vImage Programming Guide

Mathematical Computation

2010-06-14

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

Simultaneously published in the United States and Canada.

Contents

Figures and Listings

Introduction to vImage Programming Guide

vImage, introduced in Mac OS X v10.3, is a high-performance 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. You can call vImage functions from Cocoa, Carbon, and command line applications.

vImage optimizes image processing by using the CPU's vector processor. If a vector processor is not available, vImage uses the next best available option. This framework allows you to reap the benefits of vector processors without the need to write vectorized code.

To understand the information in this document, you should be familiar with Macintosh application development, the C programming language, and the basics of image representation and manipulation.

The vImage framework isn't the only image processing API that Mac OS X provides. Starting in Mac OS X v10.5, you also have the option of using Core Image. vImage is the ideal choice if you need to process large quantities of high-resolution images for scientific and medical projects.

Who Should Read This Document?

This document is for developers who need to write Macintosh programs that process large images quickly. Because technologies such as Quartz 2D and Core Image provide most common image manipulation routines, vImage is not intended for general-purpose image processing. It is particularly suited for:

- Incorporating high-performance graphics into their applications
- Efficiently processing large images
- Real-time video processing software
- Scientific applications that require high-accuracy numerical calculations
- Getting consistent numerical results across platforms despite video card arithmetic inconsistencies

Organization of This Document

This document is organized into the following chapters:

- "Overview of vImage" (page 9) introduces vImage and discusses how it optimizes image processing. It introduces the Accelerate framework, and explains how vector processing is used to achieve better performance. This also offers general usage guidelines for vImage and describes the workflow for incorporating vImage into an application.

- "Performing Convolution Operations" (page 13) explains the theory behind kernel convolution, a technique frequently employed in image processing to apply a filter to an image. It describes how convolutions can be performed with vImage and provides several sample kernels.
- "Performing Geometric Operations" (page 25) explains what a geometric operation is and which ones are supported in vImage.
- "Performing Morphological Operations" (page 37) describes what morphological operations are and the types available in vImage.
- "Performing Histogram Operations" (page 43) discusses histograms and how they can be useful for calibrating or analyzing images.
- "Performing Alpha Compositing Operations" (page 47) explains the theory behind alpha compositing and the alpha channel and shows how they can be used in vImage to create layered visual effects.
- "Performing Image Transformation Operations" (page 49) introduces the use of callback functions to alter each pixel in an image.

See Also

You might find these other Apple developer documents of value as you determine your image processing needs:

- *Image I/O Programming Guide*

  Image I/O facilitates the process of reading and writing images of various formats to and from the disk. vImage does not do this for you. It may be worth familiarizing yourself with Image I/O so that you can more effectively supply vImage with data.

- *Core Image Programming Guide*

  If you want to add image processing capability to an application to support color adjustment, halftone effects, stylizing filters, compositing, and transition effects, you might want to consider Core Image. The Core Image framework is a high-level Objective-C programming interface that has more than 800 built-in filters and supports writing your own custom filters.

- *OpenGL Programming Guide for Mac OS X*

  If you have a need for hardware-accelerated 3D graphics, Mac OS X provides an implementation of the OpenGL graphics standard.

Overview of vImage

vImage is a two-dimensional image processing framework. Part of the Accelerate Framework, vImage provides optimized routines for features such as image filters, scalings, reflections and rotations. While its features share several commonalities with other imaging frameworks such as Core Image, what distinguishes vImage from the rest is that it uses vectorized code. If your code runs on a G5 processor, vImage takes advantage of AltiVec. If it runs on an Intel-based Mac, vImage uses SSE. When running on a G3 architecture(which does *not* feature a vector processor), it will still run, just without vector optimization. By taking advantage of the CPU's on-board vector processor, vImage is the fastest image processing framework available, as of Mac OS X v.10.5.

When to use vImage

If real-time image processing is a need for your application, you should use vImage. Be sure to make use of temporary buffers when available to avoid blocking on calls to `malloc()`.

Aside from applying effects to images, vImage is also well-suited for applications that require consistent, standards-compliant arithmetic results.

Generally speaking, vImage is a timesaver, but there are certain costs associated with using the specialized vector processor and data caches to process your images. For example, it would not be practical to use vImage to process a single, average-sized photo. While vImage is perfectly capable of doing such processing, its benefit lies in processing images in real-time or repeating an operation successively. Core Image is a better choice if you are not dealing with large, high-resolution images.

Because vImage is a pure C framework, there are no classes to keep track of, only functions, image formats, and data types. All vImage functions begin with the word "vImage" followed by the name of the operation. Some functions also have an underscore ("_") in their names. The characters that follow the underscore usually indicate the image format that the function operates upon.

Image Formats Available in vImage

vImage supports several images formats. **Image formats** are specifications for how pixel data is represented in memory. Image file formats are the specific file types (such as JPG, PNG, GIF, and TIFF) used to exchange image data between programs and store them on the hard disk. Frameworks like Image I/O assist you in loading the various image file formats from disk and using them in memory. In memory, images are stored as two-dimensional arrays of pixel intensities (of type `int` or `float`). There is one pixel in the array for each pixel in the image.

Image formats are either planar or interleaved. A planar image format stores image data so that the data for each channel (plane) is in a separate buffer. For example, a typical planar image would have separate buffers for the red, green, blue, and alpha channels. An interleaved image format stores image data so that the data from each pixel alternates: ARGBARGBARGB . . .

Data values for images can be integer or floating-point. In vImage, image formats that use integer values represent an intensity level as an 8-bit unsigned value. Values can range from 0 to 255, inclusive, with 255 indicating full intensity and 0 no intensity. Image formats that use floating-point values typically use values in the range of 0.0 (lowest intensity) to 1.0 (full intensity). However, vImage does not enforce this range restriction and does not clip calculated values that lie outside this range.

vImage uses the following image formats for its core operations:

- **Planar8** The image is a single channel (one color or alpha value). Each pixel is an 8-bit unsigned integer value. The data type for this image format is `Pixel_8`.
- **PlanarF** The image is a single channel (one color). Each pixel is a 32-bit floating-point value. The data type for this image format is `Pixel_F`.
- **ARGB8888** The image has four interleaved channels, for alpha, red, green, and blue, in that order. Each pixel is 32 bits, an array of four 8-bit unsigned integers. The data type for this image format is `Pixel_8888`.
- **ARGBFFFF** The image has four interleaved channels, for alpha, red, green, and blue, in that order. Each pixel is an array of four floating-point numbers. The data type for this image format is `Pixel_FFFF`.
- **RGBA8888** The image has four interleaved channels, for red, green, blue, and alpha, in that order. Each pixel is 32 bits, an array of four 8-bit unsigned integers. The data type for this image format is `Pixel_8888`.
- **RGBAFFFF** The image has four interleaved channels, for red, green, blue, and alpha, in that order. Each pixel is an array of four floating-point numbers. The pixel data type for this image format is `Pixel_FFFF`.

You can also use vImage to process images in other formats by first converting them to one of vImage's core image formats. For example, you could take an image defined with 16-bit pixels and convert it to a 32-bit pixel format supported by vImage using a conversion function like `vImageConvert_16SToF`. These functions can help you convert images to and from non supported formats and supported ones:

- `vImageConvert_16SToF`

  Converts a planar (or interleaved—multiply `vImage_Buffer.width` by 4) `vImage_Buffer` of 16-bit signed integers to a buffer containing floating-point values.

- `vImageConvert_16UToF`

  Converts a planar (or interleaved—multiply `vImage_Buffer.width` by 4) `vImage_Buffer` of 16-bit unsigned integers to a buffer containing floating-point values.

- `vImageConvert_FTo16S`

  Converts a planar (or interleaved—multiply `vImage_Buffer.width` by 4) `vImage_Buffer` of floating-point values to a buffer containing 16-bit signed integers.

- `vImageConvert_FTo16U`

  Converts a planar (or interleaved—multiply `vImage_Buffer.width` by 4) `vImage_Buffer` of floating-point values to a buffer containing 16-bit unsigned integers.

- `vImageConvert_16UtoPlanar8`

  Converts a planar (or interleaved—multiply `vImage_Buffer.width` by 4) `vImage_Buffer` of 16-bit unsigned integers to a buffer containing 8-bit integer values.

- `vImageConvert_Planar8to16U`

  Converts a planar (or interleaved—multiply `vImage_Buffer.width` by 4) `vImage_Buffer` of 8-bit integer values to a buffer containing 16-bit unsigned integer values.

- `vImageConvert_ARGB1555toPlanar8`

Converts 16 bits/pixel images (with a 1-bit alpha channel and 5-bit red, green, and blue channels) to Planar8 format.

- `vImageConvert_ARGB1555toARGB8888`

  Converts 16 bits/pixel images (with a 1-bit alpha channel and 5-bit red, green, and blue channels) to ARGB8888 format.

- `vImageConvert_Planar8toARGB1555`

  Converts Planar8 images to 16 bits/pixel images with 1-bit alpha channels, and 5-bit red, green, and blue channels.

- `vImageConvert_ARGB8888toARGB1555`

  Converts ARGB8888 images to 16 bits/pixel images with 1-bit alpha channels, and 5-bit red, green, and blue channels.

- `vImageConvert_RGB565toPlanar8`

  Converts 16 bits/pixel images with 5-bit red channels, 6-bit green channels, and 5-bit blue channels to Planar8 format.

- `vImageConvert_RGB565toARGB8888`

  Converts 16 bits/pixel images with 5-bit red channels, 6-bit green channels, and 5-bit blue channels to ARGB8888 format.

- `vImageConvert_Planar8toRGB565`

  Converts Planar8 images to 16 bits/pixel images with 5-bit red channels, 6-bit green channels, and 5-bit blue channels.

- `vImageConvert_ARGB8888toRGB565`

  Converts ARGB8888 images to 16 bits/pixel images with 5-bit red channels, 6-bit green channels, and 5-bit blue channels.

- `vImageConvert_Planar16FtoPlanarF`

  Converts planar images containing 16-bit floating-point values to 32-bit floating-point values.

  Note: The 16-bit floating-point format used is identical to OpenEXR. Limit your usage of this function as conversions between 16-bit floating-point values and 32-bit floating-point values are expensive.

- `vImageConvert_PlanarFtoPlanar16F`

  Converts planar images containing 32-bit floating-point values to 16-bit floating-point values.

  Note: The 16-bit floating-point format used is identical to OpenEXR. Limit your usage of this function as conversions between 16-bit floating-point values and 32-bit floating-point values are expensive.

