Sample Code

Using vImage pixel buffers to generate video effects

Render real-time video effects with the vImage Pixel Buffer.

Download

macOS 13.0+ | Xcode 14.3+

Overview

This sample code project captures video from a macOS device's camera and applies video effects in real time. The sample converts the 8-bit YpCbCr video frames to 32-bit RGB `vImage.PixelBuffer` images and demonstrates image-processing techniques that are available only for 32-bit data.

Before exploring the code, build and run the app to familiarize yourself with the different visual results the app generates from the camera.

Create the any-to-any converter

The code creates a `vImageConverter` instance that converts the YpCbCr video frames to three-channel, 32-bit-per-channel, floating-point interleaved image data.

```
let cvImageFormat = vImageCVImageFormat.make(
    format: .format422YpCbCr8,
    matrix: kvImage_ARGBToYpCbCrMatrix_ITU_R_601_4.pointee,
    chromaSiting: .center,
    colorSpace: CGColorSpaceCreateDeviceRGB(),
    alphaIsOpaqueHint: true)!

let cgImageFormat = vImage_CGImageFormat(
    bitsPerComponent: 32,
    bitsPerPixel: 32 * 3,
    colorSpace: CGColorSpaceCreateDeviceRGB(),
    bitmapInfo: CGBitmapInfo(
        rawValue: kCGBitmapByteOrder32Host.rawValue |
        CGBitmapInfo.floatComponents.rawValue |
        CGImageAlphaInfo.none.rawValue),
    renderingIntent: .defaultIntent)!

lazy var converter: vImageConverter = {
    guard let converter = try? vImageConverter.make(
        sourceFormat: cvImageFormat,
        destinationFormat: cgImageFormat) else {
        fatalError("Unable to create converter")
    }

    return converter
}()
```

Convert a Core Video pixel buffer to RGB

The code defines `destinationBuffer` as a `vImage.InterleavedFx3` pixel buffer. The conversion function creates a `vImage.DynamicPixelFormat` source buffer that references the locked `CVPixelBuffer` instance and passes that to the any-to-any converter.

```
func populateDestinationBuffer(pixelBuffer: CVPixelBuffer) {

    let sourceBuffer = vImage.PixelBuffer(
        referencing: pixelBuffer,
        converter: converter,
        destinationPixelFormat: vImage.DynamicPixelFormat.self)

    do {
        try converter.convert(
            from: sourceBuffer,
            to: destinationBuffer)
    } catch {
        fatalError("Any-to-any conversion failure.")
    }
}
```

On return, `destinationBuffer` contains the RGB representation of the YpCbCr video frame.

Apply the noise effect

The sample simulates noise or film grain by adding Gaussian noise (with a mean of zero) to each frame. The image below shows an example of the noise effect:

Accelerate's BNNS library provides the `BNNSRandomFillNormalFloat( : : : :)` function that fills an array descriptor with random floating-point values mapped to a normal distribution.

Use the `withUnsafeMutableBufferPointer(_:)` function to pass a pointer to the pixel buffer's underlying data to a `BNNSNDArrayDescriptor`.

The following code generates the noise effect:

```
func applyNoise() {

    noiseBuffer.withUnsafeMutableBufferPointer { noisePtr in

        if var descriptor = BNNSNDArrayDescriptor(
            data: noisePtr,
            shape: BNNS.Shape.tensor3DFirstMajor(
                noiseBuffer.width,
                noiseBuffer.height,
                noiseBuffer.channelCount)) {

            /// Fill `noiseBuffer` with random values mapped to a normal distributio
            /// of `0` and a standard deviation of `0.125`.
            let mean: Float = 0
            let stdDev: Float = 0.125

            BNNSRandomFillNormalFloat(
                randomNumberGenerator,
                &descriptor,
                mean,
                stdDev)
        }
    }

    /// Fill `mutableDestinationPtr` with the sum of the corresponding pixels
    /// in `destinationBuffer` and `noiseBuffer`.
    destinationBuffer.withUnsafeMutableBufferPointer { mutablDestinationPtr in

        vDSP.add(destinationBuffer, noiseBuffer,
                 result: &mutablDestinationPtr)
    }
}
```

Apply the temporal blur effect

The temporal blur effect blurs the image over time by calculating a weighted average of the current frame and previous frames. The effect is analogous to an exaggerated motion blur.

The image below shows an example of a rotating image with the temporal blur effect:

The `linearInterpolate(bufferB:interpolationConstant:destination:)` function calls the vDSP function `vDSP_vintb` to calculate the linear interpolation between the current frame and the previous interpolated frame.

The following code generates the temporal blur effect:

```
func applyTemporalBlur() {

    let interpolationConstant: Float = 0.925

    destinationBuffer.linearInterpolate(
        bufferB: temporalBuffer,
        interpolationConstant: interpolationConstant,
        destination: temporalBuffer)

    temporalBuffer.copy(to: destinationBuffer)
}
```

Apply the posterization effect

The posterization effect reduces the continuous colors of an image to fewer tones. The effect produces results with regions of solid colors. The image below shows an example of the posterization effect:

The sample generates the posterization effect using histogram specification. The code achieves the reduced color count by calculating and specifying a histogram that has a low bin count. For more information about histogram specification, see Specifying histograms with vImage.

The code populates a multiple-plane pixel buffer from the interleaved destination buffer. The multiple-plane pixel buffer contains three discrete planar buffers, and the `vImage.PixelBuffer.Histogram888` function returns the histogram for the individual red, green, and blue channels. Specifying a bin count of `4` returns a result that contains a maximum of `4 * 4 * 4` (64) colors.

The following code generates the posterization effect:

```
func applyPosterization() {

    destinationBuffer.deinterleave(
        destination: histogramBuffer)

    let histogram = histogramBuffer.histogram(
        binCount: 4)

    histogramBuffer.specifyHistogram(
        histogram,
        destination: histogramBuffer)

    histogramBuffer.interleave(
        destination: destinationBuffer)
}
```

Apply the color threshold effect

The color threshold effect is similar to the posterization effect, but reduces each color channel to a single-bit, so each color is either `0` or `1`. The image below shows the color threshold effect:

The `colorThreshold( :destination:)` function sets pixel values equal to or greater than the specified threshold to 1 and other pixel values to `0`. Because the function works over the individual red, green, and blue values, the result contains a maximum of 2 * 2 * 2 (8) colors. The effect is identical to the posterization effect with `binCount` set to 2.

The following code generates the color threshold effect:

```
func applyColorThreshold() {

    let threshold: Float = 0.5

    destinationBuffer.colorThreshold(
        threshold,
        destination: destinationBuffer)
}
```

See Also

Core Video Interoperation

{} Integrating vImage pixel buffers into a Core Image workflow

Share image data between Core Video pixel buffers and vImage buffers to integrate vImage operations into a Core Image workflow.

{} Applying vImage operations to video sample buffers

Use the vImage convert-any-to-any functionality to perform real-time image processing of video frames streamed from your device's camera.

{} Improving the quality of quantized images with dithering

Apply dithering to simulate colors that are unavailable in reduced bit depths.

Core Video interoperability

Pass image data between Core Video and vImage.