRotate0DegreesClockwise constant 329
kRotate0DegreesCounterClockwise constant 329
kRotate180DegreesClockwise constant 329
kRotate180DegreesCounterClockwise constant 329
kRotate270DegreesClockwise constant 329
kRotate270DegreesCounterClockwise constant 329
kRotate90DegreesClockwise constant 329
kRotate90DegreesCounterClockwise constant 329
kvImageBackgroundColorFill constant 21, 328
kvImageBackgroundColorFill flag 20, 30
kvImageBufferSizeMismatch constant 328
kvImageCopyInPlace constant 21, 328
kvImageCopyInPlace flag 20, 30
kvImageDoNotTile constant 328
kvImageDoNotTile flag 23
kvImageEdgeExtend constant 21, 328
kvImageEdgeExtend flag 20, 30
kvImageGetTempBufferSize constant 26, 328
kvImageHighQualityResampling constant 328
kvImageHighQualityResampling flag 49
kvImageInvalidKernelSize constant 327
kvImageInvalidOffset_X constant 327
kvImageInvalidOffset_Y constant 327
kvImageInvalidParameter constant 328
kvImageLeaveAlphaUnchanged constant 328
kvImageMemoryAllocationError constant 327
kvImageNoEdgeStyleSpecified constant 327
kvImageNoError constant 327
kvImageNoFlags constant 328
kvImageNoFlags flag 20
kvImageNullPointerArgument constant 328
kvImageRoiLargerThanInputBuffer constant 327
kvImageTruncateKernel constant 30, 328
kvTruncateKernel constant 21

L

Lanczos filter [49](#)
look-up tables [69](#)

M

max functions [46](#)
max
 definition [44](#)
min functions [47](#)
min
 definition [44](#)
morphological functions
 dilate [45](#)
 erode [46](#)
 max [46](#)
 min [47](#)
 vImageGetMinimumTempBufferSizeForMinMax [48](#)
morphological operations
 on grayscale images [43](#)
 close
 definition [44](#)
 definition [43](#)
 dilation
 definition [43](#)
 erosion
 definition [44](#)
 for full-color images [44](#)
kernels for [45](#)
max
 definition [44](#)
min
 definition [44](#)
open
 definition [44](#)
morphology functions
 edge conditions [45](#)
multiprocessing [23](#)
MyResamplingFilter callback [321](#)

N

non-premultiplied alpha [77](#)

O

open

definition [44](#)

P

performance [23](#)
Pixel_8 data type [324](#)
Pixel_8888 data type [324](#)
Pixel_F data type [324](#)
Pixel_FFFF data type [325](#)
planar [24](#)
Planar8 [17](#)
PlanarF [17](#)
premultiplied alpha [77](#)

R

real-time processing [24](#)
region of interest [19](#)
resampling [49](#)
resampling filter [49](#)
ResamplingFilter data type [325](#)
rotate functions [52](#)
Rotate90 functions [58](#)
Rotation Constants for Use With Rotate90 Function
 [329](#)

S

scale functions [54](#)

T

temporary buffers [26](#)
tiling [23](#)

V

vertical reflect functions [57](#)
vertical shear functions [61](#)
vImage Error Codes [327](#)
vImage Flags [328](#)
vImageAffineWarp_ARGB8888 function [277](#)
vImageAffineWarp_ARGBFFFF function [275](#)
vImageAffineWarp_Planar8 function [280](#)
vImageAffineWarp_PlanarF function [278](#)

