Article

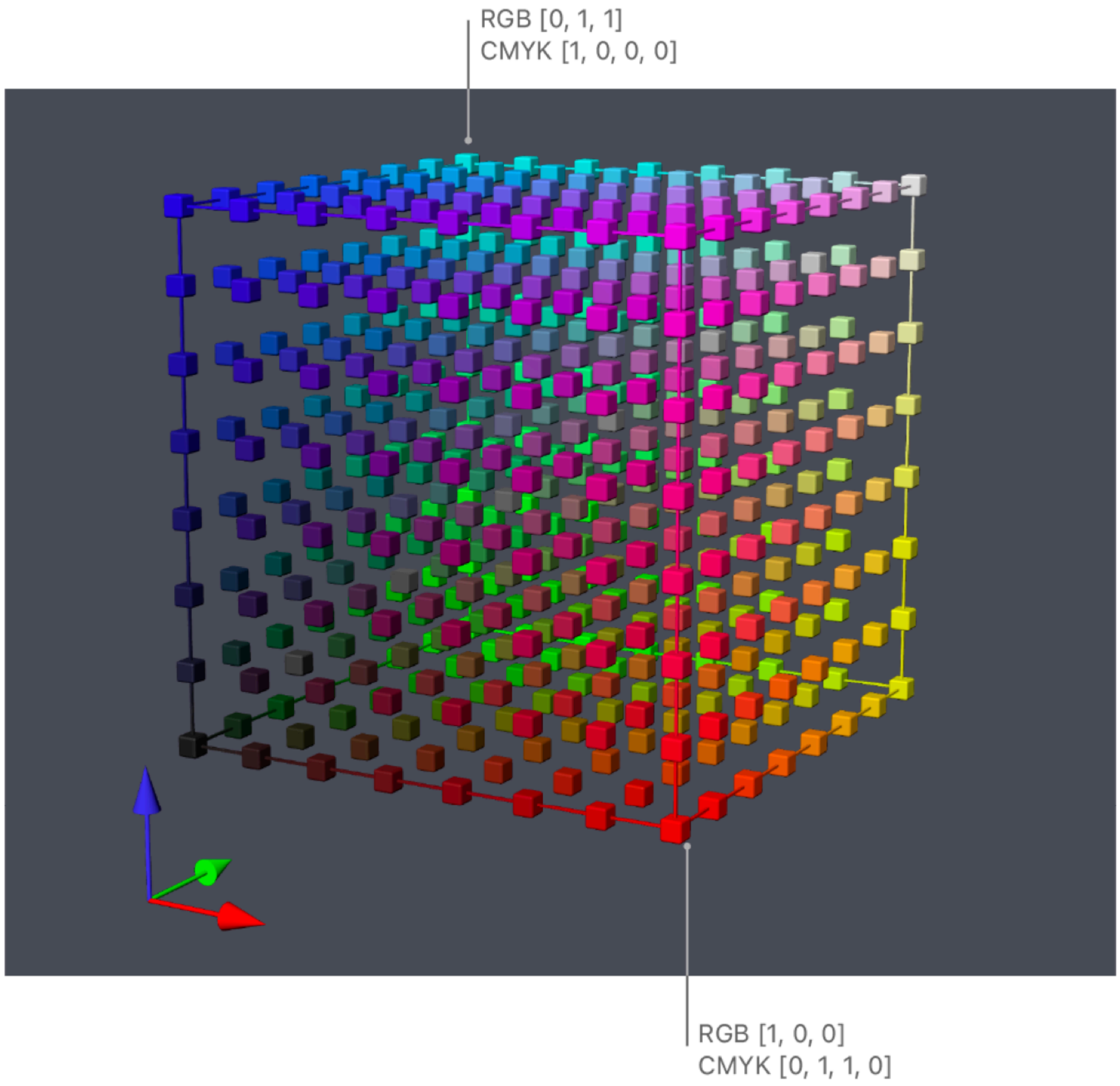
Applying color transforms to images with a multidimensional lookup table

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

Overview

When you perform color space conversions or other color transformations, it can be quicker to precompute the conversion values rather than performing the calculation for each individual pixel. The vImage library provides functionality for transforming images using multidimensional lookup tables. A multidimensional lookup table allows you to define such complex color transformations between image formats with the same or different numbers of channels.

