Sample Code

Cropping to the subject in a chroma-keyed image

Convert a chroma-key color to alpha values and trim transparent pixels using Accelerate.

Download

macOS 14.0+ | Xcode 15.1+

Overview

The Accelerate framework provides libraries that allow you to isolate objects from a chroma-key background and trim transparent pixels. This technique has applications in fields such as machine learning, where code needs to operate on the smallest useful region of an image to reduce processing time, optimize performance, and minimize energy consumption.

The image on the left below shows an original image with a chroma-key background. The middle image shows the chroma-key converted to alpha values, and the image on the right shows the original image cropped to the smallest bounding box that contains nontransparent pixels:

Convert the chroma-key color to alpha values

The sample code project includes separate functions to convert the chroma-key color to alpha values and to trim the transparent pixels.

The `static chromaKeyToAlpha(source:chromaKeyColor:tolerance:)` function accepts a four-channel ARGB (alpha, red, green, blue) `vImage.PixelBuffer` and uses a `vImage.MultidimensionalLookupTable` to generate a planar pixel buffer that contains the alpha channel.

For more information about multidimensional lookup tables, see Applying color transforms to images with a multidimensional lookup table.

This sample code project uses the red, green, and blue channels of the source image and generates a single-channel output. The code below allocates the memory for the lookup table values:

```
let entriesPerChannel = UInt8(32)
let ramp = vDSP.ramp(in: 0 ... 1.0, count: Int(entriesPerChannel))

let sourceChannelCount = 3
let destinationChannelCount = 1

let lookupTableElementCount = Int(pow(Float(entriesPerChannel),
                                      Float(sourceChannelCount))) * Int(destinationC
```

```
let lookupTableData = UnsafeMutableBufferPointer<UInt16>.allocate(capacity: lookupTa
defer {
    lookupTableData.deallocate()
}
```

The sample calculates the alpha channel value for each permutation of red, green, and blue as a function of the Euclidean distance between each permutation and the chroma-key color in L*a*b* color space.

The code below iterates over red, green, and blue values, and populates each lookup table entry with a corresponding alpha value.

```
let chromaKeyRGB = chromaKeyColor.components ?? [0, 0, 0]
let chromaKeyLab = ColorConverter.rgbToLab(r: chromaKeyRGB[0],
                                           g: chromaKeyRGB.count > 1 ? chromaKeyRGB[
                                           b: chromaKeyRGB.count > 2 ? chromaKeyRGB[

var bufferIndex = 0
for red in ramp {
    for green in ramp {
        for blue in ramp {

            let lab = ColorConverter.rgbToLab(r: red, g: green, b: blue)

            let distance = simd_distance(chromaKeyLab, lab)

            let contrast = Float(20)
            let offset = Float(0.25)
            let alpha = saturate(tanh(((distance / tolerance ) - 0.5 - offset) * cor

            lookupTableData[bufferIndex] = UInt16(alpha * Float(UInt16.max))
            bufferIndex += 1
        }
    }
}
```

Trim the transparent pixels

The `static boundingBoxForNonTransparentPixels(alphaBuffer:)` function accepts an alpha channel pixel buffer and returns the smallest bounding box that contains the pixel buffer's nontransparent pixels. The function calculates the top of the bounding box by iterating over each row of pixels, starting at the top of the alpha buffer. For each iteration, the code calls `sum(_:)`.

When the sum of the alpha values isn't equal to `0`, the corresponding row contains nontransparent pixels and, therefore, that row is the top of the bounding box.

The function uses the same technique to find the bottom of the bounding box, but in this case, it starts from the bottom and works upward.

The code below finds the top and bottom of the bounding box:

```
alphaBuffer.withUnsafeBufferPointer { alphaPointer in

    let rowStride = alphaBuffer.rowStride

    // Find the bounding box top.
    for i in 0 ..< alphaBuffer.height {

        let start = alphaPointer.baseAddress?.advanced(by: i * rowStride)
        let row = UnsafeBufferPointer<Float>(start: start, count: alphaBuffer.width)
        let sum = vDSP.sum(row)

        if sum != 0 {
            top = i
            break
        }
    }

    // Find the bounding box bottom.
    for i in stride(from: alphaBuffer.height - 1, through: top, by: -1) {

        let start = alphaPointer.baseAddress?.advanced(by: i * rowStride)
        let row = UnsafeBufferPointer<Float>(start: start, count: alphaBuffer.width)
        let sum = vDSP.sum(row)

        if sum != 0 {
            bottom = i
            break
        }
    }

    let height = bottom - top
```

To find the left and right sides of the bounding box, the code uses vDSP_sve rather than sum(_:). The vDSP_sve(_:_:_:_:) function supports the `stride` parameter that the code

defines as the pixels buffer's `rowStride`. Setting this value specifies that the summation is computed along the buffer's column rather than row.

