Sample Code

Reducing artifacts with custom resampling filters

Implement custom linear interpolation to prevent the ringing effects associated with scaling an image with the default Lanczos algorithm.

Download

macOS 13.0+ | Xcode 14.3+

Overview

Most vImage geometry operations, such as scale and rotate, use a process known as *resampling* to prevent image artifacts. vImage resamples with kernels that combine data from a target pixel and other nearby pixels to calculate a value for the destination pixel.

Because resampling involves evaluating the kernel at fractional pixel locations, the process relies on a family of kernel matrices for use at different fractional distances through a given pixel. This sample code app provides a function that generates this family of kernels – unlike operations such as convolution and morphology, which apply a single kernel matrix at the center of each pixel.

For most vImage geometric operations, vImage supplies a default resampling filter that is an implementation of the Lanczos resampling method. However, the Lanczos method can produce ringing effects near regions of high-frequency signals (that is, regions that contain a lot of pixel variation, such as the hard edges typical of line art). To correct this, this sample code app implements a linear interpolation as a custom resampling filter.

Declare the resampling filter

This app allows the user to toggle between the default resampling filter (Lanczos) and the custom resampling filter. The code declares the filter independently of initialization to support that functionality:

```
let resamplingFilter: ResamplingFilter

let scale: Float = 30
```

The following code initializes a default Lanczos resampling filter:

```
resamplingFilter = vImageNewResamplingFilter(scale,
                                             vImage_Flags(kvImageHighQualityResampli
```

On return, `resamplingFilter` is an initialized Lanczos resampling filter with the specified scale factor.

Use shear operations to scale an image

The vImage shear functions accept the resampling filter and perform the scaling. The shear functions operate in one dimension at a time, so to scale an image in both dimensions, the sample code calls `shear(direction:translate:slope:resamplingFilter:backgroundColor:destination:)` twice. The first call passes `vImage.ShearDirection.vertical`, and the second call passes `vImage.ShearDirection.horizontal`.

Because these functions don't work in place – that is, they require separate input and output buffers – the code uses an intermediate buffer to pass data from the vertical shear to the horizontal shear.