An example of a multidimensional lookup table is a 3D lookup table that converts RGB colors to the CMYK color model that’s commonly used in printing. As the following image illustrates, an RGB to CMYK multidimensional lookup table is a cube that represents the RGB color space, with axes for red, green, and blue. Each point in that cube corresponds to a unique RGB color that contains the corresponding CMYK value.



Each dimension of the RGB cube in the image above contains eight entries — that is, there are 8^3 or 512 source samples, which appear as small cubes. Each sample contains four values that represent the CMYK values for the corresponding RGB color. For example, the sample for pure red (RGB `[1, 0, 0]`) contains the CMYK values `[0, 1, 1, 0]`, and the sample for cyan (RGB `[0, 1, 1]`) contains the CMYK values `[1, 0, 0, 0]`.

Define the lookup table size

Define the number of elements in the lookup table from the number of table entries per channel, the number of source channels, and the number of destination channels. For the RGB to CMYK conversion, in the following example, the lookup table contains four values (for cyan, magenta, yellow, and black) at each point in the RGB cube. The RGB cube contains the number of table entries per channels cubed.

```
let entriesPerChannel = UInt8(16)
let srcChannelCount = UInt32(3)
let destChannelCount = UInt32(4)

let lookupTableElementCount = Int(pow(Float(entriesPerChannel),
                                      Float(srcChannelCount))) *
                                  Int(destChannelCount)
```

You can set the vImage multidimensional lookup table functions to linearly interpolate between values. A high `entriesPerChannel` value provides greater color fidelity than a low value, but with a corresponding performance and memory overhead.

Create the table data

Supply the lookup table as a contiguous array of samples that define the lookup table values. The following code iterates over red, green, and blue values and creates an RGB CGColor instance for each permutation. The code converts the RGB color instance to the CMYK color space and populates the lookup table with the cyan, magenta, yellow, and black components.

```
let tableData = [UInt16](unsafeUninitializedCapacity: lookupTableElementCount) {
    buffer, count in

    /// Supply the samples in the range `0...65535`. The transform function
    /// interpolates these to the range `0...1`.
    let multiplier = CGFloat(UInt16.max)
    var bufferIndex = 0

    for red in ( 0 ..< entriesPerChannel) {
        for green in ( 0 ..< entriesPerChannel) {
            for blue in ( 0 ..< entriesPerChannel) {

                /// Create normalized red, green, and blue values in the range `0..
                let normalizedColor = simd_double3(
                    x: CGFloat(red),
                    y: CGFloat(green),
                    z: CGFloat(blue)) / CGFloat(entriesPerChannel - 1)

                /// Create a CMYK representsation of the RGB color.
                let cmyk = ColorConverter.rgbToCMYK(normalizedColor) * multiplier

                /// Append the cyan, magenta, yellow, and black components to the bu
                buffer[ bufferIndex ] = UInt16(cmyk.x)
                bufferIndex += 1
                buffer[ bufferIndex ] = UInt16(cmyk.y)
                bufferIndex += 1
                buffer[ bufferIndex ] = UInt16(cmyk.z)
                bufferIndex += 1
                buffer[ bufferIndex ] = UInt16(cmyk.w)
                bufferIndex += 1
            }
        }
    }

    count = lookupTableElementCount
}
```

