Article

Applying geometric transforms to images

Reflect, shear, rotate, and scale image buffers using vImage.

Overview

The vImage library provides a suite of functions to apply different geometric transforms to images. The image below shows examples of reflect, scale, rotate, and composite translate-rotate-scale transforms:

By default, vImage uses the Lanczos-3 algorithm when resampling. If you need the higher-quality results of the Lanczos-5 algorithm and don't mind slightly slower performance, pass the `kvImageHighQualityResampling` flag to the geometric transform operations.

The vImage geometry operations don't work in-place, that is, you can't use the same buffer as the source and destination.

If you're applying a geometric transform to an image with premultiplied alpha, you may see artifacts in high-frequency regions of the image. To avoid these artifacts, call `vImageUnpremultiplyData_ARGB8888(_:_:_:)` to remove the premultiplied alpha value from the image data before the operation. After the operation, call `vImagePremultiplyData_ARGB8888(_:_:_:)` to premultiply the result.

Reflect a vImage buffer

The vImage reflect functions generate either a vertical or a horizontal reflection of the image. The image below shows the original buffer contents and the contents after vertical reflection:

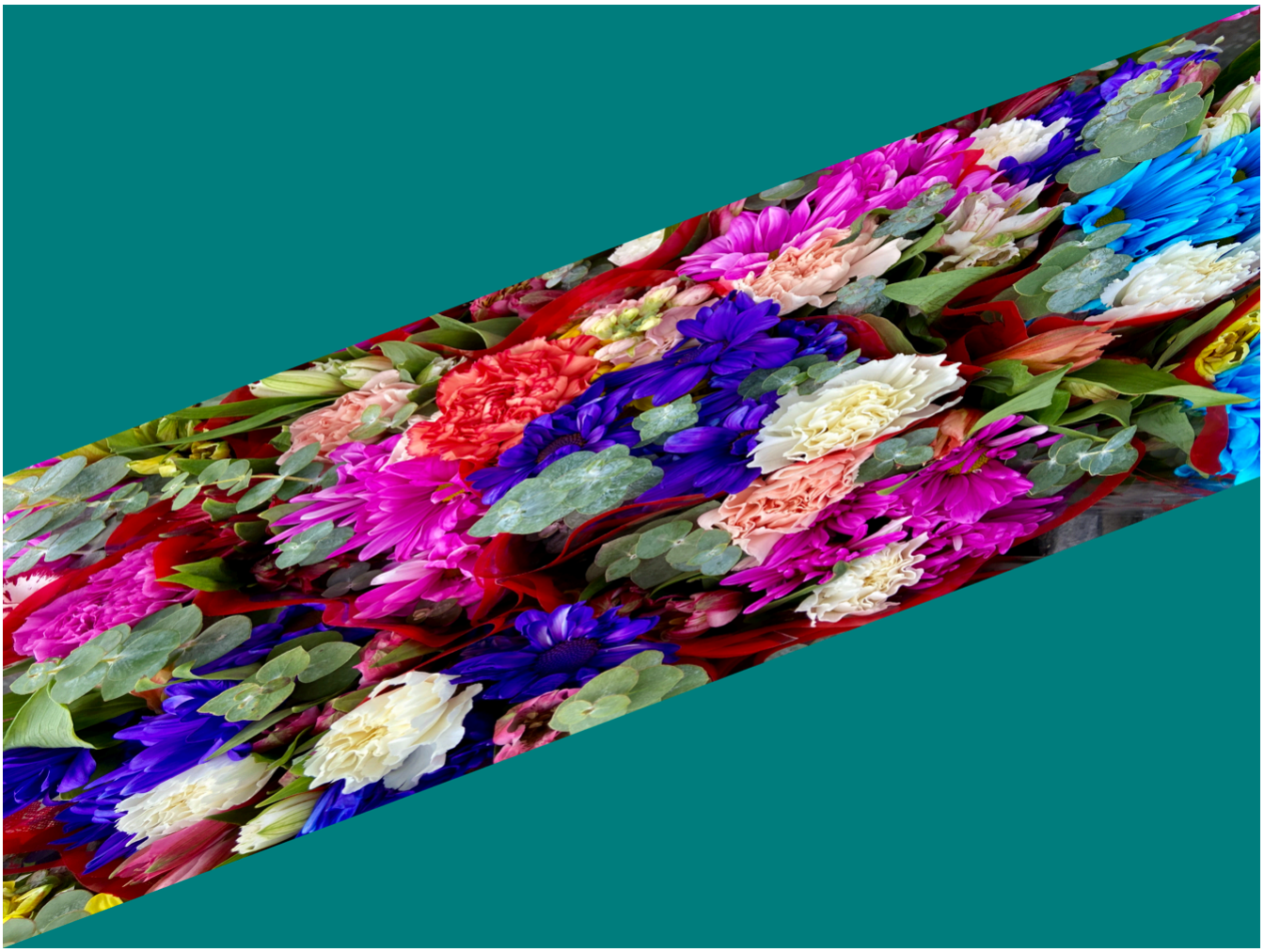
The destination buffer you pass to the vImage reflect functions needs to be the same size as the source buffer. The code below uses a precondition to ensure that both buffer sizes match:

```
static func verticalReflectBuffer(source: vImage_Buffer,
                                  destination: inout vImage_Buffer) {

    precondition(source.size == destination.size,
                 "Source and destination buffers need to have the same size.")

    _ = withUnsafePointer(to: source) { srcPointer in
        vImageVerticalReflect_ARGB8888(srcPointer,
                                       &destination,
                                       vImage_Flags(kvImageNoFlags))
    }
}
```