Data Types and 64-Bit Processing

Starting with version 10.4, Mac OS X supports 64-bit addressing for those applications compiled for 64-bit architectures. The vImage framework natively supports 64-bit architectures, which means that all vImage functions 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 64-bit processors, vImage accepts images with buffers larger than 4 gigapixels wide or tall (or both) and passes data with 64-bit pointers.

vImage uses several opaque data types to simplify handling raw image data. Most of the data types are just typedefs for `int` or `float` arrays. If you are writing shared source code that targets both the 32-bit and 64-bit architectures, you should be careful about your 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 in 64-bit architectures, and 32 bits for the 32-bit 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. See *vImage Data Types and Constants Reference* for more information.

For more information on 64-bit programming in Mac OS X, see *64-Bit Transition Guide*.

Performing Convolution Operations

Convolution is a common image processing technique that changes the intensities of a pixel to reflect the intensities of the surrounding pixels. A common use of convolution is to create image filters. Using convolution, you can get popular image effects like blur, sharpen, and edge detection—effects used by applications such as Photo Booth, iPhoto, and Aperture.

If you are interested in efficiently applying image filters to large or real-time image data, you will find the vImage functions useful. In terms of image filtering, vImage's convolution operations can perform common image filter effects such as embossing, blurring, and, posterizing.

vImage convolution operations can also be useful for sharpening or otherwise enhancing certain qualities of images. Enhancing images can be particularly useful when dealing with scientific images. Furthermore, since scientific images are often large, using these vImage operations can become necessary to achieve suitable application performance. The kinds of operations that you'd tend to use this on are edge detection, sharpening, surface contour outlining, smoothing, and motion detection.

This chapter describes convolution and shows how to use the convolution functions provided by vImage. By reading this chapter, you'll:

- See the sorts of effects you can get with convolution
- Learn what a kernel is and how to construct one
- Get an introduction to commonly used kernels and high-speed variations
- Find out, through code examples, how to apply vImage convolutions functions to an image

Convolution Kernels

Figure 2-1 shows an image before and after processing with a vImage convolution function that causes the emboss effect. To achieve this effect, vImage performs convolution using a grid-like mathematical construct called a **kernel**.

Figure 2-1 Emboss using convolution

Original Emboss

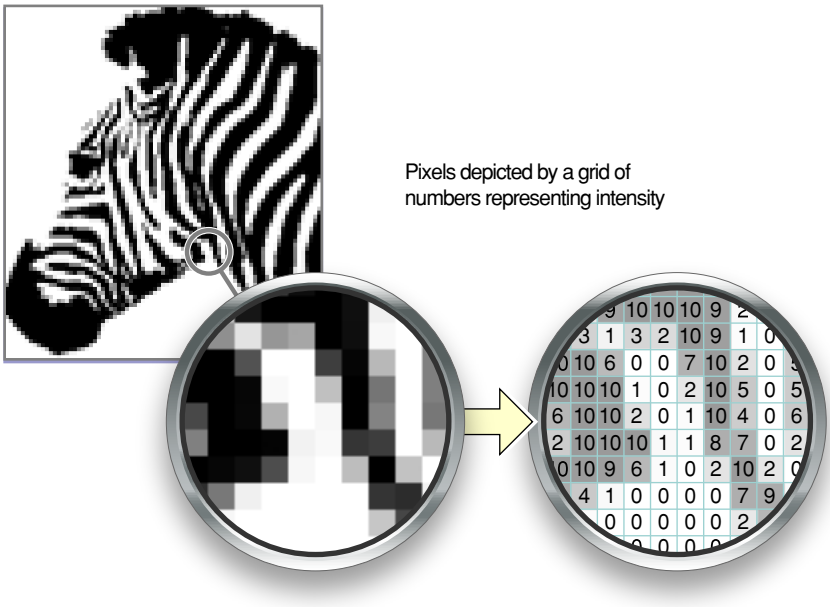
Figure 2-2 represents a 3 x 3 kernel. The height and width of the kernel do not have to be same, though they must both be odd numbers. The numbers inside the kernel are what impact the overall effect of convolution (in this case, the kernel encodes the emboss effect). The kernel (or more specifically, the values held within the kernel) is what determines how to transform the pixels from the original image into the pixels of the processed image. It may seem non-intuitive how the nine numbers in this kernel can yield an effect such as the previous emboss example. Convolution is a series of operations that alter pixel intensities depending on the intensities of neighboring pixels. vImage handles all the convolution operations based on the kernel that you supply. The kernel provides the actual numbers that are used in those operations (see Figure 2-4 (page 15) for the steps involved in convolution). Using kernels to perform convolutions is known as **kernel convolution**.

Figure 2-2 3 x 3 kernel

-2	-2	0
-2	6	0
0	0	0

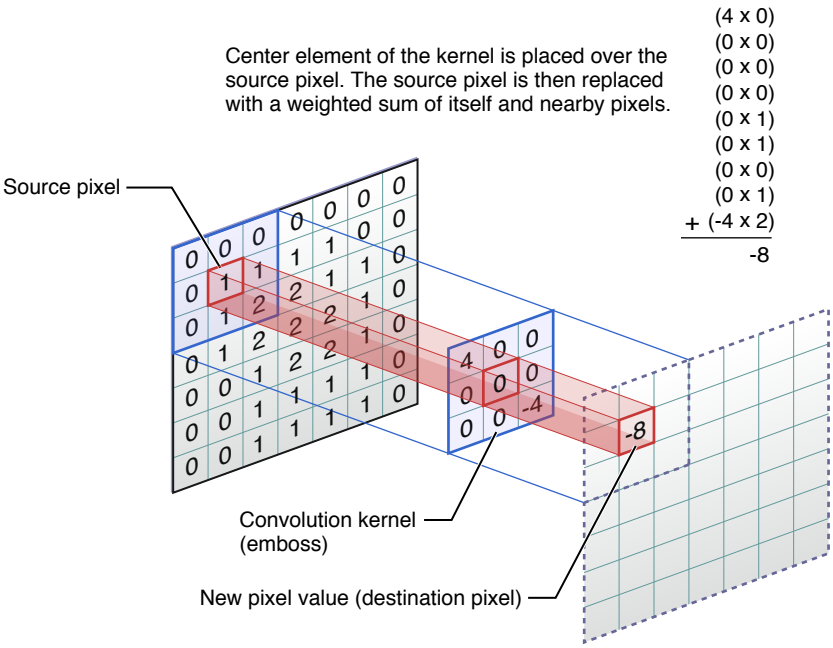
Convolutions are per-pixel operations—the same arithmetic is repeated for every pixel in the image. Bigger images therefore require more convolution arithmetic than the same operation on a smaller image. A kernel can be thought of as a two-dimensional grid of numbers that passes over each pixel of an image in sequence, performing calculations along the way. Since images can also be thought of as two-dimensional grids of numbers (or pixel intensities—see Figure 2-3), applying a kernel to an image can be visualized as a small grid (the kernel) moving across a substantially larger grid (the image).

Figure 2-3 An image is a grid of numbers



The numbers in the kernel represent the amount by which to multiply the number underneath it. The number underneath represents the intensity of the pixel over which the kernel element is hovering. During convolution, the center of the kernel passes over each pixel in the image. The process multiplies each number in the kernel by the pixel intensity value directly underneath it. This should result in as many products as there are numbers in the kernel (per pixel). The final step of the process sums all of the products together, divides them by the amount of numbers in the kernel, and this value becomes the new intensity of the pixel that was directly under the center of the kernel. Figure 2-4 shows how a kernel operates on one pixel.

Figure 2-4 Kernel convolution



Even though the kernel overlaps several different pixels (or in some cases, no pixels at all), the only pixel that it ultimately changes is the source pixel underneath the center element of the kernel. The sum of all the multiplications between the kernel and image is called the weighted sum. To ensure that the processed image is not noticeably more saturated than the original, vImage gives you the opportunity to specify a divisor by which to divide the weighted sum—a common practice in kernel convolution. Since replacing a pixel with the weighted sum of its neighboring pixels can frequently result in a much larger pixel intensity (and a brighter overall image), dividing the weighted sum can scale back the intensity of the effect and ensure that the initial brightness of the image is maintained. This procedure is called normalization. The optionally divided weighted sum is what becomes the value of the center pixel. The kernel repeats this procedure for each pixel in the source image.

Note: To perform normalization, you must pass a divisor to the convolution function that you are using. Divisors that are an exact power of two may perform better in some cases. You can supply a divisor only when the image's pixel type is `int`. floating-point convolutions do not make use of divisors. You can directly scale the floating-point values in the kernel to achieve the same normalization.

The data type used to represent the values in the kernel must match the data used to represent the pixel values in the image. For example, if the pixel type is `float`, then the values in the kernel must also be `float` values.

Keep in mind that vImage does all of this arithmetic for you, so it is not necessary to memorize the steps involved in convolution to be able to use the framework. It is a good idea to have an idea of what is going on so that you can tweak and experiment with your own kernels.

Deconvolution

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 are many deconvolution algorithms; the one vImage uses is called the Richardson-Lucy deconvolution.

The goal of Richardson-Lucy deconvolution is to find the original value of a pixel given the post-convolution pixel intensity and the number by which it was multiplied (the kernel value).

Due to these requirements, to use vImage's deconvolution functions you must provide the convolved image and the kernel used to perform the original convolution. Represented as `k` in the following equation, the kernel serves as a way of letting vImage know what type of convolution it should undo. For example, sharpening an image (a common use of deconvolution) can be thought of as doing the opposite of a blur convolution. If the kernel you supply is not symmetrical, you must pass a second kernel to the function that is the same as the first with the axes interchanged.

Richardson-Lucy deconvolution is an iterative operation; your application specifies the number of iterations desired. The more iterations you make, the greater the sharpening effect (and the time consumed). As with any sharpening operation, Richardson-Lucy amplifies noise, and at some number of iterations the noise becomes noticeable as artifacts. Figure 2-5 shows the Richardson-Lucy algorithm expressed mathematically.

Figure 2-5 Richardson-Lucy deconvolution equation

$$e_{i+1} = e_i \bullet \left(k_0 * \left(\frac{I}{k_1 * e_1} \right) \right)$$

In this equation:

- *I* is the starting image.
- e_i is the current result and e_{i+1} is being calculated; e_0 is 0.
- k_0 represents the kernel.
- k_1 is a second kernel to be used if *k[0]* is asymmetrical. If *k[0]* is symmetrical, *k[0]* is used as *k[1]*.
- The * operator indicates convolution.
- The . operator indicates multiplication of each element in one matrix by the corresponding element in the other matrix.

As with convolution, vImage handles all the individual steps of deconvolution, thus it is not necessary to memorize the steps involved. When deconvolving, all you must provide is the original convolution kernel (plus an additional inverted convolution kernel if the original kernel is not symmetrical).

Using Convolution Kernels

Now that you better understand the structure of kernels and the process behind convolution, it's time to actually use a few vImage convolution functions. This section shows you how to perform the emboss shown in Figure 2-1 (page 14), and also explains the differences between convolving with and without bias.

