Sample Code

Adjusting saturation and applying tone mapping

Convert an RGB image to discrete luminance and chrominance channels, and apply color and contrast treatments.

Download

macOS 13.3+ | Xcode 14.3+

Overview

This sample code project allows you to apply saturation adjustments to an image without affecting luminosity, and change the luminance response curve without affecting color.

Many image-processing techniques, such as saturation adjustment and tone mapping, are simpler to implement when you can work on an image's luminance data separately from its color data. This article explains how you can convert an RGB image — with its pixels represented as red, green, and blue values — to YpCbCr, which stores luminance and chrominance discretely. The *Yp* in YpCbCr refers to the luminance, and the *Cb* and *Cr* refer to the blue-luminance difference, and red-luminance difference, respectively.

This sample app converts an ARGB image to YpCbCr and applies adjustments based on user-interface controls. When you decrease the saturation, the sample app applies gamma to the CbCr buffers. When you increase the saturation, the sample app scales the CbCr buffers, and when you change contrast, the sample app applies gamma to the Yp buffer.

The following images show two photographs with a range of saturation adjustments that illustrate the variety of color changes you can make using the sample code app:

Before exploring the code, try building and running the app to familiarize yourself with the effect of the different transformations on the image.

Create source and destination ARGB pixel buffers

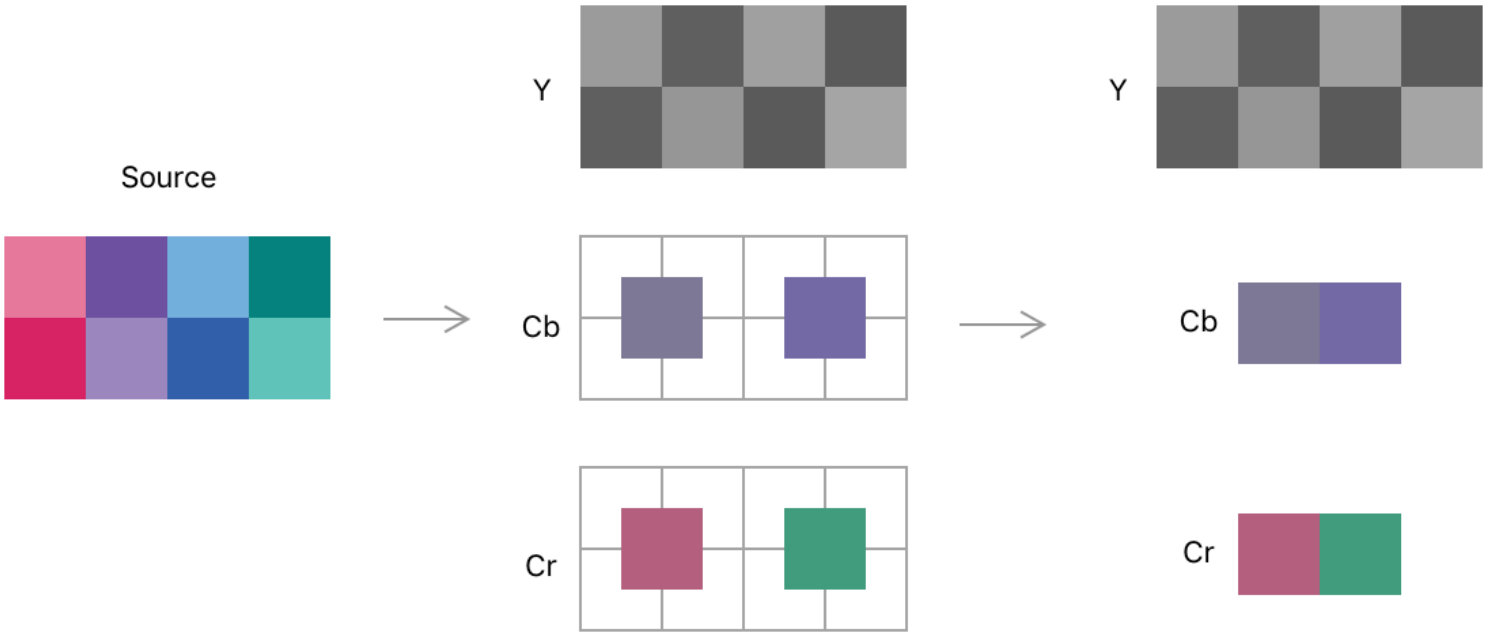
The sample declares two 8-bit, four-channel pixel buffers. The `argbSource` pixel buffer stores the source image, and the `argbDestination` stores the transformed image.

```
private lazy var argbSource: vImage.PixelBuffer<vImage.Interleaved8x4> = {
    return try! vImage.PixelBuffer<vImage.Interleaved8x4>(cgImage: sourceCGImage,
                                                          cgImageFormat: &format)
}()

private lazy var argbDestination: vImage.PixelBuffer<vImage.Interleaved8x4> = {
    return vImage.PixelBuffer<vImage.Interleaved8x4>(width: self.width,
                                                     height: self.height)
}()
```