In the following example, the RGB-to-CMYK value conversion code uses a `vImageConverter` instance to convert a single RGB pixel to a single CMYK pixel:

```
struct ColorConverter {
    static let rgbColorSpace = CGColorSpace(name: CGColorSpace.displayP3)!
    static let cmykColorSpace = CGColorSpace(name: CGColorSpace.genericCMYK)!

    static let bitmapInfo = CGBitmapInfo(
        rawValue: kCGBitmapByteOrder32Host.rawValue |
        CGBitmapInfo.floatComponents.rawValue |
        CGImageAlphaInfo.none.rawValue)

    static let cmykToLabConverter = try! vImageConverter.make(
        sourceFormat: .init(bitsPerComponent: 32,
                            bitsPerPixel: 32 * 3,
                            colorSpace: rgbColorSpace,
                            bitmapInfo: bitmapInfo)!,
        destinationFormat: .init(bitsPerComponent: 32,
                                 bitsPerPixel: 32 * 4,
                                 colorSpace: cmykColorSpace,
                                 bitmapInfo: bitmapInfo)!)

    @inlinable
    static func rgbToCMYK(_ src: simd_double3) -> simd_double4 {

        let srcPixelBuffer = vImage.PixelBuffer<vImage.InterleavedFx3>(
            pixelValues: [src.x, src.y, src.z].map { Float($0) },
            size: .init(width: 1, height: 1))
        let dstPixelBuffer = vImage.PixelBuffer<vImage.InterleavedFx4>(
            size: .init(width: 1, height: 1))

        try! cmykToLabConverter.convert(from: srcPixelBuffer, to: dstPixelBuffer)

        let dstColor = dstPixelBuffer.array.map { CGFloat($0 )}

        return .init(x: dstColor[0], y: dstColor[1], z: dstColor[2], w: dstColor[3]
    }
}
```

Create the lookup table

Call `vImageMultidimensionalTable_Create(_:_:_:_:_:_:_:)` to create a multidimensional lookup table from the table data array. Because the code in this example only uses the 32-bit transform function, pass the `kvImageMDTableHint_Float` hint to reduce memory overhead:

```
var error = kvImageNoError

let tableEntriesPerDimension = [UInt8](repeating: entriesPerChannel,
                                       count: Int(srcChannelCount))
guard let lookupTable = vImageMultidimensionalTable_Create(
        tableData,
        srcChannelCount,
        destChannelCount,
        tableEntriesPerDimension,
        kvImageMDTableHint_Float,
        vImage_Flags(kvImageNoFlags),
        &error) else {
    fatalError("Unable to create multidimensional table \(error).")
}

defer {
    vImageMultidimensionalTable_Release(lookupTable)
}
```

The lookup table structure is immutable and thread-safe, and therefore you can use it with multiple and concurrent calls to the appropriate transform function. After you finish using the lookup table, call `vImageMultidimensionalTable_Release(_:)` to free its resources.

Apply the transform

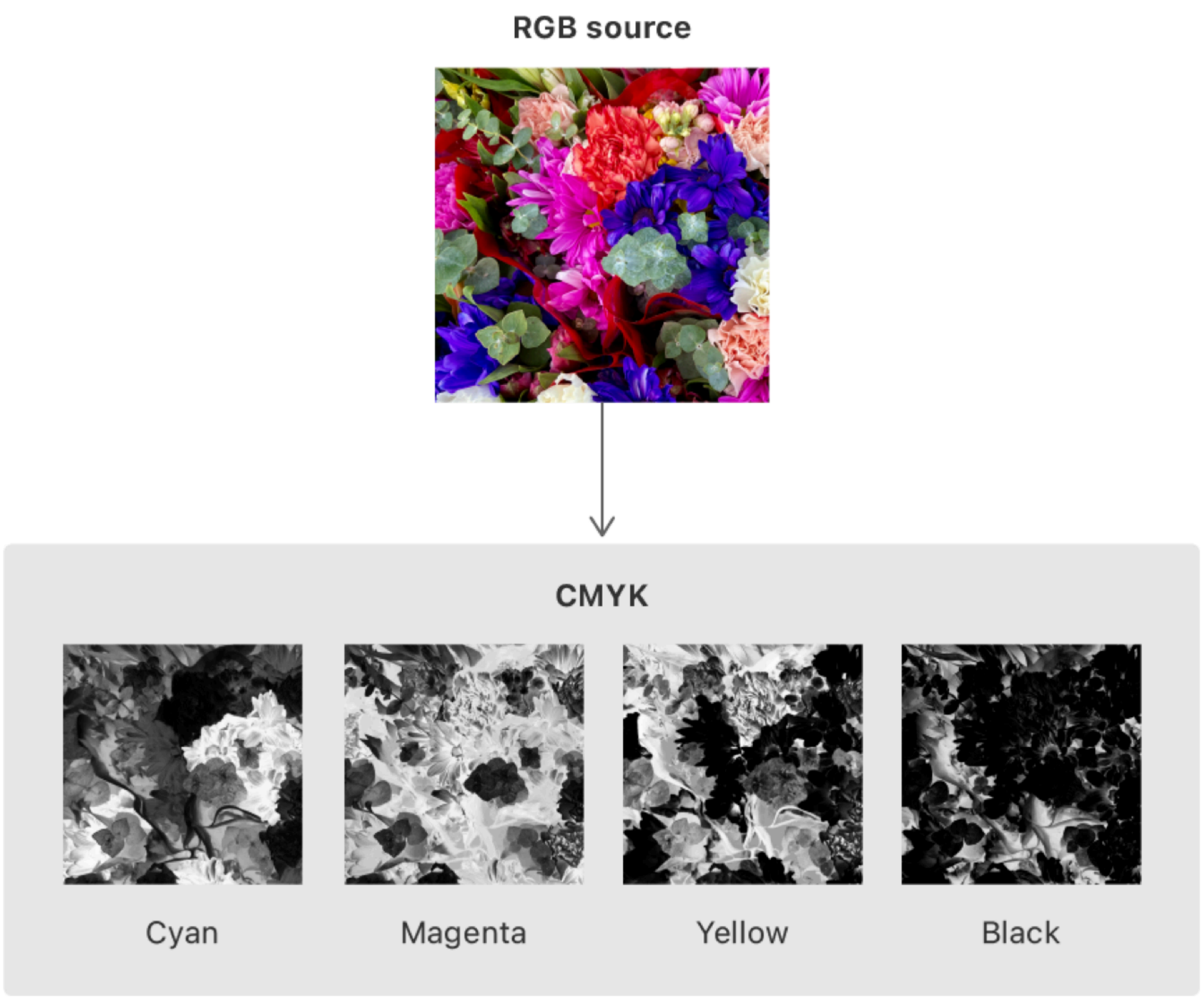
vImage provides two functions to apply the multidimensional lookup table to an image: `vImageMultiDimensionalInterpolatedLookupTable_PlanarF(_:_:_:_:_:_:)` for 32-bit planar buffers, and `vImageMultiDimensionalInterpolatedLookupTable_Planar16Q12(_:_:_:_:_:_:)` for 16Q12 planar buffers.