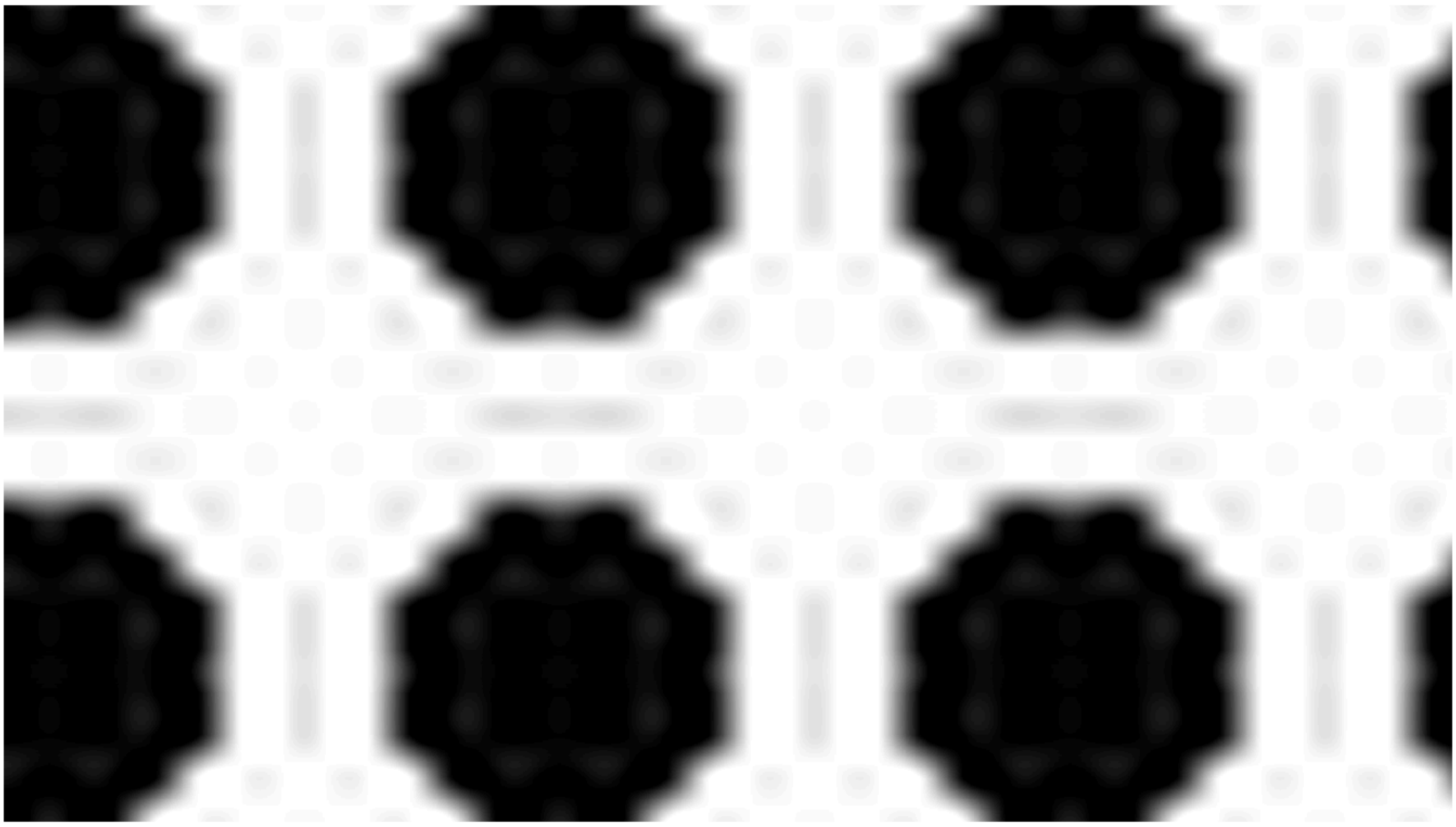
```
let height = Float(sourceBuffer.height)
let yTranslate = (height - height * scale) * 0.5

sourceBuffer.shear(direction: .vertical,
                   translate: yTranslate,
                   slope: 0,
                   resamplingFilter: resamplingFilter,
                   destination: intermediateBuffer)

let width = Float(sourceBuffer.width)
let xTranslate = (width - width * scale) * 0.5

intermediateBuffer.shear(direction: .horizontal,
                         translate: xTranslate,
                         slope: 0,
                                    resamplingFilter: resamplingFilter,
                                    destination: destinationBuffer)
```

On return of the horizontal shear function, `destinationBuffer` contains the source image, scaled about its center. The following shows an image of several small dots, magnified 30 times using the Lanczos resampling filter:



The ringing artifacts appear as faint lines between the magnified dots.

Write a linear resampling filter function

The shear functions that scale an image are both 1D, and therefore the resampling filter function the sample code project creates is also 1D. The code applies the same filter function for both the vertical and horizontal passes.

The function generates a set of kernel values based on a set of distances that the pixel being transformed supplies –- read from `inPointer`. The system assigns the generated kernel values to `outPointer`.