Convolving

vImage takes care of the specific convolution operations for you. Your job is to provide the kernel, or in other words, describe the effect that convolution should produce. Listing 2-1 shows how to use convolution to produce an emboss effect. To illustrate this effect, convolving the leftmost image in Figure 2-1 (page 14) produces the embossed image on the right. You could use the same code to produce a different effect, such as sharpening, by specifying the appropriate kernel.

Listing 2-1 Producing an emboss effect

```
int myEmboss(void *inData,
 unsigned int inRowBytes,
 void *outData,
 unsigned int outRowBytes,
 unsigned int height,
 unsigned int width,
 void *kernel,
 unsigned int kernel_height,
 unsigned int kernel_width,
 int divisor ,
 vImage_Flags flags )
{
    uint_8 kernel = {-2, -2, 0, -2, 6, 0, 0, 0, 0};                          // 1
    vImage_Buffer src = { inData, height, width, inRowBytes };               // 2
    vImage_Buffer dest = { outData, height, width, outRowBytes };            // 3
    unsigned char bgColor[4] = { 0, 0, 0, 0 };                               // 4
    vImage_Error err;                                                        // 5
```

```
    err = vImageConvolve_ARGB8888(    &src,     //const vImage_Buffer *src
                                      &dest,    //const vImage_Buffer *dest,
                                      NULL,
                                      0,    //unsigned int srcOffsetToROI_X,
                                      0,    //unsigned int srcOffsetToROI_Y,
                                      kernel,    //const signed int *kernel,
                                      kernel_height,     //unsigned int
                                      kernel_width,    //unsigned int
                                      divisor,    //int
                                      bgColor,
                                     flags | kvImageBackgroundColorFill
                                     //vImage_Flags flags
                                  );


    return err;
}
```

Here's what the code does:

1. Declares the emboss kernel as an `int` array. The data type of the kernel values should match the intended data type of the corresponding vImage function. Because the code sample calls `vImageConvolve_ARGB8888`, the kernel values should be unsigned, 8-bit integers (`uint_8`). vImage expects the kernel elements to be in the array in left-to-right, 1st row, 2nd row, 3rd row order (see Figure 2-2 (page 14) to see how this ordered).

2. Declares `vImage_Buffer` data structure for the source image information. Image data is store as an array of bytes (`inData`). The other members store the height, width, and bytes-per-row of the image. This data allows vImage to know how large the image data array is, and how to properly handle it.

3. Declares a `vImage_Buffer` data structure for the destination image information as done previously with the source image.

4. Declares a `Pixel8888`-formatted pixel to represent the background color for the destination image (black in this case).

5. Declares a `vImage_Err` data type to store the convolution function's return value.

After that, the code sends the declared values as parameters to the `vImageConvolve_ARGB8888` function where vImage handles the processing and stores the result in `dest`. The `vImageConvolve_ARGB8888` function is only one of several convolution functions available in vImage. vImage typically offers four variants of the same function—each for a different image format. The `ARGB8888` suffix lets you know this function is intended for interleaved (full-color) images, where each pixel is a grouping of four 8-byte integers representing the alpha (A), red (R), green (G), and blue (B) channels. For more details on the image formats vImage uses, refer to "Image Formats Available in vImage" (page 9).

In order for vImage to acknowledge the appropriate flags, the sample `myEmboss` function accepts a `vImage_Flags` parameter. This parameter represents a composite of one or more flags that the caller of `myEmboss` submitted. When the example then calls the vImage `vImageConvolve_ARGB8888` function, it passes those same flags, but this time combined with an additional specific flag constant, `kvImageBackgroundColorFill`. It does this by using the OR bitwise operator (`|`) to combine several flags

into single composition of flags that can be represented with one variable. This is the standard way to signal flags to vImage. This code example requests that vImage use the provided background image color (along with whichever flags were initially passed to the function).

To become familiar with kernel effects, try using the values from the following two kernels in your own code. Figure 2-6 shows the kernel for producing a Gaussian blur, and Figure 2-7 shows a Prewitt filter, which is a kernel for edge detection.

Figure 2-6 A kernel for Gaussian blur

1	2	1
2	4	2
1	2	1

Figure 2-7 A kernel for edge detection

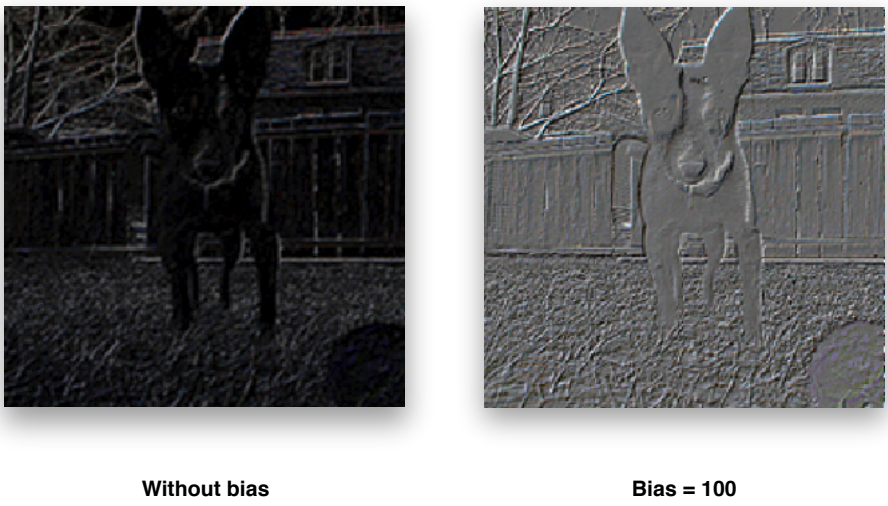
-1	-1	-1
0	0	0
1	1	1

Convolving with Bias

vImage gives you the option to perform convolution with bias or without. Bias is the value that you can add to each element in a convolution result to add additional influence from neighboring pixels. Since with certain convolutions it is possible to get negative numbers (which are not representable in a 0–255 format), bias prevents the signal from drifting out of range. You can choose to add a bias of 127 or 128 to allow some negative numbers to be representable (with an implicit +127 or +128 in their value). The overall effect of a bias brightens or darkens the image.

For every standard convolution function in vImage (such as `vImageConvolve_PlanarF`), there is a corresponding version that uses bias, indicated by "`WithBias`" in the function name (`vImageConvolveWithBias_PlanarF`, for example). Using the biased functions is virtually identical to the non biased functions, with the exception of the additional `bias` parameter you must pass to the convolution functions that use bias. The data type for the `bias` parameter must match that of the pixel data in the image. See *vImage Convolution Reference* for details on using these functions.

Figure 2-8 Bias versus No Bias



Without bias **Bias = 100**

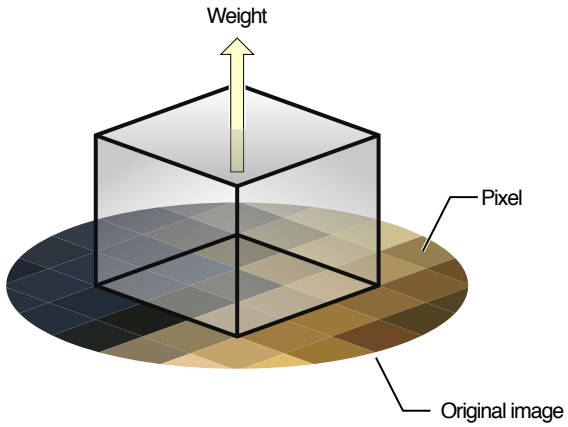
Using High-Speed Filters

vImage also offers functions for specific types of convolution that can be significantly faster than general convolution. Starting in Mac OS X v.10.4, you have the option to use the supplied box and tent filter functions for integer formats `Planar_8` and `ARGB8888`. These filters perform blur operations. The functions' names are from their shape when graphed in a cartesian coordinate system. They are equivalent to convolving with certain simple kernels, but you do not need to supply any kernel. These functions can be orders of magnitude faster than performing an equivalent convolution using custom kernels.

Note: vImage does not provide floating-point versions of these functions because the constant cost algorithm relies on exact arithmetic to be robust. Rounding differences in the floating-point computation can leave artifacts in the image in low intensity areas near high intensity areas.

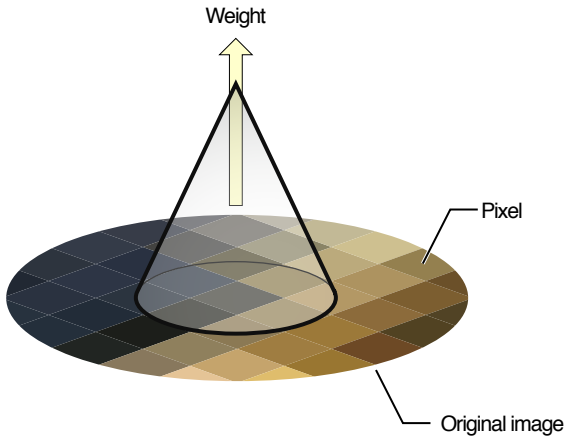
Box filters blur images by replacing each pixel in an image with the un-weighted average of its surrounding pixels. The operation is equivalent to applying a convolution kernel filled with all 1s. The box functions available in vImage are `vImageBoxConvolve_Planar8` and `vImageBoxConvolve_ARGB8888`. Each resulting pixel is the mean of surrounding pixels as defined by the kernel height and width.

Figure 2-9 Box filter



You use tent filters to blur each pixel of an image with the weighted average of its surrounding pixels. The tent functions available in vImage are `vImageTentConvolve_Planar8` and `vImageTentConvolve_ARGB8888`. These blur operations are equivalent to applying a convolution kernel filled with values that decrease with distance from the center. As with `vImageBoxConvolve_Planar8` and `vImageBoxConvolve_ARGB8888`, you do not need to pass any kernel to the function — just the height and width. Specifically, for an M x N kernel size, the kernel would be the product of an M x 1 column matrix and a 1 x 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.

Figure 2-10 Tent filter



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

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

and the second is

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

The product is

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

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

Using Multiple Kernels

vImage allows you to apply multiple kernels in a single convolution. The `vImageConvolveMultiKernel` functions allow for you to specify four separate kernels—one for each channel in the image. This allows for a greater level of control when applying filters to an image since you can operate on the red, green, blue, and alpha channels individually. For example, you can use multikernel convolutions to resample the color channels of an image differently to compensate for the positioning of RGB phosphors on the screen. Since each of the four kernels can operate on a single channel, the `vImageConvolveMultiKernel` functions are available only for interleaved image formats.

The use of these functions is identical to convolution with a single kernel, except that you must supply an array of pointers to the four kernels instead of one single kernel.