vImageAlphaBlend_ARGB8888 function 118
 vImageAlphaBlend_ARGBFFFF function 117
 vImageAlphaBlend_NonpremultipliedToPremultiplied_-
 ARGB8888 function 128
 vImageAlphaBlend_NonpremultipliedToPremultiplied_-
 ARGBFFFF function 128
 vImageAlphaBlend_NonpremultipliedToPremultiplied_-
 Planar8 function 130
 vImageAlphaBlend_NonpremultipliedToPremultiplied_-
 PlanarF function 129
 vImageAlphaBlend_Planar8 function 120
 vImageAlphaBlend_PlanarF function 118
 vImageBoxConvolve_ARGB8888 function 201
 vImageBoxConvolve_Planar8 function 199
 vImageBufferFill_ARGB8888 function 139
 vImageBufferFill_ARGBFFFF function 140
 vImageClip_PlanarF function 140
 vImageContrastStretch_ARGB8888 function 233
 vImageContrastStretch_ARGBFFFF function 231
 vImageContrastStretch_Planar8 function 235
 vImageContrastStretch_PlanarF function 233
 vImageConvert_16SToF function 141
 vImageConvert_16UToF function 142
 vImageConvert_16UToPlanar8 function 143
 vImageConvert_ARG1555toARGB8888 function 143
 vImageConvert_ARGB1555toPlanar8 function 144
 vImageConvert_ARGB8888toARGB1555 function 145
 vImageConvert_ARGB8888toPlanar8 function 146
 vImageConvert_ARGB8888toRGB565 function 147
 vImageConvert_ARGB8888toRGB888 function 147
 vImageConvert_ARGBFFFtoPlanarF function 148
 vImageConvert_ChunkToPlanar8 function 149
 vImageConvert_ChunkToPlanarF function 150
 vImageConvert_FTo16S function 151
 vImageConvert_FTo16U function 152
 vImageConvert_Planar16FtoPlanarF function 153
 vImageConvert_Planar8to16U function 154
 vImageConvert_Planar8toARGB1555 function 154
 vImageConvert_Planar8toARGB8888 function 155
 vImageConvert_Planar8toPlanarF function 156
 vImageConvert_Planar8toRGB565 function 157
 vImageConvert_Planar8toRGB888 function 158
 vImageConvert_PlanarFtoARGBFFF function 160
 vImageConvert_PlanarFtoPlanar16F function 161
 vImageConvert_PlanarFtoPlanar8 function 161
 vImageConvert_PlanarFtoRGBFFF function 159
 vImageConvert_PlanarToChunky8 function 162
 vImageConvert_PlanarToChunkyF function 163
 vImageConvert_RGB565toARGB8888 function 166
 vImageConvert_RGB565toPlanar8 function 165
 vImageConvert_RGB888toARGB8888 function 166
 vImageConvert_RGB888toPlanar8 function 167
 vImageConvert_RGBFFFtoPlanarF function 168
 vImageConvolveMultiKernel_ARGB8888 function
 197
 vImageConvolveMultiKernel_ARGBFFFF function
 195
 vImageConvolveWithBias_ARGB8888 function 189
 vImageConvolveWithBias_ARGBFFFF function 187
 vImageConvolveWithBias_Planar8 function 193
 vImageConvolveWithBias_PlanarF function 191
 vImageConvolve_ARGB8888 function 180
 vImageConvolve_ARGBFFFF function 178
 vImageConvolve_Planar8 function 184
 vImageConvolve_PlanarF function 183
 vImageCreateGammaFunction function 222
 vImageDestroyGammaFunction function 224
 vImageDestroyResamplingFilter function 63, 281
 vImageDilate_ARGB8888 function 255
 vImageDilate_ARGBFFFF function 254
 vImageDilate_Planar8 function 257
 vImageDilate_PlanarF function 256
 vImageEndsInContrastStretch_ARGB8888 function
 237
 vImageEndsInContrastStretch_ARGBFFFF function
 235
 vImageEndsInContrastStretch_Planar8 function
 240
 vImageEndsInContrastStretch_PlanarF function
 238
 vImageEqualization_ARGB8888 function 242
 vImageEqualization_ARGBFFFF function 241
 vImageEqualization_Planar8 function 244
 vImageEqualization_PlanarF function 243
 vImageErode_ARGB8888 function 259
 vImageErode_ARGBFFFF function 258
 vImageErode_Planar8 function 262
 vImageErode_PlanarF function 261
 vImageGamma_Planar8ToPlanarF function 224
 vImageGamma_PlanarF function 225
 vImageGamma_PlanarFToPlanar8 function 225
 vImageGetMinimumBufferSizeForHistogram
 function 76
 vImageGetMinimumGeometryTempBufferSize
 function 56, 291
 vImageGetMinimumTempBufferSizeForConvolution
 function 39, 207
 vImageGetMinimumTempBufferSizeForHistogram
 function 245
 vImageGetMinimumTempBufferSizeForMinMax
 function 48, 263
 vImageGetResamplingFilterSize function 64, 292
 vImageHistogramCalculation_ARGB8888 function
 247
 vImageHistogramCalculation_ARGBFFFF function
 246