Similar to the top and bottom calculation, when the horizontal iterations find a nonzero column sum, the function sets the column index as the left or right edge of the bounding box.

```
    // Find the bounding box left.
    for i in 0 ..< alphaBuffer.width {

        let columnStart = alphaPointer.baseAddress!.advanced(by: i + (top * rowStrid

        var sum = Float()

        vDSP_sve(columnStart, rowStride, &sum, vDSP_Length(height))

        if sum != 0 {
            left = i
            break
        }
    }

    // Find the bounding box right.
    for i in stride(from: alphaBuffer.width - 1, through: 0, by: -1) {

        let columnStart = alphaPointer.baseAddress!.advanced(by: i + (top * rowStrid

        var sum = Float()

        vDSP_sve(columnStart, rowStride, &sum, vDSP_Length(height))

        if sum != 0 {
            right = i
            break
        }
    }
} // Ends `alphaBuffer.withUnsafeBufferPointer { alphaPointer in`.
```

After the code has completed the horizontal and vertical passes, the function returns a `CGRect` structure that represents the smallest bounding box that doesn't contain either a row or column of entirely transparent pixels.

```
return CGRect(x: left,
              y: top,
              width: right - left,
              height: bottom - top)
```

Create the final image

The code below applies the `static chromaKeyToAlpha(source:chromaKeyColor:tolerance:)` and `static boundingBoxForNonTransparentPixels(alphaBuffer:)` functions to a CGImage instance named `originalImage`.

```
// Create an `InterleavedFx4` pixel buffer from the original image.
let sourceBuffer = try! vImage.PixelBuffer<vImage.InterleavedFx4>(
    cgImage: originalImage,
    cgImageFormat: &cgImageFormatARGB)

// Create a `PlanarF` pixel buffer that represents the alpha channel.
let alpha = Self.chromaKeyToAlpha(source: sourceBuffer,
                                  chromaKeyColor: .init(red: 91 / 255,
                                                        green: 155 / 255,
                                                        blue: 244 / 255,
                                                        alpha: 0),
                                  tolerance: 60)

// Compute the bounding box for nontransparent pixels.
let boundingBox = Self.boundingBoxForNonTransparentPixels(alphaBuffer: alpha)

// Create an `InterleavedFx4` pixel buffer that's the cropped version
// of the original image.
let cropped = sourceBuffer.cropped(to: boundingBox)

// Overwrite the alpha channel of the cropped image so that the chroma-key
// background is transparent.
cropped.overwriteChannels([0],
                          withPlanarBuffer: alpha.cropped(to: boundingBox),
                          destination: cropped)

// Create a Core Graphics image of the final result.
outputImage = cropped.makeCGImage(cgImageFormat: cgImageFormatARGB)!
```

On return, `outputImage` contains the original image, without transparent rows and columns, and with the chroma-key background transparent.

See Also

Transforming with a multidimensional lookup table

Applying color transforms to images with a multidimensional lookup table

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

{} Applying transformations to selected colors in an image

Desaturate a range of colors in an image with a multidimensional lookup table.

```
func vImageMultidimensionalTable_Create(UnsafePointer<UInt16>, UInt32, UInt32, UnsafePointer<UInt8>, vImageMDTableUsageHint, vImage_Flags, UnsafeMutablePointer<vImage_Error>!) -> vImage_MultidimensionalTable!
```

Creates a multidimensional lookup table.

```
func vImageMultiDimensionalInterpolatedLookupTable_PlanarF(UnsafePointer<vImage_Buffer>, UnsafePointer<vImage_Buffer>, UnsafeMutableRawPointer!, vImage_MultidimensionalTable, vImage_InterpolationMethod, vImage_Flags) -> vImage_Error
```

Uses a multidimensional lookup table to transform a 32-bit planar image.

```
func vImageMultiDimensionalInterpolatedLookupTable_Planar16Q12(UnsafePointer<vImage_Buffer>, UnsafePointer<vImage_Buffer>, UnsafeMutableRawPointer!, vImage_MultidimensionalTable, vImage_InterpolationMethod, vImage_Flags) -> vImage_Error
```

Uses a multidimensional lookup table to transform a 16Q12 planar image.

```
func vImageMultidimensionalTable_Retain(vImage_MultidimensionalTable!) -> vImage_Error
```

Retains a multidimensional table.

```
func vImageMultidimensionalTable_Release(vImage_MultidimensionalTable!) -> vImage_Error
```

Releases a multidimensional table.

```
typealias vImage_MultidimensionalTable
```

An opaque pointer that represents a multidimensional lookup table.

`struct` `vImageMDTableUsageHint`

Constants that indicate the use for a multidimensional lookup table.

`struct` `vImage_InterpolationMethod`

Constants that represent different interpolation methods.