Deconvolving

As with the operations behind convolution, vImage takes care of performing the previously mentioned Richardson-Lucy algorithm for you. Your job is to provide the kernel (referred to as the point spread function) of the initial convolution. Listing 2-2 shows how to deconvole an emboss effect. You could use the same code to deconvolve various effects, such as blurring, by specifying the appropriate kernel. Note that unlike the convolution operations, there is an additional kernel parameter. This parameter can be `NULL` unless the height and width of the kernel are not the same size. If the kernel height and width are unequal, you must supply an identical kernel, but with its rows and columns inverted.

Listing 2-2 is an example of how to use vImage to deconvolve an `ARGB8888`-formatted image.

Listing 2-2 Deconvolving an embossed image

```
int myDeconvolve(void *inData,
 unsigned int inRowBytes,
 void *outData,
 unsigned int outRowBytes,
 unsigned int height,
 unsigned int width,
 void *kernel,
 unsigned int kernel_height,
 unsigned int kernel_width,
 int divisor,
 int iterationCount,
```

```
 vImage_Flags flags )
{
    //Prepare data structures
    uint_8 kernel = {-2, -2, 0, -2, 6, 0, 0, 0, 0};                          // 1
    vImage_Error err;                                                        // 2
    unsigned char bgColor[4] = { 0, 0, 0, 0 };                               // 3
    vImage_Buffer src = { inData, height, width, inRowBytes };               // 4
    vImage_Buffer dest = { outData, height, width, outRowBytes };            // 5



    //Send data to vImage for processing

    err = vImageRichardsonLucyDeConvolve_ARGB8888(    &src,                  // 6
                                        &dest,    //const vImage_Buffer *dest,
                                        NULL,
                                        0,    //unsigned int srcOffsetToROI_X,
                                        0,    //unsigned int srcOffsetToROI_Y,
                                        kernel,    //const signed int *kernel,
                                        NULL, //assumes symmetric kernel
                                        kernel_height,     //unsigned int
kernel_height,
                                        kernel_width,    //unsigned int
kernel_width,
                                        0, //height of second kernel
                                        0, //width of second kernel
                                        divisor,    //int
                                        0, //for second kernel
                                        bgColor,
                                        iterationCount, //uint32_t
                                        kvImageBackgroundColorFill | flags
                                        //vImage_Flags
                                    );


    //Report result
    return err;                                                              // 7
}
```

Here's what the code does:

1. Specifies the kernel used in the original convolution that vImage should deconvolve. The example uses a symmetrical kernel (both the height and width are 3), and therefore it isn't necessary to supply a second kernel to vImage.

2. Declares the `vImage_Error` structure to capture the result of the deconvolution function.

3. Declares a `Pixel8888`-formatted pixel to represent the background color for the destination image (black in this case).

4. Declares a `vImage_Buffer` data structure for the source image information. Image data is received as an array of bytes (`inData`). The other members store the height, width, and bytes-per-row of the image. This data allows vImage to know how large the image data array is, and how to properly handle it.

5. Declares a `vImage_Buffer` data structure for the destination image information as done previously with the source image.

6. Supplies the previously declared data structures to vImage for processing. Note how the example supplies the `vImageRichardsonDeConvolve_ARGB8888` function with a `NULL` parameter for the second kernel. This is because the first kernel is symmetric, and therefore inverted kernel is not necessary.

7. Returns the same `vImage_Error` data structure returned from deconvolution.

Deconvolution is an iterative process. You can specify how many times that you would like vImage to perform the deconvolution. Depending on how many iterations you choose, the image result may vary, so it may be worth experimenting with several iterations to see which one yields ideal results.

Performing Geometric Operations

Unlike convolutions, which enhance or filter images, geometric operations distort the geometry of images. Image rotation, scaling, and resizing are examples of geometric operations. They involve changing the shape of an image by modifying its geometric properties, such as the height, width, and the angles in its form.

Developers who are in search of efficient ways to alter the geometric properties of images in real time will find these vImage operations useful. In particular, vImage's geometric operations are well suited for:

- Applying arbitrary rotations to large images in real time
- Changing the perspective of large images in real time
- Scaling large images efficiently

vImage's geometry transforms usually look better than what you get from Quartz 2D or Core Image since vImage uses Lanczos resampling methods by default (rather than simpler approaches like linear interpolation). This makes vImage's transforms less susceptible to Moire patterns and offer better image fidelity.

This chapter describes the various classes of geometric operations vImage offers, and explains how to use them. By reading this chapter you'll:

- See the types of effects geometric operations can produce
- Learn how resampling works and why it can be necessary when performing geometric operations
- Find out, through code samples, how to apply vImage geometric operations to images

Geometric Operations Overview

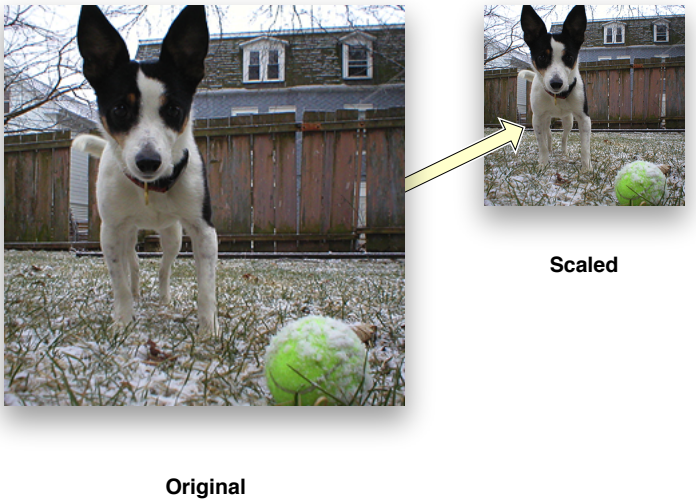
Although often simple conceptually, geometric operations (such as rotating an image by an arbitrary angle) can be computationally expensive. vImage provides both high-level and low-level functions for performing these operations efficiently. vImage offers several types of geometric operations, each meant to distort the geometry of the initial image in a different way.

Here are some of the geometric operations available in vImage:

- Affine warp
- Horizontal reflect
- Vertical reflect
- Horizontal shear
- Vertical shear
- Rotation
- Scale

"Affine Warp" shows one of the available scaling functions.

Figure 3-1 Scaling



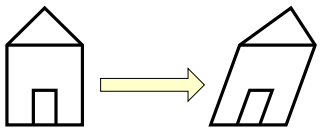
This function scales images in `ARGB8888` format:

```
vImageScale_ARGB8888( inBuffer, outBuffer, NULL, flags );
```

The function scales the image by comparing the height and width of the original image (passed through the `inBuffer` parameter) and the height and width of the destination image (passed through the `outBuffer` parameter).

All of the geometric operations that vImage provides are examples of affine transformations. An affine transformation is the most general of linear transformations on an image. Under this transformation, all parallel lines in an image remain parallel. Figure 3-2 shows a basic affine transformation (notice how all parallelism is maintained).

Figure 3-2 An affine transformation



Affine transformations work by mapping each pixel in the source image (based on its [*x*, *y*] coordinate location) to a new location [*x′*, *y′*] in the destination image based on this simple arithmetic formula:

$$x' = a*x + b*y + c$$
$$y' = d*x + e*y + f$$

Affine transformations can also be described in terms of matrix arithmetic. The transformation is determined by an affine transform matrix

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

according to the formula

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

tx and *ty* are each symbols for individual values. The additional component "1" on the right side is not part of the location of the source pixel; it is added here 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 x 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 the same as the `CGAffineTransform` data type. See *CGAffineTransform Reference* for the functions used for creating and manipulating matrices.

Unlike convolutions, geometric operations distort the geometry of images. As a result, the number of pixels in the output image might differ from the number of pixels in the input image. Take, for example, the affine warp shown in Figure 3-3.

Figure 3-3 Affine warp

Original | Affine warp

In this example it's clear that area of the output image is different from the original image because the height, width, and angles have been changed. When an image processing function is unable to maintain a one-to-one mapping between each pixel in the source image and each pixel in the destination image, it must use something called a resampling filter to get the desired pixel values in the destination image. You will learn how vImage handles resampling filters in greater detail in the next section.

Resampling

Changing the dimensions of a digital image (as done in scaling and rotation) can potentially create unintended features to appear in the image. Many of these features occur when trying to evaluate pixels at fractional pixel locations. Most of the vImage geometric functions use a process known as **resampling** in order to avoid creating artifacts, such as interference patterns, in the destination image. vImage performs resampling 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 convolution.

However, in the geometric operations, the resampling kernel must itself be resampled during the process of pairing kernel values against the sampled pixel data. vImage must evaluate the kernel at fractional pixel locations ("in-between" pixels), 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. The default resampling filters are implementations of the commonly used Lanczos resampling method. The Lanczos resampling method usually produces better-looking results than simpler approaches such as linear interpolation. However, the Lanczos method can produce ringing effects near regions of high frequency signals (such as line art). vImage provides two variations to choose from. If you do not set the `kvImageHighQualityResampling` flag for the call, vImage uses a `Lanczos3` filter. If you do set the flag, vImage uses a `Lanczos5` filter. A `Lanczos5` filter does higher-quality resampling than a `Lanczos3` filter but is slower to use.

The `Rotate` functions use resampling. The `Rotate90` functions do not.

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.

Listing 3-1 Creating a default resampling filter for a shear function

```
ResamplingFilter filter = vImageNewResamplingFilter( float scale, vImage_Flags
 kvImageNoFlags )
```

The procedure is different if you want to provide a custom resampling filter. Because the shear functions operate on a line-by-line basis, a resampling kernel used with a shear function is one-dimensional, a 1 x 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 must 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 Resampling Filter

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` (as previously seen in Listing 3-1 (page 29)). 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 same `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. See *vImage Geometry Reference* for the details of usage. 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` objects include a scale factor that you provide. This scale factor is used to scale the transformed image in the shear operation.

A Sample Custom Resampling Filter

Listing 3-2 shows how to declare and use a custom resampling filter function. 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`.

Listing 3-2 Using a custom filter function

```
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;
}
```

Frequently Used Operations

Rotations and scaling are the most common geometric operations, and vImage's scaling and rotation functions are particularly suited for developers that need these scalings to look as clean as possible. Applications that manage photo layout (like iPhoto's digital photo books) are perfect candidates for this. Listing 3-3 shows how to rotate an image that is in `ARGB8888` format.