Create the YpCbCr buffers

The conversion routine that this sample uses creates a YpCbCr result with a chroma of 4:2:0, which means there is one Cb and one Cr pixel for every four luminance pixels. That is, each chrominance buffer is half of the width, and half of the height of the luminance channel. Reducing the resolution for the chrominance channels is known as *chroma subsampling*, and it relies on the fact that human vision is less sensitive to color than luminance.

The image below shows that a 4 x 2 image is represented by a 4 x 2 luminance channel, but each chrominance channel is 2 x 1 pixels:



To support the 4:2:0 YpCbCr representation of the source image, the sample project defines a `Yp8CbCr8PixelBuffers` structure that contains two pixel buffers. The luminance buffer is the same size as the source buffer. The chrominance buffer's height is half the source height, and its width is the same as the source width. This size enables the chrominance buffer to store both the Cb and Cr data as interleaved pixels.

```
struct Yp8CbCr8PixelBuffers {
    /// The luminance pixel buffer.
    let yp: vImage.PixelBuffer<vImage.Planar8>

    /// The chrominance pixel buffer.
    let cbcr: vImage.PixelBuffer<vImage.Planar8>

    init(width: Int, height: Int) {
        yp = vImage.PixelBuffer<vImage.Planar8>(width: width,
                                                height: height)

        cbcr = vImage.PixelBuffer<vImage.Planar8>(width: width,
                                                  height: height / 2)
    }
```

The following code creates two `Yp8CbCr8PixelBuffers` structures that contain a representation of the source image before and after saturation adjustment and tone mapping:

```
lazy private var ypCbCrPreTransformBuffers: Yp8CbCr8PixelBuffers = {
    return Yp8CbCr8PixelBuffers(width: width, height: height)
}()
```

```
lazy private var ypCbCrPostTransformBuffers: Yp8CbCr8PixelBuffers = {
    return Yp8CbCr8PixelBuffers(width: width, height: height)
}()
```

Define the RGB-to-YpCbCr conversion

The `vImage_YpCbCrPixelRange` structure defines the range and clamping information for the destination YpCbCr format. The destination buffer is 8-bit, therefore, the minimum and maximum values for luminance and chrominance are `0` and `255`, respectively. `CbCr_bias` specifies the middle of the CbCr range (that is, where the blue-luminance difference or red-luminance difference is `0`), and the sample sets that to `128`.

```
private let pixelRange = vImage_YpCbCrPixelRange(Yp_bias: 0,
                                                 CbCr_bias: 128,
                                                 YpRangeMax: 255,
                                                 CbCrRangeMax: 255,
                                                 YpMax: 255,
                                                 YpMin: 0,
                                                 CbCrMax: 255,
                                                 CbCrMin: 0)
```

The `Yp8CbCr8PixelBuffers` structure uses `vImageConvert_ARGBToYpCbCr_GenerateConversion(_:_:_:_:_:)` to generate the conversion from ARGB to YpCbCr. The sample calculates the conversion of RGB values using the conversion matrix for ITU Recommendation BT.709-2.

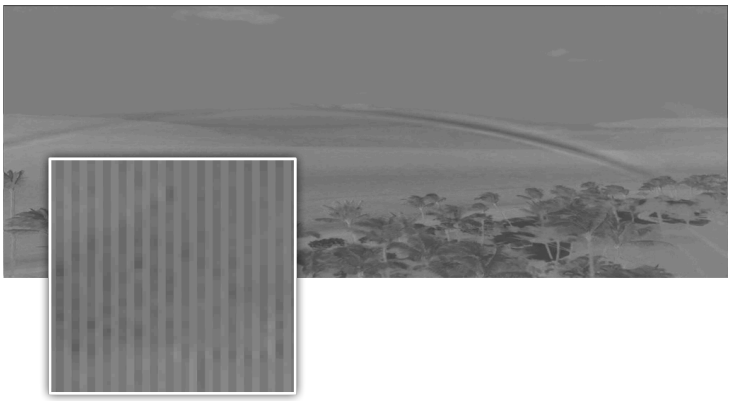
```
private var argbToYpCbCr: vImage_ARGBToYpCbCr {
    var outInfo = vImage_ARGBToYpCbCr()

    withUnsafePointer(to: pixelRange) { ptr in
        _ = vImageConvert_ARGBToYpCbCr_GenerateConversion(
            kvImage_ARGBToYpCbCrMatrix_ITU_R_709_2,
            ptr,
            &outInfo,
            kvImageARGB8888,
            kvImage420Yp8_CbCr8,
            vImage_Flags(kvImageNoFlags))
    }
    return outInfo
}
```