In the following example, the kernel values are inversely proportional to the distance; the further a pixel is from the transformed pixel, the smaller the corresponding kernel value. After calculating the kernel values, the values *scale* (normalize) so that their sum is `1.0`. This normalization step ensures the final image is the same brightness as the original.

```
func kernelFunc(inPointer: UnsafePointer<Float>?,
                outPointer: UnsafeMutablePointer<Float>?,
                count: UInt,
                userData: UnsafeMutableRawPointer?) {
    if let inPointer = inPointer, let outPointer = outPointer {
        let absolutePixelPositions =
        Array(UnsafeBufferPointer(start: inPointer,
                                  count: Int(count))).map {
            abs($0)
        }

        let kernelValues = absolutePixelPositions.map {
            (absolutePixelPositions.max()! - $0)
        }

        let divisor = vDSP.sum(kernelValues)
        let normalizedKernelValues = vDSP.multiply(1 / divisor, kernelValues)

        outPointer.update(from: normalizedKernelValues,
                          count: Int(count))
    }
}
```

For example, if the system passes the following pixel positions to `inPointer`:

```
[-3.0, -2.0, -1.0, 0.0, 1.0, 2.0, 3.0, 4.0]
```

The values in the `kernelValues` array are:

```
[1.0, 2.0, 3.0, 4.0, 3.0, 2.0, 1.0, 0.0]
```

Dividing each of the values in `kernelValues` by its sum returns the normalized kernel values that the code assigns to the resampling function's `outPointer`:

```
[0.0625, 0.125, 0.1875, 0.25, 0.1875, 0.125, 0.0625, 0.0] // sum = 1
```

The values that the resampling function generates form a 1D convolution kernel that the shear functions use in a similar way to the 1D convolution described in Blurring an image. However, unlike the kernels used for convolution, the resampling kernel is suitable for use with fractional pixel positions.

Allocate the resampling filter function memory

The resampling function, the scale factor, and the kernel width combine to determine the memory that the resampling function requires. The sample code uses the `vImageGetResamplingFilterSize(_:_:_:_:)` function to calculate the size in bytes, and the `allocate(byteCount:alignment:)` function to allocate the necessary memory.

```
let kernelWidth: Float = 1.5


let size = vImageGetResamplingFilterSize(scale,
                                          kernelFunc,
                                          kernelWidth,
                                          vImage_Flags(kvImageNoFlags))


resamplingFilter = ResamplingFilter.allocate(byteCount: size,
                                              alignment: 1)
```

On return, `resamplingFilter` is a `ResamplingFilter` structure, allocated with the correct amount of uninitialized memory.

Create a linear resampling filter

The sample code calls `vImageNewResamplingFilterForFunctionUsingBuffer(_:_:_:_:_:_:)` to create the resampling filter and populate `resamplingFilter`.

```
vImageNewResamplingFilterForFunctionUsingBuffer(resamplingFilter,
                                                scale,
                                                kernelFunc,
                                                kernelWidth,
                                                nil,
                                                vImage_Flags(kvImageNoFlags))
```

Scaling using a custom resampling filter is the same process as using the default Lanczos resampling:

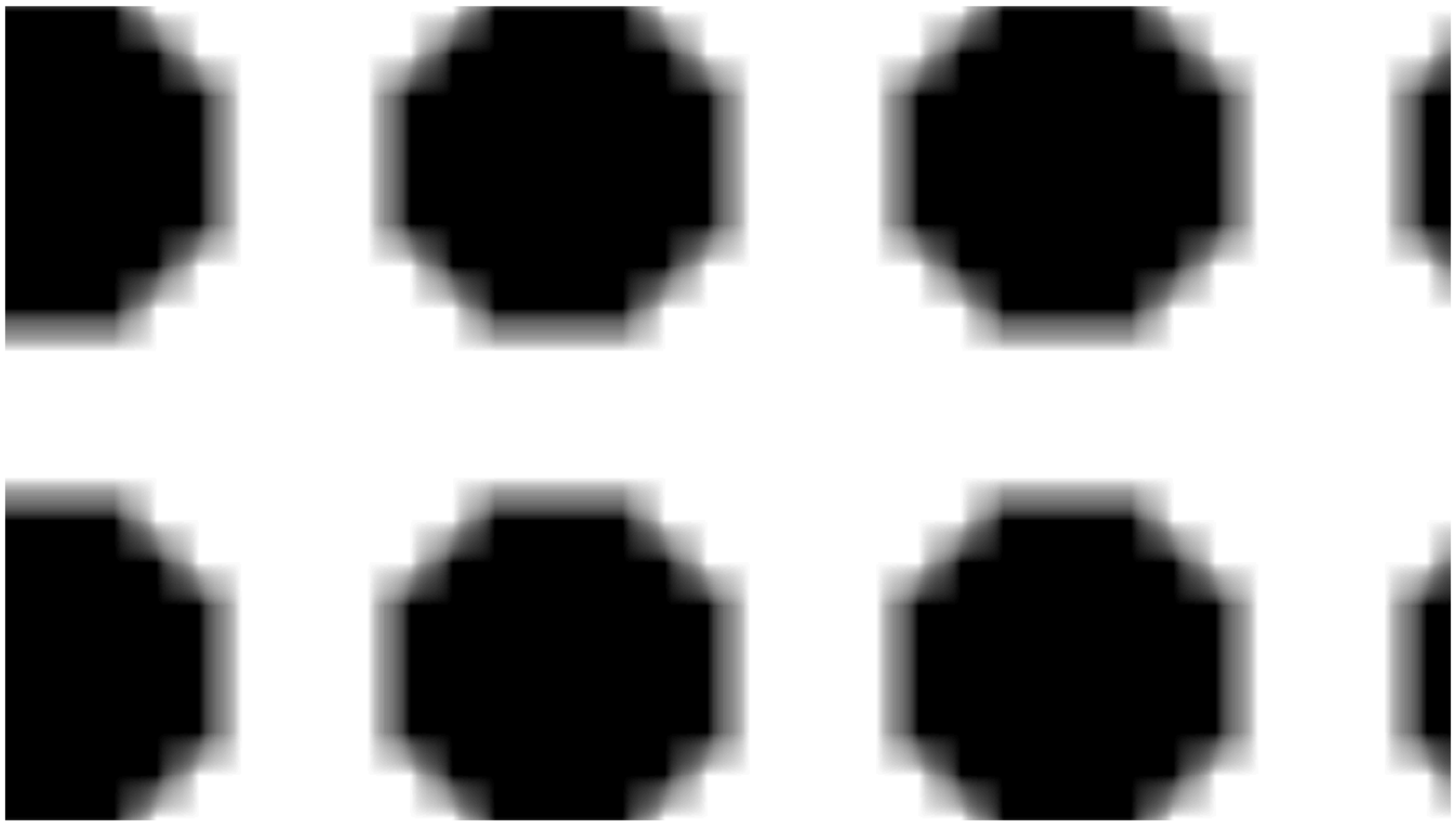
```
let height = Float(sourceBuffer.height)
let yTranslate = (height - height * scale) * 0.5


sourceBuffer.shear(direction: .vertical,
                   translate: yTranslate,
                   slope: 0,
                   resamplingFilter: resamplingFilter,
                   destination: intermediateBuffer)

let width = Float(sourceBuffer.width)
let xTranslate = (width - width * scale) * 0.5

intermediateBuffer.shear(direction: .horizontal,
                         translate: xTranslate,
                         slope: 0,
                         resamplingFilter: resamplingFilter,
                         destination: destinationBuffer)
```

The following shows the same image that the Lanczos example uses, also maginifed 30 times.



Linear resampling eliminates the ringing artifacts.

Free the resampling filter memory

After the sample app is finished working with the resampling filter, it's important that it frees the allocated memory. This process will vary depending on whether the code has used the default or a custom filter. The following code frees the memory for the default:

```
vImageDestroyResamplingFilter(resamplingFilter)
```

The following code frees the memory for custom resampling filters:

```
resamplingFilter.deallocate()
```

See Also

Image Resampling

Resampling in vImage

Learn how vImage resamples image data during geometric operations.

Image shearing

Shear images horizontally and vertically.