Listing 3-3 Rotating an image by a specified angle using vImageRotate_ARGB8888

```
int MyRotateFilter(void *inData, unsigned int inRowBytes, void *outData, unsigned
 int outRowBytes, unsigned int height, unsigned int width, void *kernel, unsigned
 int kernel_height, unsigned int kernel_width, int colorChannel, vImage_Flags
flags )
{
    vImage_Buffer in = { inData, height, width, inRowBytes };
    vImage_Buffer out = { outData, height, width, outRowBytes };
    TransformInfo *info = kernel;
    float angle = info->rotate * 2.0f * M_PI / 360.0f;
    Pixel_8888 backColor;

    backColor[0] = UCHAR_MAX * info->a;
    backColor[1] = UCHAR_MAX * info->r;
```

```
    backColor[2] = UCHAR_MAX * info->g;
    backColor[3] = UCHAR_MAX * info->b;

    return vImageRotate_ARGB8888( &in, &out, NULL, angle, backColor, flags );
}
```

Rotation

When performing rotation, vImage maps the center point of the source image 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. Figure 3-4 shows a photo of a bluejay that is rotated, but portions of the image appear clipped since they now reside outside the image buffer (these regions are colored with a caller-supplied background color).

Figure 3-4 Rotated image with clipping

Original **Rotated**

90 Degree Rotation

This rotation operation rotates a source image by either 0, 90, 180, or 270 degrees (depending on a value you supply). vImage maps the center point of the source image 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 background color you supply) may appear in the destination image (as seen in Figure 3-4 (page 31)).

Because no resampling is done—instead, individual pixels are copied unchanged to new locations—this function places certain restrictions on the pixel heights 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 an image 90 or 270 degrees, the height in pixels 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.

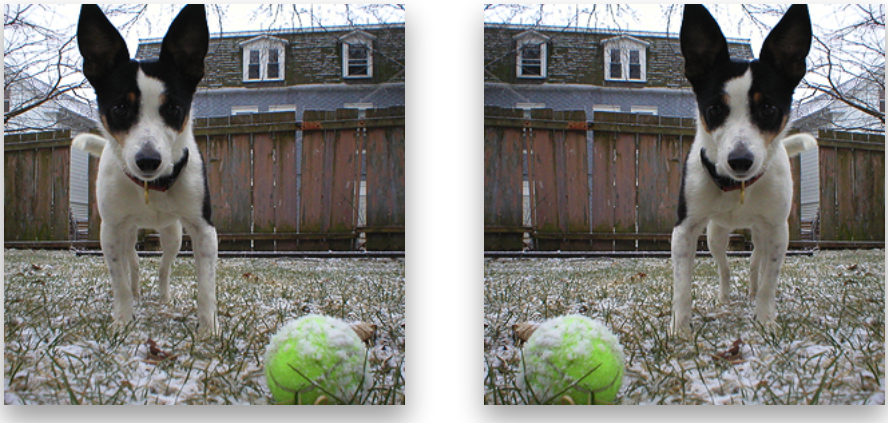
- If you are rotating an image 0 or 180 degrees, the height in pixels of the source image and the destination image must both be even or both be odd; and the width of the source image and the destination image must both be even or both be odd.

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

Horizontal Reflections

A horizontal reflection creates a mirror image of the input image by reflecting about the y-axis of the image. This can be thought of as placing an imaginary line from the top of the image to the bottom, and swapping each pixel on one side with the corresponding pixel on the other side. Figure 3-5 shows an example of a horizontal reflection operation.

Figure 3-5 Horizontal reflect

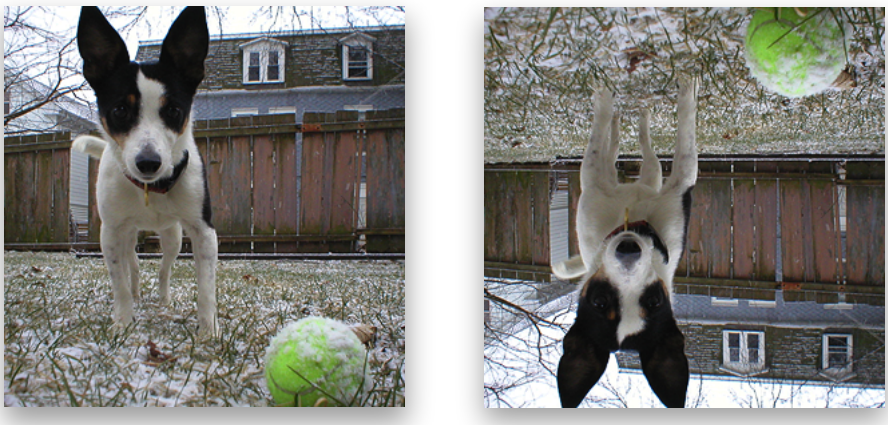


Original **Horizontal reflect**

Vertical Reflections

A vertical reflection is identical to a horizontal reflect, except it reflects about the x-axis instead. Figure 3-6 shows an example of a vertical reflection operation.

Figure 3-6 Vertical reflect



Original | Vertical reflect

Horizontal Shear

vImage's horizontal shear function does shearing, translation, and scaling (in the horizontal dimension only) on a source image. The shear functions (and `Rotate90`) serve as the foundation upon which many of vImage's other functions are implemented. 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.

The operation starts at the bottom of the rectangle and shifts 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 you to specify the resampling filter. It can either be a default supplied by vImage—a Lanczos kernel—or a custom resampling filter you provide. In either case you pass a `ResamplingFilter` object to the horizontal shear function. The `ResamplingFilter` object you supply must contain the information necessary to calculate the resampling filter. It also contains a scale factor, which scales the transformed image in the horizontal direction.

In addition to simple shearing and scaling, the vImage horizontal shearing function allows you 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.

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 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.)

Figure 3-7 shows the photo after a horizontal shear operation with no translation component.

Figure 3-7 Horizontal shear

Original **Horizontal shear**

Vertical Shear

vImage's vertical shear function does shearing, translation, and scaling (in the vertical dimension only) on a source image. 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.

The vertical shear starts at the left edge of the rectangle and shifts 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 you to specify the resampling filter. It can either be a default filter supplied by vImage—a Lanczos kernel—or a custom resampling filter supplied by the user. In either case you pass a `ResamplingFilter` object to the vertical shear function. The `ResamplingFilter` object you supply must contain the information necessary to calculate the resampling filter. It also contains a scale factor, which scales the transformed image in the vertical direction.

In addition to simple shearing and scaling, the vImage vertical shearing function allows you 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 outside areas are assigned colors using either the background color fill technique or the edge extend technique, depending on a flag setting.

Figure 3-8 shows a picture after a vertical shear operation.

Figure 3-8 Vertical shear

Original | Vertical shear

Affine Warp

Using functions such as `vImageAffineWarp_ARGB8888` you can perform customized affine transformations that combine the effects of various rotations, shears, and reflections into a single function call. You do this by first constructing a `vImage_AffineTransform` data type which represents your affine transformation matrix. You can then pass this as a parameter (along with the rest of your image data) to the various affine warp functions vImage offers.

Performing Morphological Operations

Morphological operations alter the intensities of specific regions of an image. Unlike the convolution or geometric operations which affect an entire image, morphological operations isolate certain sections of an image (acknowledging them as either foreground or background) and then expands or contracts just those regions to achieve a desired effect.

Developers who are looking for efficient ways to enhance or isolate qualities of an image that are found in either the foreground or the background of an image will find the morphological operations in vImage useful. For example, you can apply morphological operations to scientific images of the terrain of Mars to perform topological analyses that isolate the craters and valleys along the planet's surface. In general, morphological operations are well-suited for:

- Isolating background features from foreground features in an image (and vice-versa)
- Performing feature detection
- Performing motion planning of an object along a surface with visible obstacles

This chapter describes the basics of image processing using morphology. By reading this chapter, you'll:

- See the types of tasks morphological operations can perform
- Learn how images and objects differ, and how vImage operates upon objects
- Find out, through code samples, how to apply morphological operations to images

Objects

The target of vImage morphological operations differs from that of the other types of operations found in vImage. Instead of applying an operation to an entire image, vImage applies morphological operations to an **object**. An object is composed of either the brightest pixels in an image or the darkest pixels in the image, where brightness is defined relative to the particular image.

Figure 4-1 shows objects outlined in an image.

Figure 4-1 Example of objects in an image

Morphological functions change the shape of an object by performing dilation, erosion, maximum, and minimum operations. Dilation expands objects. Erosion contracts them. Maximum is a special case of dilation, while minimum is a special case of erosion. As with convolution, the precise nature of the expanding or shrinking is determined by a kernel the calling function provides. The number of rows and number of columns of the image does not change after applying a morphological operation.

When you define bright pixels as the object, dark pixels become the background. In this case dilation expands objects with erosion contracts them. When you define dark pixels as the object, bright pixels become the background. In this case, dilation contracts objects and erosion expands them.

Figure 4-2 Examples of object dilation and erosion

You can use morphological functions on grayscale images, where the source image is planar (single-channel) or on full-color images. The kernel itself (explained in more detail in the following section) is always planar.

Kernels

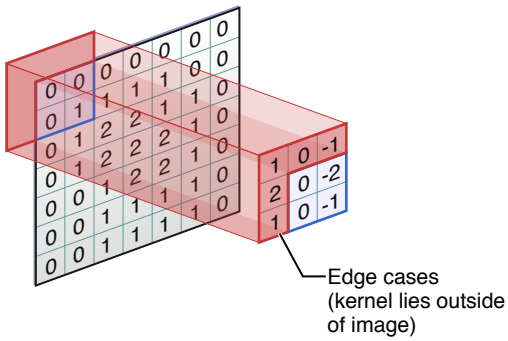
Each morphological function requires that you pass it a convolution kernel that determines how the values of neighboring pixels are used to compute the value of a destination pixel. A kernel is a packed array, without padding at the ends of the rows. See also "Convolution Kernels" (page 13). The elements 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 x 3 convolution kernel for a `Planar8` image consists of an array of nine 8-bit (1-byte) values, arranged consecutively. The first three values represent the first row of the kernel, the next three values the second row, and the last three values the third row.

Morphology functions perform 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 elements. This scenario is known as an **edge case**, as illustrated in Figure 4-3.

Figure 4-3 Example of an edge case



In this case the morphology functions make use of only that part of the kernel which overlaps the source buffer. The other kernel elements are ignored.

vImage performs morphological operations on full-color images in the following way:

- It separates the image into four planar images each (one planar image for each of alpha, red, green, and blue)
- It applies the desired morphological operation to each color plane separately, treating the planar images as grayscale. (If the `kvImageLeaveAlphaUnchanged` flag is set, the morphological operation is not performed on the alpha channel.)
- It recombines the results into a full-color image.

Operation Types

There are three main types of morphological operations: dilation, erosion, and maximizing/minimizing.

- 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 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.

  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). The following is an example of how to use a dilation filter for an image in `ARGB8888` format.