Perform the RGB-to-YpCbCr conversion

The vImageConvert_ARGB8888To420Yp8_CbCr8(_:_:_:_:_:_:) function populates two vImage buffers — one that contains luminance data and one that contains chrominance data — from the contents of a single ARGB buffer.

```
func convert(from source: vImage.PixelBuffer<vImage.Interleaved8x4>) {
    source.withUnsafePointerToVImageBuffer { src in
        withUnsafePointer(to: argbToYpCbCr) { info in
            self.yp.withUnsafePointerToVImageBuffer { yp in
                self.cbcr.withUnsafePointerToVImageBuffer { cbcr in
                    _ = vImageConvert_ARGB8888To420Yp8_CbCr8(
                        src,
                        yp,
                        cbcr,
                        info,
                        [3, 2, 1, 0],
                        vImage_Flags(kvImagePrintDiagnosticsToConsole))
                }
            }
        }
    }
}
```

The following image shows the luminance result on the left and the interleaved chrominance result on the right. Because the interleaved chrominance result contains both the Cb and Cr information, it's half the height of the luminance channel, but has the same width.



Apply saturation adjustment to the image

This sample uses two techniques to adjust saturation:

- Multiply CbCr values to decrease saturation.
- Apply gamma to CbCr to increase saturation.

It performs the tone mapping by applying gamma to the luminance channel.

```
if saturation > 1 {
    applyGammaToCbCr(gamma: 1 / saturation)
} else {
    applyLinearToCbCr(saturation: saturation)
}

applyGammaToLuma(lumaGamma: lumaGamma)
```

Multiply CbCr values to decrease saturation

The following formula describes how to adjust the color saturation of a YpCbCr image, without affecting its luminance:

$$cb' = ((cb - 128) \times saturation) + 128$$
$$cr' = ((cr - 128) \times saturation) + 128$$

The `multiply(by:divisor:preBias:postBias:destination:)` function performs this math on the source chrominance buffer. The function passes the saturation to the matrix multiply function as a single-element matrix, and passes the chrominance buffer as the source and destination.

```
/// Reduces saturation.
private func applyLinearToCbCr(saturation: Float) {
    let preBias = -128
    let divisor = 0x1000
    let postBias = 128 * divisor

    let factor = Int(saturation * Float(divisor))

    ypCbCrPreTransformBuffers.cbcr.multiply(by: factor,
                                            divisor: divisor,
                                            preBias: preBias,
```

```
                                                postBias: postBias,
                                                destination: ypCbCrPostTransformBuffers.
}
```

The following image shows two photographs, from left to right, with saturations of `0.25`, `0.75`, and `1.0` (that is, the rightmost image has an unchanged saturation).

Apply gamma to CbCr to increase saturation

The simple linear adjustment that `multiply(by:divisor:preBias:postBias:destination:)` provides is fine for desaturating an image, however, when increasing saturation, multiplication can clip the CbCr values, leading to areas of solid color. An alternative technique to increase saturation is to apply an exponential adjustment. The `applyGamma(_:destination:)` function applies a gamma value to the CbCr values to increase saturation.

```
/// Increases saturation.
private func applyGammaToCbCr(gamma: Float) {

    // Convert 8-bit CbCr values to 32-bit.
    ypCbCrPreTransformBuffers.cbcr.convert(to: gammaDestination)

    // Scale 32-bit values from `0.0 ... 1.0` to `-1.0 ... 1.0`.
    gammaDestination.multiply(by: 2,
                              preBias: 0, postBias: -1,
                              destination: gammaDestination)
```

```
    // Apply gamma to 32-bit values.
    gammaDestination.applyGamma(.fullPrecision(gamma),
                                destination: gammaDestination)

    // Scale 32-bit transformed values from `-1.0 ... 1.0` to `0 ... 1.0`.
    gammaDestination.multiply(by: 0.5,
                              preBias: 1, postBias: 0,
                              destination: gammaDestination)

    // Convert 32-bit transformed CbCr values to 8-bit.
    gammaDestination.convert(to: ypCbCrPostTransformBuffers.cbcr)
}
```

The following image shows two photographs, from left to right, with a saturation of 1.0 (that is, the leftmost image has an unchanged saturation), 1.5, and 2.0:

When decreasing the saturation, the gamma function is not appropriate because pixels with very saturated color will desaturate very little, or not at all.