The code below applies the lookup table to three 32-bit source buffers (`srcRedBuffer`, `srcGreenBuffer`, and `srcBlueBuffer`) and writes the result to four 32-bit destination buffers (`destCyanBuffer`, `destMagentaBuffer`, `destYellowBuffer`, and `destBlackBuffer`). For more information about working with planar buffers, see Optimizing image-processing performance.

```
error = vImageMultiDimensionalInterpolatedLookupTable_PlanarF(
    [srcRedBuffer, srcGreenBuffer, srcBlueBuffer],
    [destCyanBuffer, destMagentaBuffer, destYellowBuffer, destBlackBuffer],
    nil,
    lookupTable,
    kvImageFullInterpolation,
    vImage_Flags(kvImageNoFlags))

if error != kvImageNoError {
    fatalError("Error calling transform function` \(error).")
}
```

On return, the four destination planar buffers contain the cyan, magenta, yellow, and black channels of the original RGB source image, as the following image shows:



Apply multidimensional lookup tables to pixel buffers

If you're creating apps for recent operating systems that support the `vImage.PixelBuffer` API, the vImage library includes the `vImage.MultidimensionalLookupTable` structure, which provides a simple interface to a `vImage_MultidimensionalTable` structure. A `vImage.MultidimensionalLookupTable` structure uses the same lookup table data as a `vImage_MultidimensionalTable` structure. The code below performs an RGB-to-CMYK conversion using 32-bit planar pixel buffers:

```
// An array of three 32-bit planar pixels buffers that contain the RGB source.
let sources: [vImage.PixelBuffer<vImage.PlanarF>] = ...

// An array of four 32-bit planar pixels buffers that represent the CMYK destinatio
let destinations: [vImage.PixelBuffer<vImage.PlanarF>] = ...

let lookupTable = vImage.MultidimensionalLookupTable(
    entryCountPerSourceChannel: tableEntriesPerDimension,
    destinationChannelCount: Int(destChannelCount),
    data: tableData)

lookupTable.apply(sources: sources,
                  destinations: destinations,
```

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.

Converting color images to grayscale

Convert an RGB image to grayscale using matrix multiplication.

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.