Article

Applying vImage operations to regions of interest

Limit the effect of vImage operations to rectangular regions of interest.

Overview

You can apply vImage operations, such as blurs and color transforms, to specified rectangular areas in an image, commonly referred to as *regions of interest* (ROI). Limiting the effect of an operation is useful when, for example, you want to overlay user interface elements on top of a blurred part of an image to make them stand out.

The following image is an example of the effects possible when you use the techniques in this article. The image shows a single photograph with a portrait-format ROI that the code has desaturated and a landscape-format ROI that the code has blurred.

Applying operations to an ROI of a pixel buffer

If you're developing apps with Xcode 14.0 or later, the `vImage.PixelBuffer` structure provides the `withUnsafeRegionOfInterest(_:_:)` function, which simplifies applying operations to ROIs. The following code creates the image above:

```
// `source` is a `vImage.PixelBuffer<vImage.Interleaved8x4>` that contains an ARGB88
// `destination` is an initialized `vImage.PixelBuffer<vImage.Interleaved8x4>` that
// size as `source`.

let landscapeROI = CGRect( ... )

source.withUnsafeRegionOfInterest(landscapeROI) { src in
    src.multiply(by: desaturationMatrix,
                 divisor: divisor,
                 preBias: (0, 0, 0, 0),
                 postBias: (0, 0, 0, 0),
                 destination: src)
}
```

```
source.copy(to: destination)

let portraitROI = CGRect( ... )

source.withUnsafeRegionOfInterest(portraitROI) { src in
    destination.withUnsafeRegionOfInterest(portraitROI) { dst in
        src.tentConvolve(kernelSize: .init(width: 100, height: 100),
                         edgeMode: .truncateKernel,
                         destination: dst)
    }
}
```

On return, the destination `vImage.PixelBuffer` contains the final image.

This article implements the remaining code in this article as extensions to the `vImage_Buffer` structure. The two function calls below created the image above:

```
// `source` is a `vImage_Buffer` that contains an ARGB8888 image.

source.desaturate_ARGB8888(regionOfInterest: CGRect( ... ))

let destination = source.blurred_ARGB8888(regionOfInterest: CGRect( ... ),
                                          blurRadius: 100)
```

Apply an in-place operation to an ROI

For vImage routines that can operate in-place (that is, the operation mutates the source buffer's contents), create a mutating function that applies that routine to an ROI. The following code is the function header for a desaturation function based around `vImageMatrixMultiply_ARGB8888( : : : : : : :)`:

```
extension vImage_Buffer {

    mutating func desaturate_ARGB8888(regionOfInterest roi: CGRect) {
```

The function checks that the supplied ROI is within the bounds of the buffer.

```
        guard Int(roi.maxX) <= width && Int(roi.maxY) <= height &&
            Int(roi.minX) >= 0 && Int(roi.minY) >= 0 else {
                print("ROI is out of bounds.")
```

```
            return
        }
```

The following code calculates the first pixel in the source buffer for the ROI:

```
        let bytesPerPixel = 4

        let start = Int(roi.origin.y) * rowBytes +
                    Int(roi.origin.x) * bytesPerPixel
```

The `start` value defines the offset to the data pointer for a second `vImage_Buffer` structure that references the source buffer's data with a size that equals the ROI.

```
        var desaturationBuffer = vImage_Buffer(data: data.advanced(by: start),
                                               height: vImagePixelCount(roi.height),
                                               width: vImagePixelCount(roi.width),
                                               rowBytes: rowBytes)
```

On return, `desaturationBuffer` contains a reference to the data in the source buffer that the supplied ROI defines. Calling `vImageMatrixMultiply_ARGB8888(_:_:_:_:_:_:)` with `desaturationBuffer` as both the source and the destination performs the matrix multiplication on the pixels in the ROI:

```
        let divisor: Int32 = 0x1000

        let desaturationMatrix = [
            0.0722, 0.0722, 0.0722, 0,
            0.7152, 0.7152, 0.7152, 0,
            0.2126, 0.2126, 0.2126, 0,
            0,      0,      0,      1
            ].map {
                return Int16($0 * Float(divisor))
        }

        let error = vImageMatrixMultiply_ARGB8888(&desaturationBuffer,
                                                  &desaturationBuffer,
                                                  desaturationMatrix,
                                                  divisor,
                                                  nil, nil,
                                                  vImage_Flags(kvImageNoFlags))
```

```
        if error != kvImageNoError {
            print("Error: \(error)")
        }
    }
}
```

To learn more about using matrix multiplication to convert color images to grayscale, see Converting color images to grayscale.

The following shows the result of desaturating an ROI:

Apply an out-of-place operation to an ROI

For vImage routines that don't operate in-place, create a non-mutating function that applies that routine to an ROI and returns a new `vImage_Buffer` structure that contains the result.

The following code is the function header for a blurring function that applies `vImageTentConvolve_ARGB8888(_:_:_:_:_:_:_:_:)`:

```
extension vImage_Buffer {
```

```
func blurred_ARGB8888(regionOfInterest roi: CGRect,
                      blurRadius: Int) -> vImage_Buffer? {
```

The function performs the same check as Applying vImage operations to regions of interest on the ROI size:

```
guard Int(roi.maxX) <= width && Int(roi.maxY) <= height &&
    Int(roi.minX) >= 0 && Int(roi.minY) >= 0 else {
        print("ROI is out of bounds.")
        return nil
}
```

`vImage_Buffer.blurred_ARGB8888(regionOfInterest:blurRadius:)` returns a buffer that's the same size as the source. The function copies all source pixels that are outside of the ROI to the destination. The following code creates the buffer that the function returns, and copies the source pixels into the new buffer:

```
guard var destination = try? vImage_Buffer(width: Int(width),
                                           height: Int(height),
                                           bitsPerPixel: 32) else {
                                            return nil
}

let bytesPerPixel = 4

withUnsafePointer(to: self) { src in
    vImageCopyBuffer(src,
                     &destination,
                     bytesPerPixel,
                     vImage_Flags(kvImageNoFlags))
}
```