Apply gamma to luminance to perform tone mapping

The sample app adjusts the contrast of an image, with a technique known as *tone mapping*, by applying a gamma adjustment to the luminance channel.

Adjusting contrast is discussed in Adjusting the brightness and contrast of an image, however, applying a gamma adjustment to red, green, and blue channels changes both the color and tonal values.

The `applyGamma(linearParameters:exponentialParameters:boundary:destination:)` function applies a piecewise gamma tranformation on the planar `yp` `Destination` buffer, which contains the luminance data.

```
private func applyGammaToLuma(lumaGamma: Float) {

    ypCbCrPreTransformBuffers.yp.applyGamma(
        linearParameters: (scale: 1, bias: 0),
        exponentialParameters: (scale: 1, preBias: 0, gamma: lumaGamma, postBias: 0)
        boundary: 0,
        destination: ypCbCrPostTransformBuffers.yp)

}
```

The following image shows two photographs, from left to right, with a gamma applied to the luminance channel of 2.5, 0.0 (that is, the center image is unchanged), and 0.5:

Define the YpCbCr-to-RGB conversion

After the sample app completes the YpCbCr representation, it converts the YpCbCr data to RGB. The process is very similar to the RGB to YpCbCr conversion and uses the same pixel range, but

the `vImageConvert_YpCbCrToARGB_GenerateConversion(_:_:_:_:_:)` function generates the conversion.

```
private var ypCbCrToARGB: vImage_YpCbCrToARGB {
    var outInfo = vImage_YpCbCrToARGB()

    withUnsafePointer(to: pixelRange) { ptr in
        _ = vImageConvert_YpCbCrToARGB_GenerateConversion(
            kvImage_YpCbCrToARGBMatrix_ITU_R_709_2,
            ptr,
            &outInfo,
            kvImage420Yp8_CbCr8,
            kvImageARGB8888,
            vImage_Flags(kvImageNoFlags))
    }

    return outInfo
}
```

The `Yp8CbCr8PixelBuffers` structure exposes a method for converting to ARGB.

```
func convert(to destination: vImage.PixelBuffer<vImage.Interleaved8x4>) {
    _ = withUnsafePointer(to: ypCbCrToARGB) { info in
        self.cbcr.withUnsafePointerToVImageBuffer { cbcrDest in
            self.yp.withUnsafePointerToVImageBuffer { ypDest in
                destination.withUnsafePointerToVImageBuffer { argbDest in
                    vImageConvert_420Yp8_CbCr8ToARGB8888(
                        ypDest,
                        cbcrDest,
                        argbDest,
                        info,
                        [3, 2, 1, 0],
                        255,
                        vImage_Flags(kvImagePrintDiagnosticsToConsole))
                }
            }
        }
    }
}
```

Correct gamma before applying operations

Many vImage operations provide optimal results when working on images with a linear response curve. The sample app includes a `Toggle` control that applies a reciprocal gamma to the sRGB image, performs the saturation adjustments and tone mapping, and applies the original gamma.

vImage provides predefined gamma functions for converting from linear to sRGB, and from sRGB to linear. The sample implements the following function as an extension to `vImage.PixelBuffer` and remaps the buffer's contents in-place in the specified direction:

```
extension vImage.PixelBuffer where Format == vImage.Interleaved8x4 {

    enum Remap {
        case linearToSRGB
        case sRGBToLinear

        var gammaType: vImage.Gamma {
            switch self {
                case .linearToSRGB:
                    return .sRGBForwardHalfPrecision
                case .sRGBToLinear:
                    return .sRGBReverseHalfPrecision
            }
        }
    }

    func remap(_ remap: Remap) {
        self.applyGamma(remap.gammaType,
                        intermediateBuffer: nil,
                        destination: self)
    }
}
```

See Also

Color and Tone Adjustment

{} Adjusting the brightness and contrast of an image

Use a gamma function to apply a linear or exponential curve.

{} Applying tone curve adjustments to images

Use the vImage library's polynomial transform to apply tone curve adjustments to images.

{} Adjusting the hue of an image

Convert an image to L*a*b* color space and apply hue adjustment.

{} Specifying histograms with vImage

Calculate the histogram of one image, and apply it to a second image.

Enhancing image contrast with histogram manipulation

Enhance and adjust the contrast of an image with histogram equalization and contrast stretching.

Histogram

Calculate or manipulate an image's histogram.