To learn more about reflection functions, see Image reflection.

Shear a vImage buffer

The vImage shear functions scale and offset an image in one dimension. These functions accept a resampling filter that you use for fine control over the resampling algorithm. For more information on resampling, see Reducing artifacts with custom resampling filters.



Control the amount of shear by specifying the `shearSlope` parameter. The functions define the shear slope as $delta_y/delta_x$ that's the slope for a linear function `y = slope * x`. The `verticalShearBuffer(source:destination:byAngleInDegrees:verticalScale:backgroundColor:)` function accepts the shear as an angle in degrees and performs the transform.

```
static func verticalShearBuffer(source: vImage_Buffer,
                                 destination: inout vImage_Buffer,
                                 byAngleInDegrees angleInDegrees: Double,
                                 verticalScale: Float = 1,
                                 backgroundColor: [Pixel_8] = [0, 127, 127, 127]) {

    // 1. Ensure the shear angle is valid.
    precondition(angleInDegrees > -90 && angleInDegrees < 90,
                 "The shear angle needs to be greater than -90º and less than 90º.")

    // 2. Calculate `shearSlope` as the tangent of the specified angle.
```

```
    let angle = Measurement(value: angleInDegrees,
                            unit: UnitAngle.degrees)
    let radians = Float(angle.converted(to: .radians).value)
    let shearSlope = tan(radians)

    // 3. Create a default resampling filter using the specified scale.
    let resamplingFilter = vImageNewResamplingFilter(verticalScale,
                                                     vImage_Flags(kvImageNoFlags))
    defer {
        vImageDestroyResamplingFilter(resamplingFilter)
    }

    // 4. Apply the transform to `source` and write the result to `destination`.
    _ = withUnsafePointer(to: source) { srcPointer in
        vImageVerticalShear_ARGB8888(srcPointer,
                                     &destination,
                                     0, 0,
                                     0,
                                     shearSlope,
                                     resamplingFilter,
                                     backgroundColor,
                                     vImage_Flags(kvImageBackgroundColorFill))
    }
}
```

To fit the sheared image in a buffer with the same dimensions as the original image, specify a shear angle that's the arctangent of the image's height divided by twice its width.

```
let shearAngle = atan(Double(sourceImageBuffer.height) /
                      Double(sourceImageBuffer.width * 2)) *
                      180 / .pi

verticalShearBuffer(source: sourceImageBuffer,
                    destination: &destinationImageBuffer,
                    byAngleInDegrees: shearAngle,
                    verticalScale: 0.5)
```

On return, the destination buffer contains the image below:

To learn more about shearing functions, see Image shearing.

Rotate a vImage buffer by multiples of 90°

The vImage library provides 90° rotation functions that perform a simple 0°, 90°, 180°, or 270° rotation of an image around its center. The image below shows a buffer's contents without any rotation, rotated 90° counterclockwise, rotated 180°, and rotated 90° clockwise:

Note that the 90° and 270° rotations don't rotate around the true center of the image when the parity (that is, whether an integer is even or odd) of the source width and destination height don't match. The same is true if the parity of the source height and destination width don't match.

If the source and destination buffers are different sizes, the 0° and 180° rotations require that the two heights have the same parity and the two widths have the same parity.

The 90° rotation function crops source pixels that lie outside the destination buffer and fills destination pixels with the specified background color when source pixels don't cover them.