Listing 4-1 Dilation filter example

```
int MyDilateFilter(void *inData, unsigned int inRowBytes, void *outData, unsigned
 int outRowBytes, unsigned int height, unsigned int width, void *kernel, unsigned
 int kernel_height, unsigned int kernel_width, int divisor, vImage_Flags flags
 )
{
    vImage_Buffer    in = { inData, height, width, inRowBytes };
    vImage_Buffer        out = { outData, height, width, outRowBytes };
    const int        xOffset = 0;
    const int        yOffset = 0;

    return  vImageDilate_ARGB8888( &in, &out, xOffset, yOffset, kernel,
kernel_height, kernel_width, flags );
}
```

- Erosion

  Erosion of a source image (region of interest) by a kernel is defined similarly to dilation:

  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.

Listing 4-2 shows how to erode an image.

Listing 4-2 Erosion filter example

```
int MyErodeFilter(void *inData, unsigned int inRowBytes, void *outData, unsigned
 int outRowBytes, unsigned int height, unsigned int width, void *kernel, unsigned
 int kernel_height, unsigned int kernel_width, int divisor, vImage_Flags flags
 )
{
    vImage_Buffer    in = { inData, height, width, inRowBytes };
    vImage_Buffer        out = { outData, height, width, outRowBytes };
    const int        xOffset = 0;
    const int        yOffset = 0;

    return  vImageErode_ARGB8888( &in, &out, xOffset, yOffset, kernel,
kernel_height, kernel_width, flags );
}
```

Instead of expanding objects, erosion tends to spread dark pixels around, causing them to eat away (erode) at objects. The morphological operation known as 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 directly performed by vImage. In vImage, you can accomplish an open operation by using 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 you can perform the open operation by using a min operation followed by a max operation, and the close operation by using a max operation followed by a min operation.

Performing Histogram Operations

Histogram operations calculate histograms of images or manipulate a histogram to modify an image. A **histogram** is a statistic that shows the frequency of a certain occurrence within a data set. In the graphics domain, histograms can be used to plot the frequencies of certain pixel intensities.

Developers who are interested in the following will find histograms useful:

- Image calibration
- Image contrasting
- Normalizing image intensity

Histogram Operations Overview

There are a number of reasons to apply histogram operations to an image. 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 details difficult to see. Changing the image 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 you change each histogram to match some standard histogram. 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. Two pixels of the same intensity always map to two pixels of the same (but presumably altered) intensity. If the original image has N different intensity values, the transformed image will have at most N different intensity levels represented.

The vImage histogram functions either calculate histograms or perform one of these point operations:

- Contrast stretching transforms 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.
- Ends-in contrast stretching is a more complex version of the contrast stretch operation. These types of functions 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 two given intensity values do not directly define the low and high levels; percentages do: 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.
- **Histogram equalization** transforms an image so that it has a more uniform histogram. A truly uniform histogram is one in which intensity level occurs with equal frequency. These functions approximate that histogram. Figure 5-1 (page 44) is an example of equalization.

Figure 5-1 Example of equalization

Original | Histogram equalization

- **Histogram specification** transforms an image so that its histogram more closely resembles a given histogram. This is a method of image calibration. You can make two images have the same tonal qualities by changing one image to conform to the histogram of another. This may help them to look like they were shot on the same day with the same film and light conditions.

Using Histogram Operations

Listing 5-1 shows how you can apply an equalization operation to a `Planar8` image.

Listing 5-1 Histogram equalization example

```
int MyEqualization(void *inData, unsigned int inRowBytes, void *outData, unsigned
 int outRowBytes, unsigned int height, unsigned int width, void *kernel, unsigned
 int kernel_height, unsigned int kernel_width, int divisor, vImage_Flags flags
 )
{
    vImage_Error err;                                                          // 1
    vImage_Buffer src = { inData, height, width, inRowBytes };                 // 2
    vImage_Buffer       dest = { outData, height, width, outRowBytes };        // 3

    err = vImageEqualization_Planar8(    &src,
                                        &dest,
                                        flags
                                    );                                         // 4
    return err;                                                                // 5
}
```

Here's what the code does:

1. Declares a `vImage_Err` data type to store the equalization function's return value.

2. Declares a `vImage_Buffer` data structure for the source image information. Image data is stored as an array of bytes (`inData`). The other members store the height, width, and bytes-per-row of the image. This data allows vImage to know how large the image data array is so that vImage knows how to properly handle it.

3. Declares a `vImage_Buffer` data structure for the destination image information as it did previously the source image.

4. Calls vImage's equalization function an stores the result in the `vImage_Error` data type it previously declared.

5. Passes any potential error codes up to the calling function.

Common Applications

Histograms are useful for calibrating images obtained from different sources. Say, for example, you have images that are oversatured in a specific channel. You can apply a histogram specification at calibrate the image to a histogram that meets your goals.

In the field of scientific imaging, histograms can be useful in interpreting subtle qualities of an image. For example, for a graphic of a thermodynamic reading, imagine how useful it would be to analyze the histogram and tell how frequently colors of a certain intensity occur. You would be able to determine how "warm" or "cold" a region was based on the values in the histogram.

In general, histogram operations save you time when you need to analyze pixel intensity data from an image, or get an image to conform to a specific color setting.

Performing Alpha Compositing Operations

Alpha compositing is a realm of image processing operations that blend two or more images to create a final composite image. The **alpha channel** defines the degree to which the various images are blended.

Quartz 2D and most modern video cards do an excellent job of alpha compositing. vImage provides these functions for completeness. Developers who want to do image compositing should understand the principles of alpha compositing.

This chapter describes how to use the alpha compositing functions in vImage. By reading this chapter, you'll:

- Learn the basics of alpha compositing with vImage
- Learn the difference between premultiplied and non-premultiplied alpha compositing
- Learn to convert back and forth between premultiplied and non-premultiplied alpha formats

Alpha Compositing

Alpha compositing is a common image processing routine used to blend two or more images to create a final composite image. Alpha compositing is built upon the concept of layers — each image used in the composite image has a certain hierarchical layer. The image's alpha channel determines how much of the images in layers underneath it can be seen at its own layer. In other words, the alpha channels control the transparency of the image, and alpha compositing uses the alpha channel to appropriately blend this image with another to exhibit this transparency. Figure 6-1 (page 47) illustrates this concept.

Figure 6-1 Example of composited layers

Premultiplied Versus Non-Premultiplied Alpha Compositing

As with the commonly-used red, green, and blue channels, the alpha channel is a 2D array of pixel intensities. Depending on the image's pixel format, its intensities may span the ranges of 0 to 255 (integers), or 0 to 1 (floats). In a **premultiplied alpha composite**, the values of the the alpha channel are multiplied to each of the of color channels, which alleviates the need to process the alpha channel any further. In a **non-premultiplied alpha composite**, the alpha channel still needs to be composited.

Performing Image Transformation Operations

Image transformation operations alter the values of pixels in an image as defined by custom functions provided as a callback. Unlike convolution operations, transformation functions do not depend on the values of neighboring pixels.

Developers who are interested in efficiently performing gamma corrections or custom pixel manipulation functions on large or real-time image will find these functions useful. With vImage, you can use image transformation operations on the following sorts of functions:

- Gamma correction functions
- Functions that use lookup tables to determine destination pixel value
- Matrix multiplication functions
- Piecewise functions that allow you provide custom polynomial functions

This chapters describes the principles of image transformation and shows you how to use the functions provided by vImage. By reading this chapter you will learn how to:

- Correct images using pre-defined gamma functions
- Transform an image using a lookup table
- Apply polynomials and matrix multiplication to an image

Transformation Operations

Image transformation functions fall into four categories:

- Gamma correction functions correct the brightness profile of an image by multiplying each pixel by the value of the function. Gamma correction prepares an image for display or printing on a particular device.
- Lookup table functions are like the piecewise polynomial functions, but instead of applying a polynomial they use a lookup table that you supply.
- Matrix multiplication functions have a variety of uses, such as to convert between color spaces (RGB and YUV, for example), change a color image to a grayscale one, and perform "colortwisting."
- Piecewise functions are similar to the gamma correction functions, but instead of applying a predefined gamma function they apply one or more polynomials that you supply. The number of polynomials must be an integer power of 2, and they must all be of the same order.

Transformation functions use a `vImage_Buffer` structure to receive and supply image data. This buffer contains a pointer to image data, the height and width (in pixels) of the image data, and the number of row bytes.

Some transformation functions work in place. That is, the source and destination images can occupy the same memory if they are strictly aligned pixel for pixel. For these, you can provide a pointer to the same `vImage_Buffer` structure for one of the source images and the destination image.

Gamma Correction

Gamma corrections have to do with changing the intensities of the pixels in an image such that it corrects for the eye's uneven response to certain colors in an image display. The overall goal of gamma correction is to ensure an accurate depiction of an image given its display's limited signal range, (such as the number of bits in each pixel).

When vImage performs a gamma correction on an image, it takes each pixel of the source image and passes its intensity through a pre-defined gamma function, and saves this resultant (gamma-corrected) pixel intensity in the destination image.

Using Lookup Tables

You can use a lookup table to apply custom image transformations to an image. By providing an array of 256 integers (for integer pixel types) or 4096 values representing the intensity of floating-point pixel types, you define the destination pixel intensity for each potential source pixel intensity. When vImage transforms an image according to a lookup table, it takes each pixel intensity in the image, indexes into the corresponding array element for that intensity, and saves the intensity stored at that position in the array as the destination pixel intensity.

Listing 7-1 (page 50) shows how to use a lookup table to apply a custom image transformation.

Listing 7-1 Transforming an image using a lookup table

```
int MyLookup(void *inData, unsigned int inRowBytes, void *outData, unsigned int
 outRowBytes, unsigned int height, unsigned int width, void *table, unsigned
int table_height, unsigned int table_width, int divisor, vImage_Flags flags )
{
    vImage_Buffer    in = { inData, height, width, inRowBytes };              // 1
    vImage_Buffer    out = { outData, height, width, outRowBytes };           // 2

    if( table_height != 1 || table_width != 256 )                             // 3
        return kvImageInvalidKernelSize;

    return vImageTableLookUp_Planar8(&in, &out, table, flags);                // 4

}
```

1. Declares a `vImage_Buffer` data structure for the source image information. Image data is received as an array of bytes (`inData`). The other members store the height, width, and bytes-per-row of the image. This data allows vImage to know how large the image data array is, and how to properly handle it.

2. Declares `vImage_Buffer` data structure for the destination image information as done previously with the source image.

