Sample Code

# Blurring an image

Filter an image by convolving it with custom and high-speed kernels.

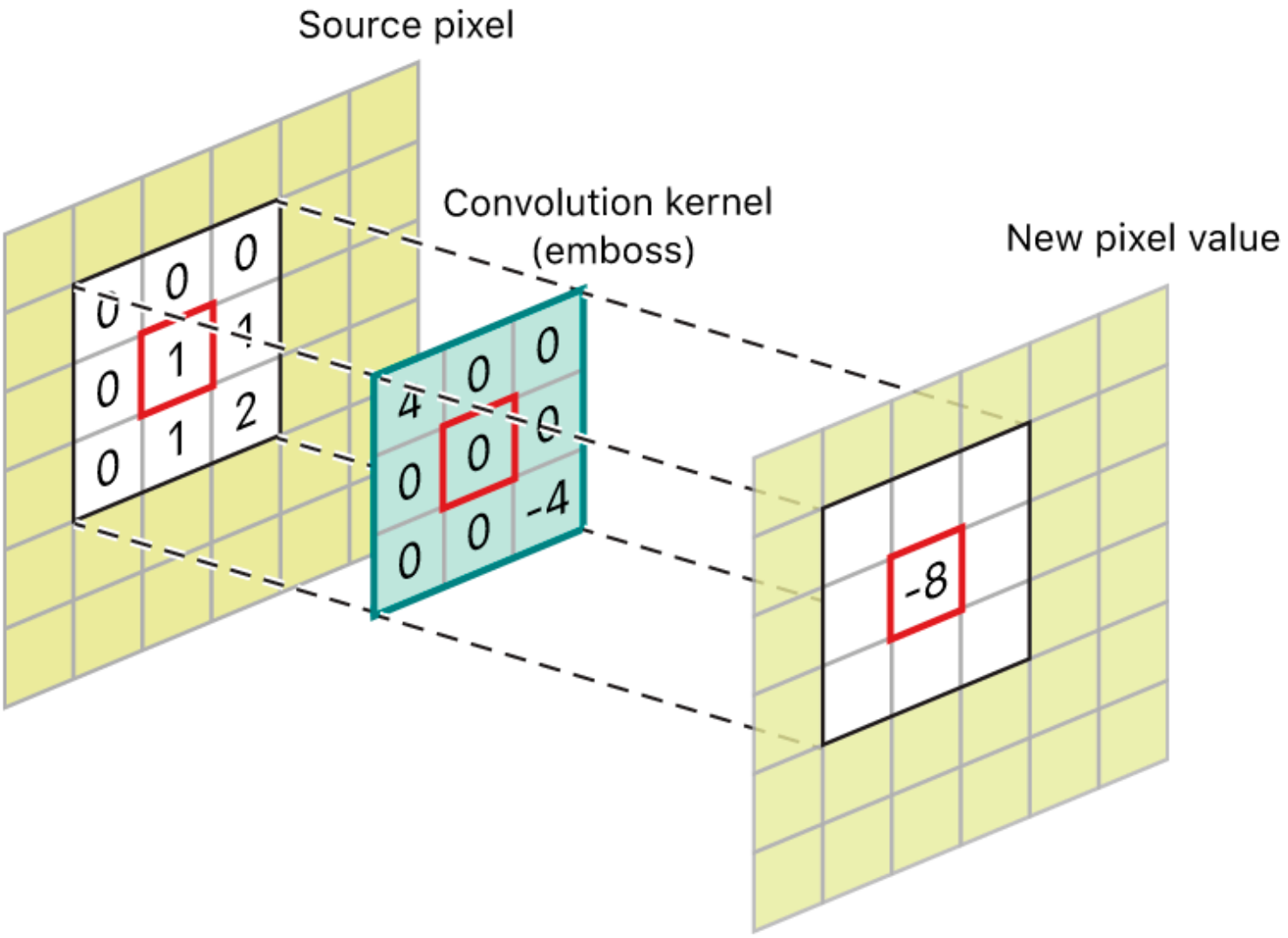
Download

macOS 13.3+ | Xcode 14.3+

## Overview

This sample code project uses a variety of convolution techniques to blur an image with custom kernels and built-in high-speed kernels. *Convolution* is a common image-processing technique that changes the value of a pixel according to the values of its surrounding pixels. Many common image filters, such as blurring, detecting edges, sharpening, and embossing, derive from convolution.