The function below applies a multiple of 90° rotation to a buffer and returns the result in a correctly oriented destination buffer:

```
static func rotateNinety(source: vImage_Buffer,
                         rotation: Int) -> vImage_Buffer? {

    // 1. Create the destination buffer.
```

```
    guard var destination: vImage_Buffer = {
        switch rotation {
            case kRotate0DegreesClockwise, kRotate180DegreesClockwise:
                return try? vImage_Buffer(size: source.size,
                                          bitsPerPixel: 8 * 4)
            case kRotate90DegreesClockwise, kRotate270DegreesClockwise:
                return try? vImage_Buffer(width: Int(source.size.height),
                                          height: Int(source.size.width),
                                          bitsPerPixel: 8 * 4)
            default:
                NSLog("Unsupported rotation constant: \(rotation).")
                return nil
        }
    }() else {
        NSLog("Unable to initialize destination buffer.")
        return nil
    }

    // 2. Apply the transform to `source` and write the result to `destination`.
    _ = withUnsafePointer(to: source) { sourcePtr in
        vImageRotate90_ARGB8888(sourcePtr,
                                &destination,
                                UInt8(rotation),
                                [0],
                                vImage_Flags(kvImageNoFlags))
    }

    return destination
}
```

To learn more about rotation functions, see Image rotation.

Rotate a vImage buffer by an arbitrary angle

The vImage library provides rotation functions that rotate an image by any angle around its center. The image below shows a buffer rotated 60° counterclockwise:

The function below rotates a buffer by the specified angle and writes the result to the destination:

```
static func rotateBuffer(source: vImage_Buffer,
                         destination: inout vImage_Buffer,
                         byAngleInDegrees angleInDegrees: Double,
                         backgroundColor: [Pixel_8] = [0, 127, 127, 127]) {

    let angle = Measurement(value: angleInDegrees,
                            unit: UnitAngle.degrees)
    let radians = Float(angle.converted(to: .radians).value)

    _ = withUnsafePointer(to: source) { srcPointer in
        vImageRotate_ARGB8888(srcPointer,
                              &destination,
                              nil,
                              radians,
                              backgroundColor,
                              vImage_Flags(kvImageBackgroundColorFill))
    }
}
```

Scale a vImage buffer

The vImage library provides functions to scale the contents of an image buffer. The scaling can either be uniform, where the operation preserves the image's aspect ratio, or nonuniform, where the operation stretches or condenses the image. The image below shows four scaled versions of the same photograph, with the second and third image scaled nonuniformly:

The vImage scale functions accept source and destination buffers as parameters and rescale the source to fill the destination buffer. The function below wraps `vImageScale_ARGB8888(_:_:_:_:)` to provide a simple interface to the scale operation:

```
static func scaleBuffer(source: vImage_Buffer,
                        destination: inout vImage_Buffer) {

    _ = withUnsafePointer(to: source) { sourcePointer in
        vImageScale_ARGB8888(sourcePointer,
                             &destination,
                             nil,
                             vImage_Flags(kvImageNoFlags))
    }
}
```

To learn more about scaling functions, see Image scaling.

Apply a simple affine transformation to a vImage buffer

Use an affine transformation to apply composite transformation, such as a scale and a translate.

A 3-by-3 matrix represents an affine transformation.

$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ t_x & t_y & 1 \end{bmatrix}$$

Because the third column is always `(0,0,1)`, the `vImage_CGAffineTransform` data structure contains values for only the first two columns.

To perform a scale transformation, set the *a* and *d* parameters of the matrix to the required scale.

$$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

To perform the translate transformation, set the *tx* and *ty* parameters of the matrix to the required offset.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{bmatrix}$$

The `scaleAndCenterBuffer(source:destination:to:backgroundColor:)` function accepts a scale parameter and sets that parameter as the *a* and *d* parameters of the matrix.

```
static func scaleAndCenterBuffer(source: vImage_Buffer,
                                 destination: inout vImage_Buffer,
                                 to scale: Double,
                                 backgroundColor: [Pixel_8] = [0, 127, 127, 127]) {

    // 1. Calculate the translate required to center the scaled buffer.
    let sourceCenter = SIMD2<Double>(
        x: Double(source.size.width / 2),
        y: Double(source.size.height / 2))

    let desinationCenter = SIMD2<Double>(
        x: Double(destination.size.width / 2),
        y: Double(destination.size.height / 2))

    let tx = desinationCenter.x - sourceCenter.x * scale
    let ty = desinationCenter.y - sourceCenter.y * scale

    // 2. Create the affine transformation that represents the scale-translate.
    var vImageTransform = vImage_CGAffineTransform(
        a: scale, b: 0,
        c: 0,     d: scale,
        tx: tx,   ty: ty)
```

```
    // 3. Apply the transform to `source` and write the result to `destination`.
    _ = withUnsafePointer(to: source) { srcPointer in
        vImageAffineWarpCG_ARGB8888(srcPointer,
                                    &destination,
                                    nil,
                                    &vImageTransform,
                                    backgroundColor,
                                    vImage_Flags(kvImageBackgroundColorFill))
    }
}
```

The image below shows the result of applying `scaleAndCenterBuffer(_:to:backgroundColor:)` with a scale of `0.25`:

To learn more about affine transformation functions, see Applying affine transformations to images.