vImageHistogramCalculation_Planar8 function 248
vImageHistogramCalculation_PlanarF function 248
vImageHistogramSpecification_ARGB8888 function 251
vImageHistogramSpecification_ARGBFFFF function 249
vImageHistogramSpecification_Planar8 function 253
vImageHistogramSpecification_PlanarF function 251
vImageHorizontalReflect_ARGB8888 function 284
vImageHorizontalReflect_ARGBFFFF function 282
vImageHorizontalReflect_Planar8 function 283
vImageHorizontalReflect_PlanarF function 282
vImageHorizontalShear_ARGB8888 function 286
vImageHorizontalShear_ARGBFFFF function 284
vImageHorizontalShear_Planar8 function 289
vImageHorizontalShear_PlanarF function 287
vImageInterpolatedLookupTable_PlanarF function 230
vImageLookupTable_Planar8ToPlanarF function 229
vImageLookupTable_PlanarFToPlanar8 function 229
vImageMatrixMultiply_ARGB8888 function 220
vImageMatrixMultiply_ARGBFFFF function 221
vImageMatrixMultiply_Planar8 function 218
vImageMatrixMultiply_PlanarF function 219
vImageMax_ARGB8888 function 265
vImageMax_ARGBFFFF function 264
vImageMax_Planar8 function 268
vImageMax_PlanarF function 267
vImageMin_ARGB8888 function 271
vImageMin_ARGBFFFF function 270
vImageMin_Planar8 function 274
vImageMin_PlanarF function 272
vImageNewResamplingFilter function 63, 292
vImageNewResamplingFilterForFunctionUsingBuffer function 64, 293
vImageOverwriteChannelsWithScalar_ARGB8888 function 172
vImageOverwriteChannelsWithScalar_ARGBFFFF function 173
vImageOverwriteChannelsWithScalar_Planar8 function 174
vImageOverwriteChannelsWithScalar_PlanarF function 174
vImageOverwriteChannels_ARGB8888 function 171
vImageOverwriteChannels_ARGBFFFF function 171
vImagePermuteChannels_ARGB8888 function 175
vImagePermuteChannels_ARGBFFFF function 176
vImagePiecewisePolynomial_Planar8ToPlanarF function 227
vImagePiecewisePolynomial_PlanarF function 226
vImagePiecewisePolynomial_PlanarFToPlanar8 function 228
vImagePremultipliedAlphaBlend_ARGB8888 function 122
vImagePremultipliedAlphaBlend_ARGBFFFF function 121
vImagePremultipliedAlphaBlend_Planar8 function 123
vImagePremultipliedAlphaBlend_PlanarF function 122
vImagePremultipliedConstAlphaBlend_ARGB8888 function 125
vImagePremultipliedConstAlphaBlend_ARGBFFFF function 124
vImagePremultipliedConstAlphaBlend_Planar8 function 127
vImagePremultipliedConstAlphaBlend_PlanarF function 126
vImagePremultiplyData_ARGB8888 function 132
vImagePremultiplyData_ARGBFFFF function 131
vImagePremultiplyData_Planar8 function 134
vImagePremultiplyData_PlanarF function 133
vImagePremultiplyData_RGBA8888 function 133
vImagePremultiplyData_RGBAFFFF function 131
vImageRichardsonLucyDeConvolve_ARGB8888 function 211
vImageRichardsonLucyDeConvolve_ARGBFFFF function 208
vImageRichardsonLucyDeConvolve_Planar8 function 215
vImageRichardsonLucyDeConvolve_PlanarF function 213
vImageRotate90_ARGB8888 function 296
vImageRotate90_ARGBFFFF function 295
vImageRotate90_Planar8 function 298
vImageRotate90_PlanarF function 297
vImageRotate_ARGB8888 function 301
vImageRotate_ARGBFFFF function 299
vImageRotate_Planar8 function 304
vImageRotate_PlanarF function 302
vImageScale_ARGB8888 function 306
vImageScale_ARGBFFFF function 305
vImageScale_Planar8 function 309
vImageScale_PlanarF function 308
vImageTableLookUp_ARGB8888 function 176
vImageTableLookUp_Planar8 function 177
vImageTentConvolve_ARGB8888 function 205
vImageTentConvolve_Planar8 function 203
vImageUnpremultiplyData_ARGB8888 function 136
vImageUnpremultiplyData_ARGBFFFF function 135

vImageUnpremultiplyData_Planar8 function 138
vImageUnpremultiplyData_PlanarF function 138
vImageUnpremultiplyData_RGBA8888 function 137
vImageUnpremultiplyData_RGBAFFF function 136
vImageVerticalReflect_ARGB8888 function 311
vImageVerticalReflect_ARGBFFF function 310
vImageVerticalReflect_Planar8 function 312
vImageVerticalReflect_PlanarF function 311
vImageVerticalShear_ARGB8888 function 314
vImageVerticalShear_ARGBFFF function 313
vImageVerticalShear_Planar8 function 318
vImageVerticalShear_PlanarF function 316
vImage_AffineTransform structure 324
vImage_Buffer structure 323
vImage_Buffer
 alignment 25
 freeing 25
 initializing 25
 size 25
vImage_Error data type 324
vImage_Flags data type 324
vImage_FlattenARGB8888ToRGB888 function 169
vImage_FlattenARGBFFFToRGBFFF function 170