*Kernels* form the basis of convolution operations. Kernels are 1D or 2D grids of numbers that indicate the influence of a pixel's neighbors on its final value. To calculate the value of each transformed pixel, add the products of each surrounding pixel value with the corresponding kernel value. During a convolution operation, the kernel passes over every pixel in the image, repeating this procedure, and then applies the effect to the entire image.




Kernels don't need to have the same height and width, and can be 1D (that is, either the height or the width is 1) or 2D (that is, both the height and the width are greater than 1). When a convolution operation transforms a pixel, both dimensions must be odd numbers to center the kernel over the pixel.

The simplest kernel, known as an *identity kernel*, contains a single value: 1. The following formula shows the result when applying the kernel to the central value in a grid of nine values. It multiplies the pixel by the central value in the convolution kernel, and then multiplies the surrounding pixel values by 9. The sum of these values is 0.5.

$$\begin{bmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{bmatrix} \otimes \begin{bmatrix} 0.0 & 0.5 & 0.0 \\ 0.5 & 0.5 & 0.5 \\ 0.0 & 0.5 & 0.0 \end{bmatrix} = [0.5]$$

An image remains unchanged when convolving it with an identity kernel.

## Run the sample

To convolve an image, select a blur filter from the SwiftUI `Picker` control.

## Blur an image with a 2D kernel

A *box blur kernel* returns the average value of the neighboring pixels. In the following example, the kernel contains nine values and the result is the sum of 1 divided by 9 multiplied by each of the pixel values:

$$\begin{bmatrix} 0.111 & 0.111 & 0.111 \\ 0.111 & 0.111 & 0.111 \\ 0.111 & 0.111 & 0.111 \end{bmatrix} \otimes \begin{bmatrix} 0.0 & 0.5 & 0.0 \\ 0.5 & 0.5 & 0.5 \\ 0.0 & 0.5 & 0.0 \end{bmatrix} = [0.277]$$

Note that the sum of the values in the convolution kernel above is 1 — that is, the kernel is *normalized*. If the sum of the values is greater than 1, the resulting image is brighter than the source. If the sum is less than 1, the resulting image is darker than the source.

A more complex blurring kernel varies the influence of pixels according to their distance from the center of the kernel, and yields a smoother blurring effect. The following kernel (based on a Hann window) is suitable for use with an integer format (for example, `vImage.Interleaved8x4`) convolution:

```
let kernel2D: [Int16] = [
    0,    0,    0,      0,      0,      0,      0,
    0,    2025, 6120,   8145,   6120,   2025,   0,
    0,    6120, 18496,  24616,  18496,  6120,   0,
    0,    8145, 24616,  32761,  24616,  8145,   0,
    0,    6120, 18496,  24616,  18496,  6120,   0,
    0,    2025, 6120,   8145,   6120,   2025,   0,
    0,    0,    0,      0,      0,      0,      0
]
```

The example below shows the result of blurring an image using `kernel2D`:

The sample passes kernels as arrays of integers to the integer format convolution filters. To normalize an integer kernel, the sample passes a divisor to the function that is the sum of the elements of the kernel.

```
let divisor = weights.map { Int32($0) }.reduce(0, +)
```

The following example shows how to use `convolve(with:divisor:bias:edgeMode:destination:)` to perform a convolution and populate a destination buffer with the result:

```
sourceBuffer.convolve(with: kernel,
                      divisor: divisor,
                      edgeMode: .extend,
                      destination: destinationBuffer)
```