3. Checks to make sure that size of the lookup table is suitable for the `Planar8` pixel type.

4. Performs the actual vImage function call and passes up potential errors.

Using Matrix Multiplication

Using matrix multiplication is similar to using a lookup table to determine the destination pixel intensity, except that vImage multiplies each pixel intensity by a specific matrix that you provide. vImage multiplies each pixel intensity by the same matrix to perform this type of transformation. The format of the matrix is a 1D array of 4x4 `int16_t` data types for integer pixel types, and a 1D array of 4x4 `float` data types for float pixel types.

vImage provides the following functions for performing pixel-matrix multiplication:

- `vImageMatrixMultiply_Planar8`
- `vImageMatrixMultiply_PlanarF`
- `vImageMatrixMultiply_ARGB8888`
- `vImageMatrixMultiply_ARGBFFFF`

Using Polynomials

You can provide a polynomial function for vImage to apply to each pixel intensity of a given image. You define the polynomial by deciding upon its **order**, and then providing an array of **coefficients** (with order - 1 coefficients). You also need to decide the boundary values for separating adjacent ranges of pixel values. This determines how vImage should truncate and/or clip the pixel intensities. You can use polynomials to serve as approximations for other functions that are to expensive to calculate. Imagine a polynomial curve fitting in a data plotting application, except that you are fitting a polynomial to a continuous function rather than a series of discrete data. You can usually create a polynomial with some number of terms that closely matches the function that you want (e.g. sine, power, etc.) over a limited range.

Additionally, vImage supports using piecewise polynomials to transform an image. In order to fit a particular curve over the entire range of possible input pixel values, it is sometimes necessary to use multiple polynomials that fit contiguous subregions of the expected input gamut. For example, if you expect all pixels to be in the range [0, 1.0], you might have one polynomial good for inputs in the range [0, 0.5], and one for [0.5, 1.0]. You may have any number of polynomials, so long as that number is an exact power of two. in order to enforce this requirement, vImage receives this value as the `log2segments` parameter — the polynomial count. For example, if you say that your function is the head-to-tail concatenation of eight polynomials, the number you should pass in the `log2segments` parameter should be 3 (because $\log_2(8) = 3$).

vImage provides the following functions for using piecewise polynomials to transform images:

- `vImagePieceWisePolynomial_PlanarF`
- `vImagePieceWisePolynomial_Planar8ToPlanarF`
- `vImagePieceWisePolynomial_PlanarFToPlanar8`
- `vImagePiecewiseRational_PlanarF`

Best Practices for Using vImage

This chapter gives guidelines for getting optimal performance from vImage. It covers the following best practices:

- Make sure the system your application runs on meets the basic requirements
- Use the Image I/O framework to load images into memory
- When possible, use planar image formats
- Take advantage of tiles
- Align data buffers
- Reuse buffers
- Use threads appropriately
- Separate 2D kernels into multiple 1D kernels

Loading Image Data

The first step to integrating vImage into your own applications is getting raw image data loaded into memory. You can use the Image I/O framework to load images of any major image file format (.JPG, .PNG, .GIF) into C-style buffers (void * arrays).

Below is an example of how to extract raw image data from a local file. If you would like to know more about the other methods and options available in Image I/O, see *Image I/O Programming Guide*.

```
NSURL* url = [NSURL fileURLWithPath:filename];
//Create the image source with the options left null for now
//Keep in mind since we created it, we're responsible for getting rid of it
CGImageSourceRef image_source = CGImageSourceCreateWithURL( (CFURLRef)url, NULL);
if(image_source == NULL)
{
    //Something went wrong
    fprintf(stderr, "VImage error: Couldn't create image source from URL\n");
    return false;
}
//Now that we got the source, let's create an image from the first image in the
 CGImageSource
CGImageRef image = CGImageSourceCreateImageAtIndex(image_source, 0, NULL);

//We created our image, and that's all we needed the source for, so let's release
 it
CFRelease(image_source);

if(image == NULL)
{
```

```
    //something went wrong
    fprintf(stderr, "VImage error: Couldn't create image source from URL\n");
    return false;
}
```

After you've loaded the images into the buffers, they are ready to be passed to vImage functions. Pay close attention to the character sequence that follows the underscore in the function name—this is the format that it expects the pixel data to match. vImage functions can either work in-place (the output is in the same buffer that was passed as input) or they use a destination buffer that you can supply.

Since vImage handles only the image processing, you need to look to another technology to actually display the image. Depending on the goal of your application, or the application environment you used (Carbon or Cocoa), you may have to find a way of displaying the resultant pixel data (Quartz, for example), or saving the image data to disk (Image I/O).

Use Planar Image Formats

Most vImage functions come with four image format variants, (one for each of the image formats understood by vImage). Planar images encode one single channel at a time (e.g. all the byte data for the red channel is stored consecutively, then all for the green channel, then the blue, then alpha, and so on), instead of mixing the bits of all channels throughout the in-memory representation of the image (interleaved).

Since most vImage functions have to separate by channel the bits of the images you pass to it anyway (thus putting it into a planar format), it frequently makes sense to do this ahead of time. Since vImage usually works only on one channel at a time anyway, having your image data grouped by channel saves you the time that would usually be spent deinterleaving and reinterleaving each pixel. So in general, use planar image formats as much as possible.

Note: In some cases you may not even need all red, green, blue, and alpha channels. For example, you may know beforehand that the alpha channel is irrelevant in the images that you're dealing with, or perhaps all of your images are grayscale and thus can use only one channel. Using planar formats makes it is possible to isolate only the channels you need.

Take Advantage of Tiles

Tiling is a technique commonly used in graphics applications that takes a large image and breaks it into several, smaller images. This is called tiling because much like how floor tiles can be placed together to create a larger floor, small subunits of an image can be stitched together to form a larger image. The benefit behind this method is that CPUs tend to handle data a lot faster when it fits in their high-speed data caches.

In general, vImage functions have much better performance when the data they process (including input and output buffers) fit in processor data caches. Data stored in the CPU caches can be accessed a lot faster than data stored in main memory. While CPU cache memory is fast, it is also limited in space. Cache size varies from processor to processor, but in general it's good to keep image tile sizes below 2 MB for Intel processors and below 512 KB for PowerPC processors.

- Here are some tips for tiling:
  - Some caches can only hold a small amount at a time (usually 512 KB or less)
    - Tile sizes of 128 KB - 256 KB give overall best throughput

- Many vImage functions use tiling (along with multithreading)internally. If you want to control tiling yourself, set the `kvImageDoNotTile` flag in the flags parameter when you call a function, which prevents the function from using tiling or multithreading internally.
- For square tiles, a tile size of 128 KB to 256 KB gives the best overall throughput.
- Which functions tile or multithread 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. You may also need the `kvImageDoNotTile` flag if you are concerned about vImage blocking on a condition variable while worker threads do the work. This may be required to prevent a priority inversion in real time code.
- The best tile size varies according to function. Functions with very little computation per byte (mostly conversion functions) are fastest with tiles smaller than 16 KB. Typical vImage functions do best with 256 KB tiles. Tile size is less important for computation-heavy functions (such as those that use multithreading).

Align Data Buffers

When allocating floating-point data for images, it's important to keep the data 4-byte aligned. This means that the amount of bytes that you allocate should be an integer multiple of 4.

Here are some tips about data alignment and buffer sizes:

- Though vImage tolerates lesser alignments, for best performance, everything should be 16-byte aligned and `rowbytes` should be a multiple of 16 bytes.
- Floating-point data must be at least 4-byte aligned or some functions will fail.
- The value you pass in the `rowBytes` parameter to a function should not be a power of 2.

Reuse Buffers

Many vImage functions use temporary buffers to hold intermediate values when performing a task. Creating this buffer once initially, and supplying it to the various functions can save time depending on how frequently you call the functions.

If you do not provide a buffer, these functions allocate memory for themselves (and, of course, deallocate it when they are finished).

If you are going to call the function only 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.

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

If possible, applications should also try to reuse the regular image buffers that the `data` field of `vImage_Buffer` data type points to. This saves time otherwise spent reallocating and zero-filling the buffer.

In order to facilitate real-time usage, vImage avoids, as much as possible, usage of the heap and other operations that block on a lock, such as memory allocation

Thread Appropriately

vImage is thread-safe and can be called reentrantly. If you tile your image, you can use separate threads for different tiles. If you use different processors to handle different tiles, you should choose tiles that are not horizontally adjacent to each other. Otherwise the tile edges may share cache lines, potentially resulting in time-consuming crosstalk 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.

In Mac OS X v10.4 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, these are most functions in `Geometry.h`, and the gamma functionality.) 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 functions reentrantly.

Thread-safe functions use locks to maintain data coherency. If you don't want functions to use locks, you may prevent vImage from multithreading and tiling by passing the `kvImageDoNotTile` flag. If you use this flag, your application is responsible for tiling its own data and doing its own multithreading.

Separate 2D Kernels into 1D Kernels

If you're using convolution to apply filters to images, you can sometimes gain a performance boost by splitting the two-dimensional kernel into multiple one-dimensional kernels, and then applying the convolution twice (once per dimension).

You can, of course, pass the 2D kernel to one of the `vImageConvolve` functions. vImage uses the 2D kernel to perform nine multiply and eight add operations to compute each pixel result. To get better performance, call the `vImageConvolve` function twice, once for each of the 1D convolve filters. When separated, vImage performs three multiply and two add operations per pixel per convolve pass, for a total of six multiply and four add operations. You'll notice that separating the convolution kernel into multiple passes, there is an algorithmic savings of one third of the multiply operations and half the add operations. For a M x N kernel, the processing cost is roughly reduced from M*N to M+N when kernels are separated. For larger kernels, the savings becomes more dramatic. A 5 x 5 kernel might be 2.5 times faster when separated, and a 11x11 kernel might be over 5 times faster!

Keep in mind that this technique may be slower in routines where the cost of traversing the image is higher than the arithmetic involved. This typically happens when the images are very large, or they don't fit into physical RAM.

There are cases involving very large filters where separating the filter may be the only way to perform the convolution operation. A very larger filter that sums to a value larger than 2^{24}, over its entirety or any part, runs the risk of overflowing the accumulators that vImage uses for 8-bit vImage convolution operations. In such cases, separating the kernel is likely to allow you to avoid the overflow. You can even use this to add more fixed-point precision into your filter, by scaling the filter values to be larger. The extent to which the loss of precision from the intermediate rounding offsets this advantage is unknown.

Separating the filters might be slower in routines where the cost of traversing the image is higher than the arithmetic involved. This typically happens with very large images, or images that don't fit into physical RAM.

