Sample Code

Finding the sharpest image in a sequence of captured images

Share image data between vDSP and vImage to compute the sharpest image from a bracketed photo sequence.

Download

iOS 16.0+ | iPadOS 16.0+ | Xcode 14.0+

Overview

This sample code project captures a sequence of photographs and uses a combination of routines from vImage and vDSP to order the images by their relative sharpness. This technique is useful in applications such as an image scanner, where your user requires the least blurry captured image. After applying the routines, the app displays the images in a list, with the sharpest image at the top.

This project uses SwiftUI to build the user interface, AVFoundation to capture a sequence of images, and a method known as *the variance of the Laplacian* to determine the sharpness of each image.

Before exploring the code, try building and running the app, and taking photographs of subjects such as documents and signs.

Configure the capture session

The 3 x 3 Laplacian kernel that this sample uses reports a lot of noise if applied to a full-resolution image. To reduce this noise, the sample uses a downscaled image and defines the capture session's preset to a size that's smaller than the camera's native resolution:

```
captureSession.sessionPreset = .hd1280x720
```

To learn more about configuring a capture session, see Setting Up a Capture Session.

Define the photo settings

The sample defines the `AVCapturePhotoBracketSettings` object, which specifies the capture features and settings, in the `BlurDetector.takePhoto()` function.

The sharpness detection algorithm in this sample works on a grayscale image. The camera's YpCbCr pixel formats, either `kCVPixelFormatType_420YpCbCr8BiPlanarVideoRange` or `kCVPixelFormatType_420YpCbCr8BiPlanarFullRange`, represent the luminance of the image using one plane and represent color information on separate planes. The code converts the luminance plane to a grayscale image.

The following code checks that the current device supports one or both of these formats:

```
let pixelFormat: FourCharCode = {
    if photoOutput.availablePhotoPixelFormatTypes
        .contains(kCVPixelFormatType_420YpCbCr8BiPlanarFullRange) {
        return kCVPixelFormatType_420YpCbCr8BiPlanarFullRange
    } else if photoOutput.availablePhotoPixelFormatTypes
        .contains(kCVPixelFormatType_420YpCbCr8BiPlanarVideoRange) {
        return kCVPixelFormatType_420YpCbCr8BiPlanarVideoRange
    } else {
        fatalError("No available YpCbCr formats.")
    }
}()
```

The `exposureSettings` array contains `AVCaptureAutoExposureBracketedStillImageSettings` instances and defines the exposure target bias of each as `currentExposureTargetBias`. The `maxBracketedCapturePhotoCount` property of the `AVCapturePhotoOutput` object defines the maximum number of items in the array.

```
let exposureSettings = (0 ..< photoOutput.maxBracketedCapturePhotoCount).map { _ in
    AVCaptureAutoExposureBracketedStillImageSettings.autoExposureSettings(
        exposureTargetBias: AVCaptureDevice.currentExposureTargetBias)
}
```

The following code uses the array of exposure settings and the first available YpCbCr format type to define the bracketed settings:

```
let photoSettings = AVCapturePhotoBracketSettings(
    rawPixelFormatType: 0,
    processedFormat: [kCVPixelBufferPixelFormatTypeKey as String: pixelFormat],
    bracketedSettings: exposureSettings)
```

The `BlurDetector.takePhoto()` function passes the `AVCapturePhotoBracketSettings` instance to capture the sequence of images:

```
photoOutput.capturePhoto(with: photoSettings,
                         delegate: self)
```

Acquire the captured image

For each captured image, AVFoundation calls the `photoOutput(_:didFinishProcessingPhoto:error:)` method.