## Blur an image with a separable kernel

The `kernel2D` kernel described in the previous section is *separable*; that is, it's the *outer product* of a 1D horizontal kernel and a 1D vertical kernel. A separable kernel allows splitting of the 2D convolution into two 1D passes, resulting in faster processing times. The following formula shows the two vectors that form `kernel2D`:

$$[0 \quad 45 \quad 136 \quad 181 \quad 136 \quad 45 \quad 0] \otimes \begin{bmatrix} 0 \\ 45 \\ 136 \\ 181 \\ 136 \\ 45 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2025 & 6120 & 8145 & 6120 & 2025 & 0 \\ 0 & 6120 & 18496 & 24616 & 18496 & 6120 & 0 \\ 0 & 8145 & 24616 & 32761 & 24616 & 8145 & 0 \\ 0 & 6120 & 18496 & 24616 & 18496 & 6120 & 0 \\ 0 & 2025 & 6120 & 8145 & 6120 & 2025 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The separable convolution functions in vImage work on planar buffers. The sample uses the following code to create planar source and destination buffers, and to convert the interleaved source image to planar:

```
let planarSourceBuffers = vImage.PixelBuffer<vImage.Planar8x4>(size: sourceBuffer.si
let planarDestinationBuffers = vImage.PixelBuffer<vImage.Planar8x4>(size: sourceBuft

sourceBuffer.deinterleave(destination: planarSourceBuffers)
```

To learn more about working with planar buffers, see Optimizing image-processing performance.

The sample declares this 1D kernel with the following code:

```
let kernel1D: [Float] = [0, 45, 136, 181, 136, 45, 0]
```

Note that the kernel for the separable convolution uses single-precision values. This allows for increased precision compared to the 2D integer convolution functions.

To apply a blur using a pair of 1D kernels, the sample calls `separableConvolve(horizontalKernel:verticalKernel:bias:edgeMode:destination:)`.

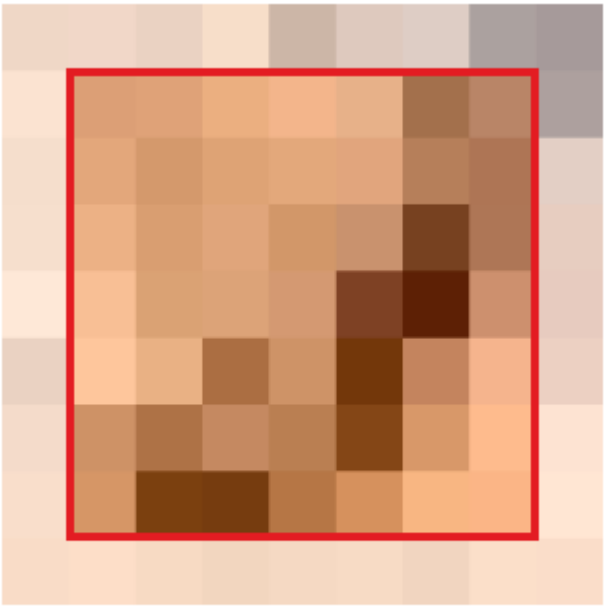
```
planarSourceBuffers.separableConvolve(horizontalKernel: kernel,
                                      verticalKernel: kernel,
                                      edgeMode: .extend,
                                      destination: planarDestinationBuffers)
```

The increase in speed from using two 1D kernels instead of a single 2D kernel is significant. For each pixel, the 2D pass requires `M * N` multiplications and additions (where `M` is the number of rows and `N` is the number of columns), but each 1D pass only requires `M + N` multiplications and additions.

## Blur an image with high-speed kernels

vImage provides two high-speed blurring convolutions for 8-bit images: a box filter and a tent filter. These blurs are equivalent to convolving with standard kernels, but the developer doesn't need to supply the kernel. These functions are typically faster than performing an equivalent convolution with custom kernels.

The box filter returns the average pixel value in a rectangular region that surrounds the transformed pixel.



