Sample Code

Converting color images to grayscale

Convert an RGB image to grayscale using matrix multiplication.

Download

macOS 13.3+ | Xcode 14.3+

Overview

The vImage `multiply(by:divisor:preBias:postBias:destination:)` and `multiply(by:preBias:postBias:destination:)` functions multiply each channel of an interleaved image with the corresponding value in a matrix and return the sum of the multiplications to generate a planar image. These functions wrap `vImageMatrixMultiply_ARGB8888ToPlanar8(_:_:_:_:_:_:_:)` and `vImageMatrixMultiply_ARGBFFFFToPlanarF(_:_:_:_:_:)`, respectively.

The following shows how the 8-bit matrix multiply operation calculates the result for each pixel:

```
let p = (sourcePixels[index].a + preBias.a) * matrix.a +
        (sourcePixels[index].r + preBias.r) * matrix.r +
        (sourcePixels[index].g + preBias.g) * matrix.g +
        (sourcePixels[index].b + preBias.b) * matrix.b

let destinationPixels[index] = (p + postBias) / divisor
```

The 8-bit matrix multiply operation creates a maximum of 255 gray tones. However, even with an 8-bit-per-channel source image, the 32-bit matrix multiply operation can create up to 255 x 255 x 255 (16,581,375) gray tones. This sample code project includes 8- and 32-bit color-to-grayscale conversion and provides a function to count the distinct tones in the output image.

Before exploring the code, build and run the app to familiarize yourself with the different visual results it generates from setting different coefficients for the red, green, and blue channels.

Define the source and destination pixel buffers

To support 8- and 32-bit matrix multiply, the sample code defines four pixel buffers: two ARGB source buffers and two grayscale destination buffers.

```
/// The 8-bit-per-channel, 4-channel source pixel buffer.
let sourceBuffer8 = try! vImage.PixelBuffer<vImage.Interleaved8x4>(
    cgImage: sourceImage,
    cgImageFormat: &GrayscaleConverter.sourceFormat8)

/// The 32-bit-per-channel, 4-channel source pixel buffer.
let sourceBufferF = try! vImage.PixelBuffer<vImage.InterleavedFx4>(
    cgImage: sourceImage,
    cgImageFormat: &GrayscaleConverter.sourceFormatF)

/// The 8-bit planar destination pixel buffer.
let destinationBuffer8 = vImage.PixelBuffer<vImage.Planar8>(width: sourceImage.width
                                                            height: sourceImage.heig

/// The 32-bit planar destination pixel buffer.
let destinationBufferF = vImage.PixelBuffer<vImage.PlanarF>(width: sourceImage.width
                                                            height: sourceImage.heig
```

Define the coefficient values

Luma coefficients model an eye's response to red, green, and blue light. The following formula shows the Rec. 709 luma coefficients for the sample app's default color-to-grayscale conversion.

```
luminance = (red × 0.2126) + (green x 0.7152) + (blue × 0.0722)
```

The sample code app provides a user interface that allows a user to change the red, green, and blue coefficients. To ensure the grayscale image isn't darker or brighter than the original image, the following code normalizes the coefficient values so that their sum equals `1.0`:

```
let scale = 1.0 / (redCoefficient + greenCoefficient + blueCoefficient)

DispatchQueue.main.async { [self] in
```

```
    normalizedRedCoefficient = redCoefficient * scale
    normalizedGreenCoefficient = greenCoefficient * scale
    normalizedBlueCoefficient = blueCoefficient * scale
}
```

Perform the 8-bit matrix multiply operation

The 8-bit matrix multiply operation accepts integer matrix values. The following code defines a divisor that it uses to multiply the floating-point coefficient values and that the function uses to renormalize the image after scaling by the matrix:

```
let divisor: Int = 0x1000
let fDivisor = Float(divisor)

sourceBuffer8.multiply(by: (0,
                            Int(normalizedRedCoefficient * fDivisor),
                            Int(normalizedGreenCoefficient * fDivisor),
                            Int(normalizedBlueCoefficient * fDivisor)),
                       divisor: divisor,
                       preBias: (0, 0, 0, 0),
                       postBias: 0,
                       destination: destinationBuffer8)
```

On return, the `destinationBuffer8` pixel buffer contains a grayscale representation of the original image.

Perform the 32-bit matrix multiply operation

The 32-bit matrix multiply operation accepts floating-point matrix values and, therefore, doesn't require a divisor. The following code performs the 32-bit color-to-grayscale conversion.

```
sourceBufferF.multiply(by: (0,
                            normalizedRedCoefficient,
                            normalizedGreenCoefficient,
                            normalizedBlueCoefficient),
                       preBias: (0, 0, 0, 0),
                       postBias: 0,
                       destination: destinationBufferF)
```

On return, the `destinationBufferF` pixel buffer contains a grayscale representation of the original image.

Create a grayscale Core Graphics image

The sample app displays the 8- and 32-bit grayscale images in the user interface. To support this, the following code defines two single-channel `vImage_CGImageFormat` structures:

```
static var destinationFormat8 = vImage_CGImageFormat(
    bitsPerComponent: 8,
    bitsPerPixel: 8,
    colorSpace: CGColorSpaceCreateDeviceGray(),
    bitmapInfo: CGBitmapInfo(rawValue: CGImageAlphaInfo.none.rawValue))!

static var destinationFormatF = vImage_CGImageFormat(
    bitsPerComponent: 32,
    bitsPerPixel: 32,
    colorSpace: CGColorSpaceCreateDeviceGray(),
    bitmapInfo: CGBitmapInfo(
        rawValue: kCGBitmapByteOrder32Host.rawValue |
        CGBitmapInfo.floatComponents.rawValue |
        CGImageAlphaInfo.none.rawValue))!
```

The `makeCGImage(cgImageFormat:)` function is available for both the 8- and 32-bit pixel buffers. The following code creates a `CGImage` instance from the 32-bit grayscale pixel buffer:

```
let result = destinationBufferF.makeCGImage(
    cgImageFormat: GrayscaleConverter.destinationFormatF)!
```

On return, `result` contains the grayscale representation of the original image:

See Also

Conversion Between Image Formats

- Building a basic image conversion workflow

  Learn the fundamentals of the convert-any-to-any function by converting a CMYK image to an RGB image.

- Applying color transforms to images with a multidimensional lookup table

  Precompute translation values to optimize color space conversion and other pointwise operations.

- Building a basic image conversion workflow

  Learn the fundamentals of the convert-any-to-any function by converting a CMYK image to an RGB image.

- Converting luminance and chrominance planes to an ARGB image

  Create a displayable ARGB image using the luminance and chrominance information from your device's camera.

- Conversion

  Convert an image to a different format.