Glossary

Accelerate framework A Mac OS X framework that serves as a container for several other frameworks related to optimization and high performance.

affine warp A transform that changes the distance between points by a linear transformation followed by a translation.An affine warp retains parallel lines, collinearity of points, and the ratio between collinear line segments, but does not preserve absolute length or angle measurements.Some common affine warps include scaling, shearing, rotation, and translation.

alpha channel A channel dedicated to representing how opaque a given pixel is. Unlike the red, green, and blue channels, which specify the intensity of their respective colors, the alpha channel specifies the opacity of the entire pixel.For example, if a pixel was defined using `float` values ranging from 0.0 to 1.0, an alpha channel intensity of 1.0 would indicate 100% opacity, while an intensity of 0.0 would indicate 0% opacity, or transparent.

convolution A common image processing technique that changes the intensities of a pixel to reflect the intensities of the surrounding pixels. Using convolution, you can get image effects like blur, emboss, and sharpen.

cache A special type of memory that is substantially faster than typical main memory (RAM). When a program asks the CPU to read or write data in memory, it first checks to see if this data is stored in cache memory (since it will be a lot faster to retrieve or write). Cache sizes are usually quite small though, (under 2 MB) so in order to make use of it, the data must be small enough to fit in the cache.

coefficient The number that is multiplied to each of the factors in a polynomial equation. For example, in the equation $x2 + 2x + 1$, the coefficients are *1*, *2*, and *1*.

dilation An effect that takes each bright pixel in the source image and expands 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).

erosion A morphological operation that is similar to dilation. It takes dark pixels in an image and spreads them around, causing them to "eat away" (or erode) objects in an image.

equalization A histogram operation that makes the resultant image conform to a uniform histogram, ensuring an equal frequency of pixel intensities.

filter An image process that when applied to an image causes its appearance to change. Many filters use kernel convolution to achieve their effect. Emboss, blur, smooth, and edge detect are all examples of common image filters that use convolution kernels.

gamma correction an operation that changes color intensity values to correct for the nonlinear response of the eye or of a display.

histogram A diagram that shows the frequency of occurrences within a data set. In the graphics domain, histograms can be used to plot the frequencies of certain pixel intensities.

horizontal reflect A type of geometric operation that reflects an image about its y-axis.

horizontal shear A filter that shifts pixels along the x-axis to create an effect that's similar to physical shearing.

image format An image encoding standard that specifies the number of color channels and number of bits per channel.

interleaved image format A format that encodes each color (and alpha) channel, one after the other, for every pixel. In contrast to planar image formats which encode an entire image using one color at a time, interleaved image formats alternate through each channel, encoding data for all channels simultaneously within each pixel. For example, an interleaved image would encode an image in a RGBRGBRGB fashion, where as a planar image would encode the same image as RRRGGGBBB.

kernel A grid of numbers used in both convolution and morphological operations (such as dilation and erosion). It is typically represented as a square grid (or matrix) whose height and width are both odd, such as a 3 x 3 grid. Each cell in the grid contains a number. An image process that uses a kernel typically takes these numbers within the kernel and applies them to the image by undergoing a series of arithmetic operations between the kernel values and the image pixel intensity values.

Lanczos resampling A commonly used math routine for resampling values in a data set. vImage uses this as a default technique for determining new pixels that did not previously exist in the input image.

lookup table A data structure used to quickly make computations or retrievals of certain values. In image processing it is used to store precalculated values of an equation instead of calculating the value each time it is requested. For example, if the equation is $y = 2x$, and you know that there are at most n distinct values, then you can create an array of size n (one for each input), that stores the precalculated value of the equation for the corresponding input (e.g. `[0] = 0, [1] = 2, [2] = 4, ... [n] = 2n`).

matrix A collection of numbers arranged in a grid. It can be thought of as the mathematical equivalent of a two-dimensional array. Much like two-dimensional arrays, matrices are composed of rows and columns with their elements referred to as *cells*.

non-premultiplied A technique for processing the alpha channel of a pixel. Instead of performing the alpha blend for each pixel, the alpha value is premultiplied to each of the other color channel values for that pixel. The pixel can then be interpreted as is from then on since all of the color channel values have been appropriately changed to reflect the alpha component.

object A group of bright, high-intensity pixels in an image, as opposed to the darker pixels, which are considered part of the background.

order The maximum number of factors in an equation. For example, the equation $x^2 + x + 1$ has an order of 3 because there are three distinct factors in the equation (x^2, x, and x^0).

planar image format A format that encodes a color channel. Planar images tend to be faster to operate on than nonplanar images because operations do not need to be repeated for each color channel. A grayscale image is an example of a planar image since it encodes only one (black and white) channel.

polynomial function An equation commonly used for transforming pixel intensities in an image that is a summation of n factors and coefficients in the form of $ax^n + bx^{n-1} + \ldots cx^0$.

premultiplied A pixel that already has its intensity levels appropriately multiplied by the alpha value.

Quartz The 2D graphics technology used throughout Mac OS X and in most Cocoa applications.

region of interest (ROI) The portion of an image data buffer that is being operated upon by a function. It is not uncommon to allocate a large pixel buffer to hold several images and then process only the smaller regions of interest when need be.

resampling An operation that changes the dimensions of an image.

resampling filter A function used to determine new pixel values for an image that has its dimensions changed somehow.

rotation An operation to rotate an image by a certain number of degrees.

scale To shrink or enlarge an image by a certain percent.

scalar programming A programming paradigm in which values are operated upon individually. Scalar programming is more common than vector code.

SIMD Single Instruction, Multiple Data. A computing technique used to achieve data level parallelism. Commonly employed in vector processors, this technique allows for multiple data elements to be processed in a single CPU instruction.

SSE Streaming SIMD Extensions. Intel's SIMD instruction set.

vector A grouped series of numbers. Commonly represented either as a row of numbers [1, 2, 3, 4] or a column of numbers. It is analogous to an array.

vector processor A processor that can perform arithmetic on several pairs of numbers simultaneously. Also called an *array processor*.

vector code Code makes use of available on-board vector processors. vImage uses vector code.

vertical reflect A type of geometric operation that reflects an image about its x-axis.

vertical shear A filter that shifts pixels along the y-axis to create an effect that's similar to physical shearing.

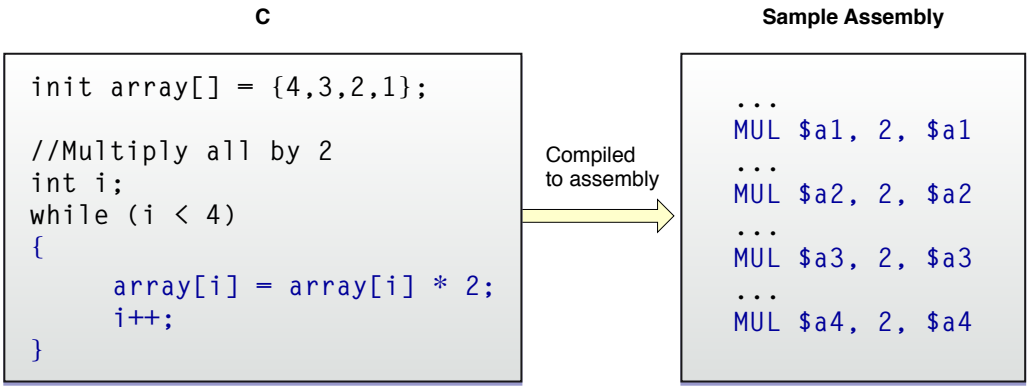
Vector Programming Primer

Vector programming is the programming paradigm that vImage uses to gain a performance benefit. This appendix explains how vector programming is able to achieve such benefits over traditional scalar programming.

With vector programming, multiple data elements (such as numbers) can be operated upon as one single unit – a vector. A vector can be thought of as an array that contains several elements, but unlike the array from traditional (or scalar) programming, all of the elements of the vector can be operated upon in a single instruction (as opposed to operating upon one array element at a time, as in scalar programming).

To better understand the significance of using vectors, you need to know how scalar code is processed by the CPU. Typically, when you program in a programming language (for example, C), the compiler converts the programming statements found in your source code files to a series of assembly instructions that the CPU can understand. These instructions usually consist of adding, subtracting, or multiplying two numbers together and storing the result somewhere in memory. Figure A-1 (page 63) shows how a compiler could handle scalar code.

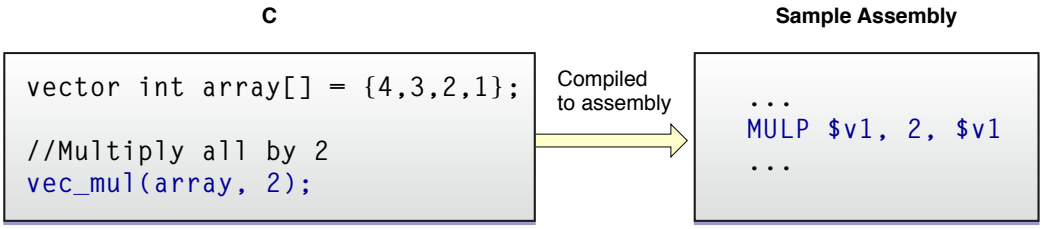
Figure A-1 Scalar code compilation example



As shown in Figure A-1 (page 63), to multiply each element of the array by 2, it is necessary to iterate through the array and multiply each individual element by 2. You can see the suggested assembly code produced in the figure to the right. (Note: This assembly code is intentionally inaccurate to more easily illustrate this concept).

Vector programming works similarly. The compiler converts programming statements into a series of assembly instructions for performing arithmetic. The main difference is that data can be stored in a new data type called a `vector`. Figure A-2 (page 64) shows the previous operation, but done using vectorized code.

Figure A-2 Vector code compilation example



When using vectorized code, to achieve the same goal of multiplying all of the numbers in the vector by 2, special operators exist that allow you to do this in one single instruction. This example uses `vec_mul` as our vector multiplication operator. Vector operator names change from processor to processor. This operator multiplies all elements in the vector, by a number (here, 2). This differs from the scalar example that iterates through the array and multiplies each number by 2.

When examining the assembly code produced by the vector version, you'll notice that the end result is the same, but with less assembly code than the scalar version.

This optimization may seem minor in this trivial example, but when considering the pixel-per-pixel operations frequently used in image processing routines (and the many layers of arithmetic involved), the performance gains can be significant.

Despite the efficiency of vector programming, in practice it can be difficult to implement. Aside from getting used to the new data types involved, you need to rewrite algorithms with data parallelism in mind. You also want to consider what types of vector processors you plan to support because the code and build process will most likely differ.

The vImage framework is built on top of vector programming routines for several different architectures, enabling it to abstract the nuances of hardware-specific vector programming. This not only saves you programming time but also allows you to deploy the software on various system architectures (including those without a vector processor, such as the G3 architecture) with the same code.

Document Revision History

This document describes the vImage framework.

This table describes the changes to *vImage Programming Guide*.

Date	Notes
2010-06-14	Corrected a glossary definition.
2008-10-15	Completely re-written, updated and restructured since previous versions. Included several new figures and provided more in-depth descriptions of the operations vImage offers.

Document Revision History