The sample uses the `pixelBuffer` property of the `AVCapturePhoto` instance that AVFoundation supplies to acquire the uncompressed `CVPixelBuffer` that contains the captured photograph. While the code is accessing the pixel data of the pixel buffer, it calls `CVPixelBufferLockBaseAddress` to lock the base address:

```
guard let pixelBuffer = photo.pixelBuffer else {
    fatalError("Error acquiring pixel buffer.")
}

CVPixelBufferLockBaseAddress(pixelBuffer,
                             CVPixelBufferLockFlags.readOnly)
```

The pixel buffer that AVFoundation vends contains two planes; the plane at index zero contains the luminance data. Because the sample app runs the sharpness detection code in a background thread, it calls `copyMemory` to create a copy of the luminance data:

```
let width = CVPixelBufferGetWidthOfPlane(pixelBuffer, 0)
let height = CVPixelBufferGetHeightOfPlane(pixelBuffer, 0)
let count = width * height

let lumaBaseAddress = CVPixelBufferGetBaseAddressOfPlane(pixelBuffer, 0)
let lumaRowBytes = CVPixelBufferGetBytesPerRowOfPlane(pixelBuffer, 0)

let lumaCopy = UnsafeMutableRawPointer.allocate(
    byteCount: count,
    alignment: MemoryLayout<Pixel_8>.alignment)
lumaCopy.copyMemory(from: lumaBaseAddress!,
                    byteCount: count)
```

After the code has copied the luminance data, it unlocks the pixel buffer's base address and passes the copied luminance data to the processing function in a background thread:

```
CVPixelBufferUnlockBaseAddress(pixelBuffer,
                               CVPixelBufferLockFlags.readOnly)

Task(priority: .utility) {
    self.processImage(data: lumaCopy,
                      rowBytes: lumaRowBytes,
                      width: width,
                      height: height,
                      sequenceCount: photo.sequenceCount,
                      expectedCount: photo.resolvedSettings.expectedPhotoCount,
                      orientation: photo.metadata[ String(kCGImagePropertyOrientatio

    lumaCopy.deallocate()
}
```

Initialize grayscale source pixel buffer

The following code creates a pixel buffer from data passed to the `BlurDetector.processImage(data:rowBytes:width:height:sequenceCount:expectedCount:orientation:)` function:

```
let imageBuffer = vImage.PixelBuffer(data: data,
                                     width: width,
                                     height: height,
```

```
                                    byteCountPerRow: rowBytes,
                                    pixelFormat: vImage.Planar8.self)
```

On return, `sourceBuffer` contains a grayscale representation of the captured image.

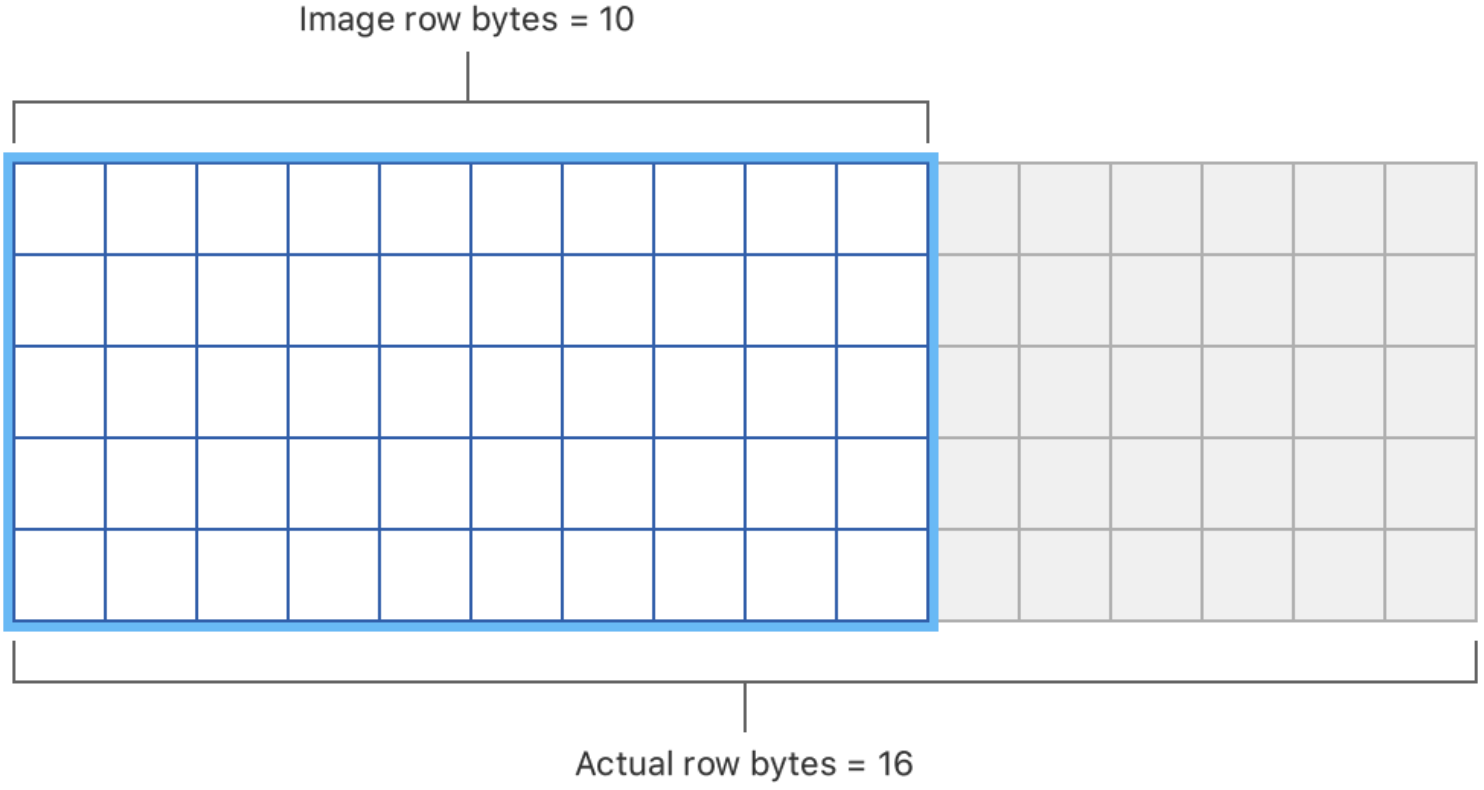
Create floating point pixels to use with vDSP