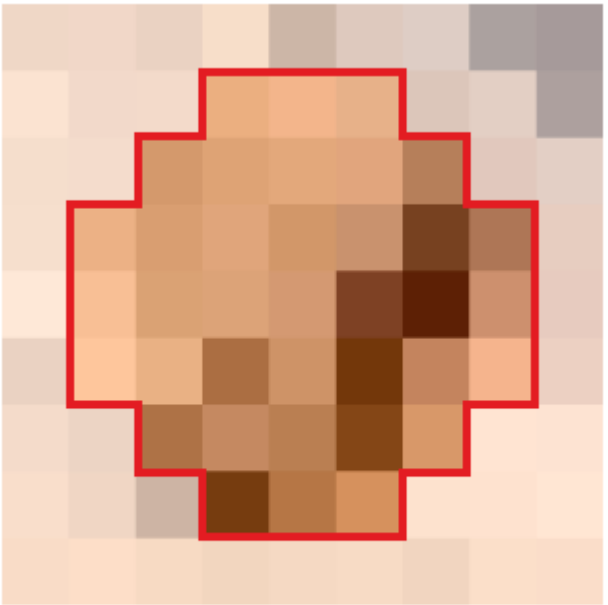
Box filter

This sample calls `boxConvolve(kernelSize:edgeMode:destination:)` to apply a box filter to an image.

```
sourceBuffer.boxConvolve(kernelSize: .init(width: kernelLength,
                                           height: kernelLength),
                         edgeMode: .extend,
                         destination: destinationBuffer)
```

Although the box filter is the fastest blur, the following example shows how it suffers from rectangular artifacts:

The tent filter returns the weighted average of pixel values in a circular region that surrounds the pixel that vImage is transforming. *Weighted average* means that the influence of pixels on the result decreases the further they are away from the transformed pixel.



Tent filter

The sample calls `tentConvolve(kernelSize:edgeMode:destination:)` to apply a tent filter to an image.

```
sourceBuffer.tentConvolve(kernelSize: .init(width: kernelLength,
                                            height: kernelLength),
                          edgeMode: .extend,
                          destination: destinationBuffer)
```

The following example shows the result of a tent filter. The result is a smoother blur, at the expense of being slightly slower to execute than the box filter.

Note that passing the `vImage.EdgeMode.truncateKernel` flag to the high-speed kernels can significantly impact their performance. This flag is only necessary when vImage needs to restrict calculations to the portion of the kernel overlapping the image.

## Blur an image with multiple kernels

vImage can apply multiple kernels in a single convolution. The `convolve(with:divisor:bias:edgeMode:destination:)` function makes it possible to specify four separate kernels — one for each channel in the image.

When using multiple kernels to apply image filters, vImage can operate on the red, green, blue, and alpha channels individually. For example, it can use multiple-kernel convolutions to resample the color channels of an image differently to compensate for the positioning of RGB phosphors on the screen. Because each of the four kernels can operate on a single channel, the vImage multiple-

kernel convolution functions are available only to the interleaved image formats, `vImage.Planar8x4` and `vImage.PlanarFx4`.

The four kernels for the convolution filters need to be the same size, but can accept padding with zeros to simulate smaller kernels. vImage is able to optimize individual passes, effectively cropping the zero padding.

The following code creates an array of four kernels, each containing a central circle of 1s of decreasing size:

```
let radius = kernelLength / 2
let diameter = (radius * 2) + 1

let kernels: [vImage.ConvolutionKernel2D<Int16>] = (1 ... 4).map { index in
    let weights = [Int16](unsafeUninitializedCapacity: diameter * diameter) {
        buffer, initializedCount in
        for x in 0 ..< diameter {
            for y in 0 ..< diameter {
                if hypot(Float(radius - x), Float(radius - y)) < Float(radius / inde
                    buffer[y * diameter + x] = 1
                } else {
                    buffer[y * diameter + x] = 0
                }
            }
        }

        initializedCount = diameter * diameter
    }

    return vImage.ConvolutionKernel2D(values: weights,
                                      size: .init(width: kernelLength,
                                                  height: kernelLength))
}
```

For example, with a kernel length of 17, the first three kernels from the code above contain the following values:

The `convolve(with:divisor:bias:edgeMode:destination:)` performs the convolution.

```
let divisors = kernels.map { return Int32($0.values.reduce(0, +)) }
```

```
sourceBuffer.convolve(with: (kernels[0], kernels[1], kernels[2], kernels[3]),
                      divisors: (divisors[0], divisors[1], divisors[2], divisors[3]),
                      edgeMode: .extend,
                      destination: destinationBuffer)
```

The example below shows the result of the multiple-kernel convolution. Note the color-fringing effect from applying different kernels to the different color channels.

---

# See Also

## Convolution and Morphology

{} Adding a bokeh effect to images

Simulate a bokeh effect by applying dilation.

Convolution

Apply a convolution kernel to an image.

Morphology

Dilate and erode images.