Article

Converting bitmap data between Core Graphics images and vImage buffers

Pass image data between Core Graphics and vImage to create and manipulate images.

Overview

When you work with bitmap images, you typically work with Core Graphics `CGImage` instances. The vImage library provides functionality that allows you to work with Core Graphics bitmap images. Typically, you convert a `CGImage` instance to a vImage buffer, apply operations to the vImage buffer, and convert the transformed data to a new `CGImage` instance.

Conversions between Core Graphics-backed images and vImage buffers require a `vImage_CGImageFormat` format structure. The format describes properties such as the color space, the number of channels and how they're ordered, and the size, in bits, of the color channels.

Initialize an 8-bit Core Graphics image format from hard-coded values

Create a `vImage_CGImageFormat` structure with hard-coded values when you need to perform operations on image data with properties that your app defines at compile time. For example, the code below initializes an 8-bit-per-channel ARGB format that's suitable for working with `vImage.PixelBuffer` structures with a `vImage.Interleaved8x4` format.

```
var format = vImage_CGImageFormat(
    bitsPerComponent: 8,
    bitsPerPixel: 8 * 4,
    colorSpace: CGColorSpace(name: CGColorSpace.displayP3)!,
    bitmapInfo: .init(rawValue: CGImageAlphaInfo.noneSkipFirst.rawValue))!
```

The `init(cgImage:cgImageFormat:pixelFormat:)` initializer creates a pixel buffer from a `CGImage` instance and converts the source image data to the format that the `vImage_CGImageFormat` describes. If you're working with `vImage_Buffer` structures, the `vImageBuffer_InitWithCGImage(_:_:_:_:)` function performs the same conversion and creates a buffer that's suitable for working with ARGB8888 operations, such as `vImageConvolve_ARGB8888(_:_:_:_:_:_:_:_:_:)`.

The code below creates a buffer from a `CGImage` instance named `cgImage` and prints the values of the first two pixels. The opaque alpha values (255) are the first value in each pixel as `CGImageAlphaInfo.noneSkipFirst` defines.

```
let buf = try vImage.PixelBuffer(
    cgImage: cgImage,
    cgImageFormat: &format,
    pixelFormat: vImage.Interleaved8x4.self)

// Prints:
// "[255, 115, 136, 165,  255, 115, 136, 165]"
//    A    R    G    B  |  A    R    G    B
print(buf.array[0 ..< 2 * format.componentCount])
```

For all the examples in this article, pass the `vImage_CGImageFormat` structure to the `makeCGImage(cgImageFormat:)` method to generate an output image.

```
let result = buf.makeCGImage(cgImageFormat: format)
```

On return, `result` is a four-channel 8-bit-per-channel image with `CGImageAlphaInfo.noneSkipLast` alpha ordering.

Initialize a 32-bit Core Graphics image format from hard-coded values

You can use the `init(cgImage:cgImageFormat:pixelFormat:)` initializer to convert an image's color model and bit depth. For example, you can specify a grayscale 32-bit format such as the example below:

```
let bitmapInfo = CGBitmapInfo(
    rawValue: kCGBitmapByteOrder32Host.rawValue |
    CGBitmapInfo.floatComponents.rawValue |
```

```
    CGImageAlphaInfo.none.rawValue)

var format = vImage_CGImageFormat(bitsPerComponent: 32,
                                  bitsPerPixel: 32,
                                  colorSpace: CGColorSpaceCreateDeviceGray(),
                                  bitmapInfo: bitmapInfo)!
```

In this case, the initializer uses the Rec. 601 luma coefficients to convert the RGB pixel values (`115, 136, 165`) to a single grayscale pixel.

```
let buf = try vImage.PixelBuffer(cgImage: cgImage,
                                 cgImageFormat: &format,
                                 pixelFormat: vImage.PlanarF.self)

// Prints:
// "[133, 133]"  // (0.299 * 115 + 0.587 * 136 + 0.114 * 165) = 133
//    Y    Y
print(buf.array[0 ..< 2 * format.componentCount].map {
    Pixel_8($0 * 255 )
})
```

As above, pass the 32-bit format to `makeCGImage(cgImageFormat:)` to create a single-channel 32-bit-per-channel image with no alpha information.

```
let result = buf.makeCGImage(cgImageFormat: format)
```

On return, `result` contains a grayscale version of the original image.

Initialize a Core Graphics image format from a Core Graphics image

The `init(cgImage:)` initializer creates a new `vImage_CGImageFormat` structure that describes a Core Graphics image's properties.

```
guard var format = vImage_CGImageFormat(cgImage: cgImage) else {
    NSLog("Unable to derive format from image.")
    return
}
```

```
print(format.bitsPerComponent)                      // 8
print(format.componentCount)                        // 4
print(format.colorSpace.takeRetainedValue().name!)  // kCGColorSpaceDisplayP3
print (format.bitmapInfo)                           // noneSkipLast
```

In this example, the image's inherent channel ordering is `CGImageAlphaInfo.noneSkipLast`. The code below prints the first two pixels and shows that the opaque alpha values (255) are the last two values in each pixel:

```
let buf = try vImage.PixelBuffer(cgImage: cgImage,
                                 cgImageFormat: &format,
                                 pixelFormat: vImage.Interleaved8x4.self)

// Prints:
// "[115, 136, 165, 255,  115, 136, 165, 255]"
//    R    G    B    A  |  R    G    B    A
print(buf.array[0 ..< 2 * format.componentCount])
```

Initialize a Core Graphics image format from an image during pixel buffer initialization

You can pass an empty `vImage_CGImageFormat` structure to the `init(cgImage:cgImageFormat:pixelFormat:)` initializer to instruct the initializer to populate the format with an image's properties. In this case, the initializer returns `nil` if the image's and the pixel buffer's bit depths aren't equal.

```
guard cgImage.bitsPerComponent == 8,
      cgImage.bitsPerPixel == 8 * 4 else {
    fatalError("Unsupported `bitsPerComponent` and `bitsPerPixel`.")
}

var format = vImage_CGImageFormat()

let buf = try vImage.PixelBuffer(cgImage: cgImage,
                                 cgImageFormat: &format,
                                 pixelFormat: vImage.Interleaved8x4.self)
```

On return, the mutable format variable contains the image properties.

```
print(format.colorSpace.takeRetainedValue().name!)  // kCGColorSpaceDisplayP3
print(format.bitmapInfo)                            // noneSkipLast

// Prints:
// "[115, 136, 165, 255,  115, 136, 165, 255]"
//    R    G    B    A  |  R    G    B    A
print(buf.array[0 ..< 2 * format.componentCount])
```

See Also

Image Processing Essentials

- Creating and Populating Buffers from Core Graphics Images
  Initialize vImage buffers from Core Graphics images.
- Creating a Core Graphics Image from a vImage Buffer
  Create displayable representations of vImage buffers.
- Building a Basic Image-Processing Workflow
  Resize an image with vImage.
- Applying geometric transforms to images
  Reflect, shear, rotate, and scale image buffers using vImage.
- Compositing images with alpha blending
  Combine two images by using alpha blending to create a single output.
- Compositing images with vImage blend modes
  Combine two images by using blend modes to create a single output.
- Applying vImage operations to regions of interest
  Limit the effect of vImage operations to rectangular regions of interest.
- Optimizing image-processing performance
  Improve your app's performance by converting image buffer formats from interleaved to planar.

- vImage
  Manipulate large images using the CPU's vector processor.