vImage buffers store their image data in row-major format. However, when you pass data between vImage and vDSP, be aware that, in some cases, vImage will add extra bytes at the end of each row. For example, the following code declares two 8-bit-per-pixel buffers that are 10 pixels wide:

```
let buffer0 = try? vImage_Buffer(width: 10,
                                 height: 5,
                                 bitsPerPixel: 8)

let buffer1 = vImage.PixelBuffer(width: 10,
                                 height: 5,
                                 pixelFormat: vImage.Planar8.self)
```

Although the code defines buffers with 10 bytes per row, to maximize performance, `vImage_Buffer_Init(_:_:_:_:)` and `init(width:height:pixelFormat:)` both initialize a buffer with 16 bytes per row.



In some cases, this disparity between the row bytes used to hold image data and the buffer's actual row bytes may not affect an app's results. However, the sample app declares a `vImage.PixelBuffer` structure with external memory that has no additional padding. This ensures that the uninitialized data in the row padding doesn't affect the blur detection algorithm.

```
var laplacianStorage = UnsafeMutableBufferPointer<Float>.allocate(capacity: width *
let laplacianBuffer = vImage.PixelBuffer(data: laplacianStorage.baseAddress!,
                                         width: width,
                                         height: height,
                                         byteCountPerRow: width * MemoryLayout<Float
                                         pixelFormat: vImage.PlanarF.self)
defer {
    laplacianStorage.deallocate()
}

imageBuffer.convert(to: laplacianBuffer)
```

On return, `laplacianStorage` and `laplacianBuffer` share the same memory that contains a 32-bit version of the image data in the `imageBuffer`.