Apply a complex affine transformation to a vImage buffer

For complex transformations, `CGAffineTransform` allows you to concatenate a series of transformations. For example, to rotate and scale an image to fit within its unrotated bounding box,

your code needs to translate, rotate, scale, and translate again.

The `rotateAndScaleBuffer(source:destination:byAngleInDegrees:backgroundColor:)` function uses `CGAffineTransform` to build the matrix required to rotate and scale an image to fit inside the bounding box of the destination buffer.

```
static func rotateAndScaleBuffer(source: vImage_Buffer,
                                 destination: inout vImage_Buffer,
                                 byAngleInDegrees angleInDegrees: Double,
                                 backgroundColor: [Pixel_8] = [0, 127, 127, 127]) {

    // 1. Convert the specified angle in degrees to radians.
    let angle = Measurement(value: angleInDegrees,
                            unit: UnitAngle.degrees)
    let radians = CGFloat(angle.converted(to: .radians).value)

    // 2. Calculate the scale based on the bounding box of the rotated image.
    let rotatedBoundingBox = CGRect(origin: .zero, size: source.size)
        .applying(CGAffineTransform(rotationAngle: radians))
    let scale = min(destination.size.width / rotatedBoundingBox.size.width,
                    destination.size.height / rotatedBoundingBox.size.height)

    // 3. Create the composite affine transformation.
    let cgTransform = CGAffineTransform.identity
        .translatedBy(x: destination.size.width / 2,
                      y: destination.size.height / 2)
        .rotated(by: radians)
        .scaledBy(x: scale, y: scale)
        .translatedBy(x: -source.size.width / 2,
                      y: -source.size.height / 2)

    // 4. Convert the `CGAffineTransform` to a `vImage_CGAffineTransform`.
    var vImageTransform = vImage_CGAffineTransform(
        a: Double(cgTransform.a),
        b: Double(cgTransform.b),
        c: Double(cgTransform.c),
        d: Double(cgTransform.d),
        tx: Double(cgTransform.tx),
        ty: Double(cgTransform.ty))

    // 5. Apply the transform to `source` and write the result to `destination`.
    _ = withUnsafePointer(to: source) { srcPointer in
        vImageAffineWarpCG_ARGB8888(srcPointer,
```

```
                                    &destination,
                                    nil,
                                    &vImageTransform,
                                    backgroundColor,
                                    vImage_Flags(kvImageBackgroundColorFill))
    }
}
```

The image below shows the result of applying `rotateAndScaleBuffer(_:byAngleInDegrees:backgroundColor:)` with an angle of 30°:

Optimize performance with temporary buffers

The vImage rotate, scale, and affine transform functions use multiple-pass algorithms that save intermediate pixel values between passes. In some cases, the destination buffer may not be large enough to store that intermediate data, so the operation requires additional, temporary storage.

Pass `nil` to the function to have vImage create and manage this temporary storage for you. For example, the scale function below relies on the operation to manage the temporary buffer:

```
static func scaleBuffer(source: vImage_Buffer,
                        destination: inout vImage_Buffer) {

    _ = withUnsafePointer(to: source) { sourcePointer in
        vImageScale_ARGB8888(sourcePointer,
                             &destination,
                             nil,
                             vImage_Flags(kvImageNoFlags))
    }
}
```

In cases where your code calls the function frequently (for example, when processing video), create and manage this temporary buffer yourself and reuse it across function calls. Reusing a buffer avoids vImage allocating the temporary storage with each call. Pass `kvImageGetTempBufferSize` to your geometry function to calculate the size, in bytes, of the temporary buffer, and use that value to allocate an `UnsafeMutableRawPointer` structure.

The version of the `scaleBuffer()` function below shows an example of creating a temporary buffer:

```
static func scaleBuffer(source: vImage_Buffer,
                        destination: inout vImage_Buffer) {

    withUnsafePointer(to: source) { sourcePointer in
        let bufferSize = vImageScale_ARGB8888(sourcePointer,
                                              &destination,
                                              nil,
                                              vImage_Flags(kvImageGetTempBufferSize)


        if bufferSize < 0 {
            fatalError("Error calculating buffer size for scale operation (\(bufferS
        }

        // In a real app, you reuse this buffer across multiple calls of `vImageScal
        let temporaryBuffer = UnsafeMutableRawPointer.allocate(byteCount: bufferSize
                                                               alignment: 64)
        defer {
            temporaryBuffer.deallocate()
        }

        vImageScale_ARGB8888(sourcePointer,
                             &destination,
```

```
                                temporaryBuffer,
                                vImage_Flags(kvImageNoFlags))
    }
}
```

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