The out-of-place function uses the same approach as Applying vImage operations to regions of interest, calculate the start of the ROI. The destination buffer for the blur operation references the copied pixels in `destination`:

```
let start = Int(roi.origin.y) * destination.rowBytes +
    Int(roi.origin.x) * bytesPerPixel

var blurDestination = vImage_Buffer(data: destination.data.advanced(by: sta
                                    height: vImagePixelCount(roi.height),
```

```
                                                width: vImagePixelCount(roi.width),
                                                rowBytes: destination.rowBytes)
```

Finally, the `vImageTentConvolve_ARGB8888(_:_:_:_:_:_:_:_:)` function applies the blur to the source and writes the result to `blurDestination`.

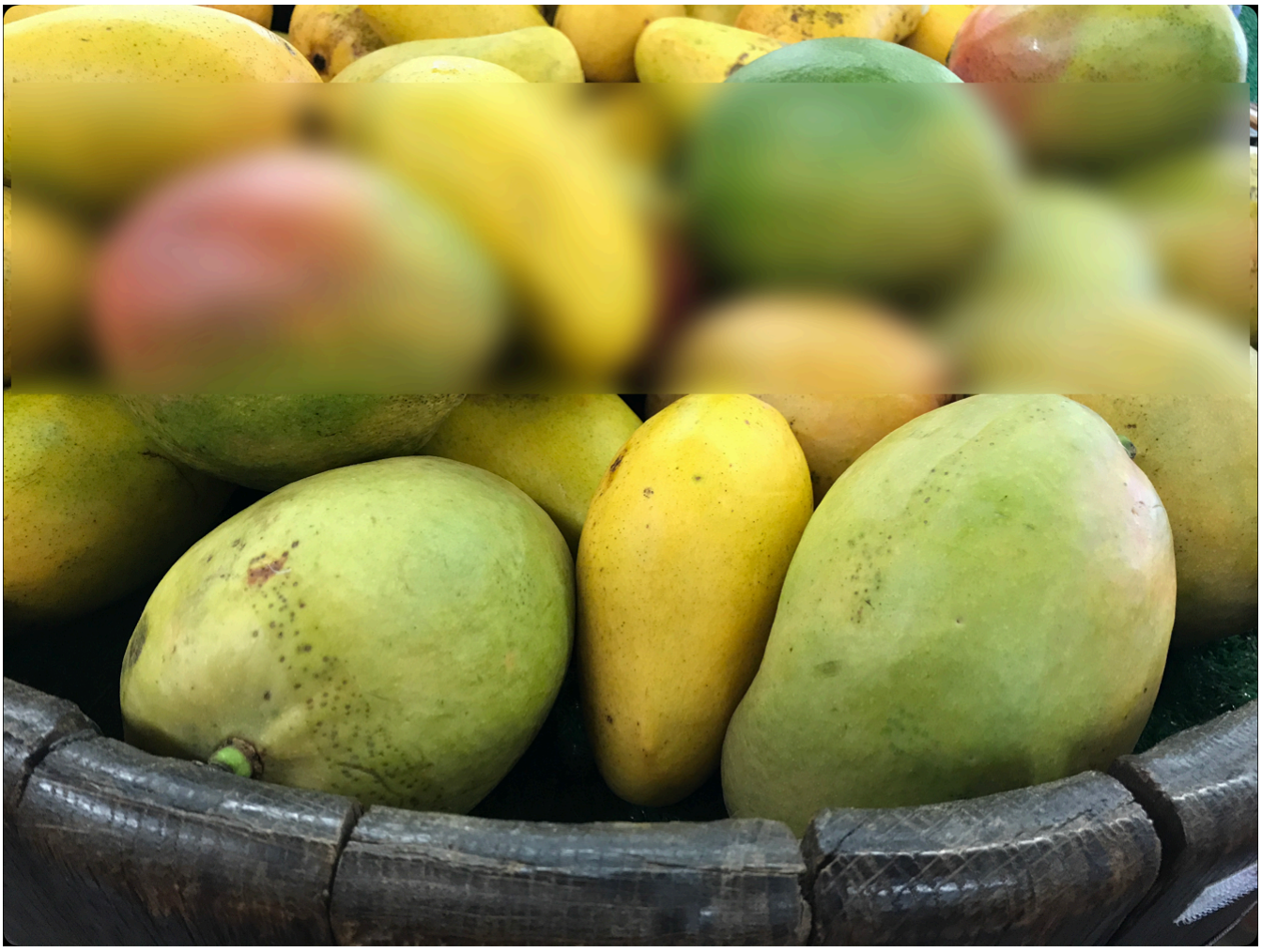
```
        var error = kvImageNoError

        withUnsafePointer(to: self) { src in
            let blurDiameter = UInt32(blurRadius * 2 + 1)
            error = vImageTentConvolve_ARGB8888(src,
                                                &blurDestination,
                                                nil,
                                                vImagePixelCount(roi.origin.x),
                                                vImagePixelCount(roi.origin.y),
                                                blurDiameter, blurDiameter,
                                                [0],
                                                vImage_Flags(kvImageTruncateKernel))
        }

        if error != kvImageNoError {
            destination.free()
            print("Error: \(error)")
            return nil
        }

        return destination
    }
}
```

The following shows the result of blurring an ROI:



See Also

Image Processing Essentials

- Converting bitmap data between Core Graphics images and vImage buffers
  Pass image data between Core Graphics and vImage to create and manipulate images.
- 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.

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.