Perform the convolution

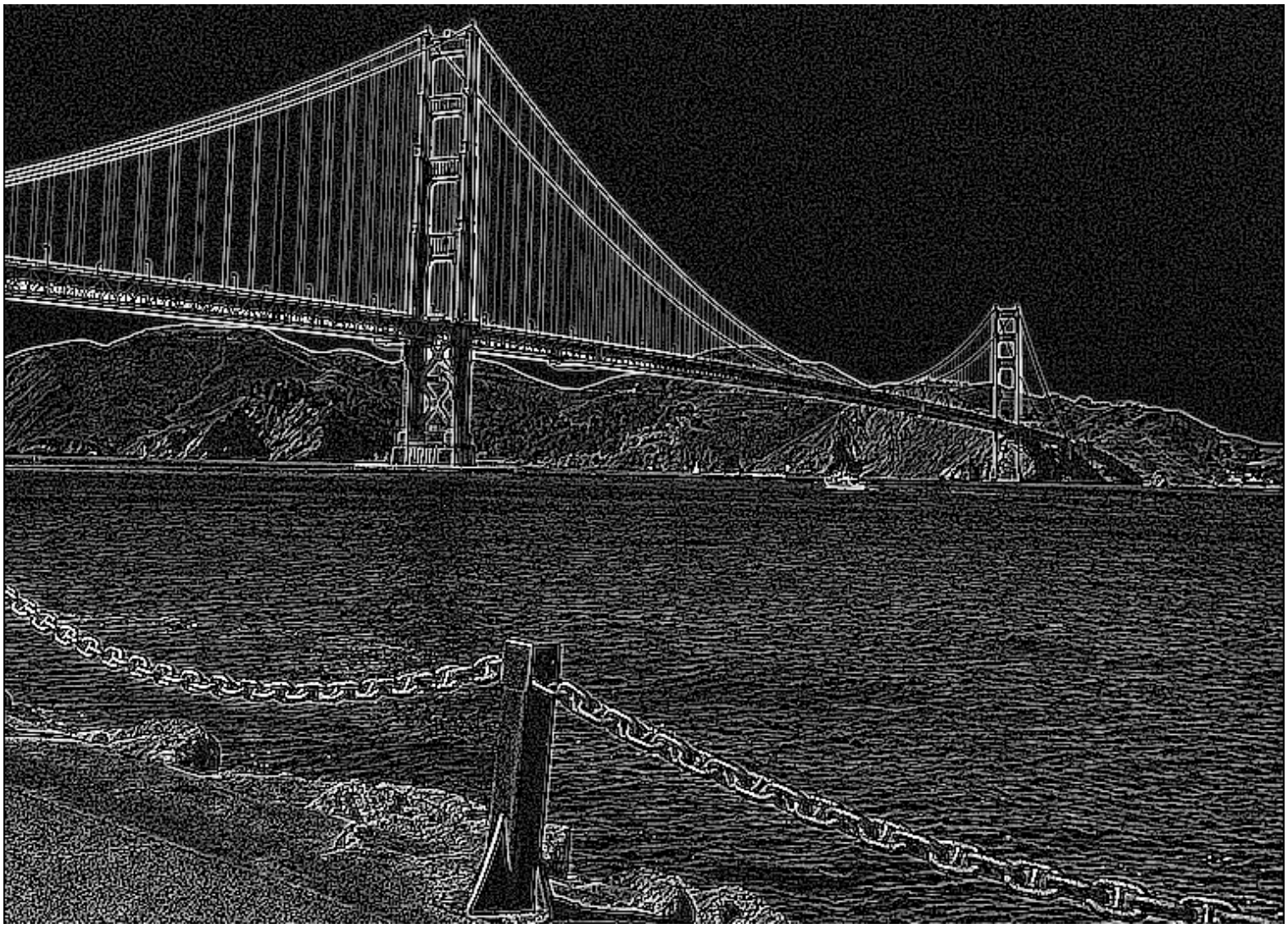
The Laplacian kernel finds edges in the single-precision pixel values:

```
let laplacian: [Float] = [-1, -1, -1,
                          -1,  8, -1,
                          -1, -1, -1]
```

The vDSP convolve function performs the convolution in place on the `laplacianStorage` memory:

```
vDSP.convolve(laplacianStorage,
              rowCount: height,
              columnCount: width,
              with3x3Kernel: laplacian,
              result: &laplacianStorage)
```

After the convolution, edges in the image have high values. The following image shows the result after convolution using the Laplacian kernel:



Calculate the variance

The vDSP_normalize function calculates the standard deviation of the pixel values after the edge detection. The following computed property returns the variance of a single-precision AccelerateMutableBuffer instance:

```
extension AccelerateMutableBuffer where Element == Float {
    var variance: Float {

        var mean = Float.nan
        var standardDeviation = Float.nan

        self.withUnsafeBufferPointer {
            vDSP_normalize($0.baseAddress!, 1,
                           nil, 1,
                           &mean, &standardDeviation,
                           vDSP_Length(self.count))
        }
```

```
        return standardDeviation * standardDeviation
    }
}
```

The sample app uses this value as a measure of relative sharpness. Images with more variance have more detail than those with less variance, and that difference is used to derive the relative sharpness.

Create a display image with the correct orientation

The sample app uses the vImage 90° rotation functions in conjunction with the `CGImage` object's orientation to create Core Graphics images that are suitable for displaying in the app. The `static BlurDetector.makeImage(fromPlanarBuffer:orientation:)` function accepts a planar buffer (either the grayscale representation of the captured image or the result of the convolution) and the orientation, and returns a `CGImage` instance.

For landscape images, meaning images with an orientation of `.left` or `.right`, the function creates a destination buffer with a width equal to the height, and a height equal to the width of the supplied buffer. For portrait images, meaning images with an orientation of `.up` or `.down`, the function creates a destination buffer with the same dimensions as the supplied buffer.

```
var outputBuffer: vImage.PixelBuffer<Format>
var outputRotation: Int

if orientation == .right || orientation == .left {
    outputBuffer = vImage.PixelBuffer<Format>(width: sourceBuffer.height,
                                              height: sourceBuffer.width)

    outputRotation = orientation == .right ?
            kRotate90DegreesClockwise : kRotate90DegreesCounterClockwise
} else if orientation == .up || orientation == .down {
    outputBuffer = vImage.PixelBuffer<Format>(width: sourceBuffer.width,
                                              height: sourceBuffer.height)
    outputRotation = orientation == .down ?
            kRotate180DegreesClockwise : kRotate0DegreesClockwise
} else {
    return nil
}
```

The following code populates the destination buffer using either `vImageRotate90_Planar8(_:_:_:_:_:)` or `vImageRotate90_PlanarF(_:_:_:_:_:)`

```
let imageFormat: vImage_CGImageFormat

let rotateFunction: (UnsafePointer<vImage_Buffer>,
                     UnsafePointer<vImage_Buffer>,
                     UInt8) -> vImage_Error

if Format.self == vImage.Planar8.self {
    imageFormat = vImage_CGImageFormat(
        bitsPerComponent: 8,
        bitsPerPixel: 8,
        colorSpace: CGColorSpaceCreateDeviceGray(),
        bitmapInfo: .init(rawValue: CGImageAlphaInfo.none.rawValue))!

    func rotate (src: UnsafePointer<vImage_Buffer>,
                 dst: UnsafePointer<vImage_Buffer>,
                 rotation: UInt8) -> vImage_Error {
        vImageRotate90_Planar8(src, dst, rotation, 0, 0)
    }
    rotateFunction = rotate
} else if Format.self == vImage.PlanarF.self {
    imageFormat = vImage_CGImageFormat(
        bitsPerComponent: 32,
        bitsPerPixel: 32,
        colorSpace: CGColorSpaceCreateDeviceGray(),
        bitmapInfo: CGBitmapInfo(rawValue:
                                    kCGBitmapByteOrder32Host.rawValue |
                                    CGBitmapInfo.floatComponents.rawValue |
                                    CGImageAlphaInfo.none.rawValue))!

    func rotate (src: UnsafePointer<vImage_Buffer>,
                 dst: UnsafePointer<vImage_Buffer>,
                 rotation: UInt8) -> vImage_Error {
        vImageRotate90_PlanarF(src, dst, rotation, 0, 0)
    }
    rotateFunction = rotate
} else {
    fatalError("This function only supports Planar8 and PlanarF formats.")
}

sourceBuffer.withUnsafePointerToVImageBuffer { src in
    outputBuffer.withUnsafePointerToVImageBuffer { dst in
        _ = rotateFunction(src, dst, UInt8(outputRotation))
    }
```

```
}
```

Finally, the function returns a `CGImage` from the destination buffer:

```
return outputBuffer.makeCGImage(cgImageFormat: imageFormat)
```

See Also

vImage / vDSP Interoperability

{} Visualizing sound as an audio spectrogram

Share image data between vDSP and vImage to visualize audio that a device microphone captures.